Politecnico di Torino

Master of Science Degree in MECHATRONIC ENGINEERING

Master Thesis

UAV/UGV control and navigation with smartphone

Supervisor:
prof. Marcello Chiaberge

Candidate:
Matteo Masuelli
S267486

27 July 2021

Abstract

This thesis project mainly concerns the use of a smartphone for controlling the flight of a drone.
Modern smartphones are mobile devices that combines the typical cell phone operations with a mobile operating system. They are equipped with many hardware components and implement a lot of software, internet and multimedia functionalities. They also contain sensors and systems that can be used for estimating some states of the mobile device, such as its orientation.
Considering that a smartphone is equipped with many sensors and components that are usually present on a drone, it would be possible to create a drone using a smartphone and a reduced number of additional items.
The smartphone should measure its position and orientation and implement a software controller in order to stabilize the drone vehicle and move it to a specific position. Its internet connection capabilities could also be used for realizing a communication link with a control station.
In order to send the proper signals to the circuits that actuate the motors and implement a reliable radio control link for sending commands to the drone, it is necessary to use an additional electronic component. In this case an Arduino board is used.

This project is based on the previous thesis "Implementation of an autopilot for UAV/UGV system based on Android smartphone" realized by Eros Giorgi.
The main purpose of this thesis is to analyze and extend the job made in the previous project in order to implement an autopilot software on an Android smartphone and achieve the flight stability of the drone. It is needed to discover how the smartphone samples data from its sensors and if it is suitable for performing a task that usually is accomplished with a different type of hardware and software. At the same time it is also important to improve the software running on the Arduino board, reducing as much as possible the delays in the overall system.
In the end it is necessary to extend the funtionalities that work over the communication link between the drone and the ground control station and implement an altitude and a position control.

Acknowledgements

Prima di procedere con la trattazione, mi sembra doveroso dedicare qualche riga a coloro che mi hanno supportato nel corso degli anni accademici.
Il ringraziamento più importante è dedicato alla mia famiglia. I miei genitori sono stati fondamentali non solo per il supporto emotivo e morale, ma soprattutto per il sostegno economico.
Ringrazio i miei amici con i quali ho trascorso preziosi momenti di svago.
Infine desidero ringraziare il mio relatore Marcello Chiaberge ed i ragazzi del PIC4Ser, in particolare Gianluca che è sempre stato disponibile nei momenti in cui avevo bisogno di aiuto.

Contents

List of Figures

Chapter 1

Introduction

A “UAV” (unmanned aerial vehicle) [1] is a aircraft designed and realized to accomplish some tasks in a totally or partially autonomous way. They are commonly known also as drone.
The main interesting characteristic of this type of vehicles is that they are able to fly and carry out some operations without a pilot on board. Drones are often driven by onboard systems commonly called autopilots but they can also be manually controlled by a human operator on the ground. It is possible to adopt a mixed strategy where some operations are done automatically, while other tasks are accomplished by a human operator on the ground.
Originally UAV vehicles were employed mainly in the military field where they were used in order to perform some dangerous operations. In the last years they started to be used also for other applications in civil and commercial fields, such as product delivery, agriculture, aerial photography and surveillance.

1.1 State of Art

1.1.1 Quadcopter

Regarding the UAV vehicles used by civilians, today the quadcopter [2] or quadrotor type is the most popular.
Their small size and low inertia allow them to be more easily controlled respect to other kind of UAV systems.
During the last years, advances in the electronics brings cheaper, lighter and smaller flight controllers, sensors, positioning systems and cameras. Quadcopter started to be equipped with these new components, becoming less expensive and also more accessible to most of people. As a result, today drones can be very lightweight and small, so they have a low kinetic energy. This features improve their maneuverability and reduce their ability to cause damages to people. They could also be used not only in a outdoor environment, but indoor too.

Mechanical structure

A quadrotor UAV is composed of a mechanical structure, which includes a frame, wings and motors. In order to actuate the motor, it is usually necessary to use a ESC (electronic speed controller). The combination of motor and wing can also be addressed as a rotor.
Typically a quadcopter can have two possible configuration: a “X” configuration and a “+” configuration. The “X” configuration is considered to be more stable, while the “+” is more used for acrobatic fly. In our project we will focus on the "X" configuration.
Quadcopters usually have two rotor spinning counterclockwise and two clockwise. Each rotor produces a lift force and the contribution of forces from all the four rotor allows the quadcopter to hover or change its altitude counterbalancing the gravitational force. In theory during the hover movement all the four rotors spin at the same angular velocity, but it does not happen because it is necessary to stabilize the orientation of the drone. In order to change the orientation of the quadcopter it is necessary to change the spinning velocity of only some rotors respect to the others. According to figure 1.2, it is needed to increase the speed of motors 2 and 3 respect to the speed of motor 1 and 4 in order to have a positive direction roll movement about the X reference axis and viceversa to obtain a

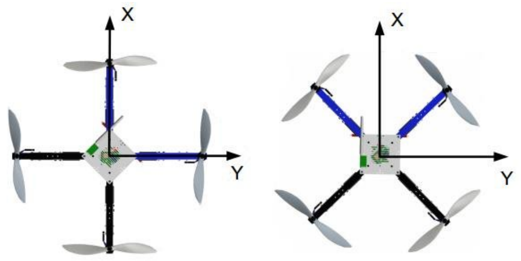
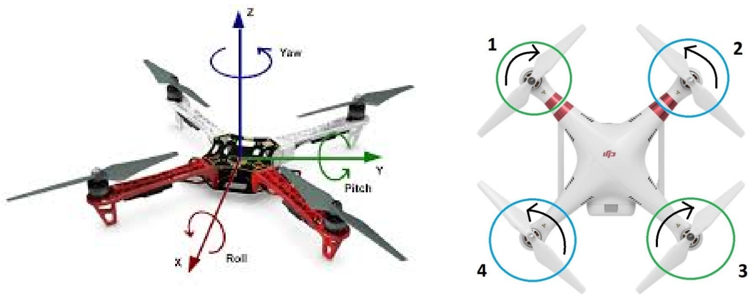


Figure 1.1: "+" and "X" quadcopter configurations

negative direction roll movement. The same principle can be applied for a pitch movement: increasing the speed of motors 1 and 2 respect to the speed of motors 3 and 4 allows the drone to carry out a positive pitch movement and viceversa to have a negative direction pitch movement. On the other hand the yaw movement can be achieved by modifying the velocity of two motors that spins in the same direction. It means rising the speed of motors 1 and 3 respect to the speed of motors 2 and 4 or viceversa.



(A) (B)

Figure 1.2: Roll, pitch and yaw movements

Sensors

It is always necessary to measure some quantities about the state of the drone, which obviously must be equipped with some sensors. The quantity and the quality of sensors present on the board can be referred with the Degree Of Freedom (DOF). 6 DOF implies a 3-axis gyroscope and accelerometer, which corresponds to a typical IMU (inertial measurement system). 9 DOF means an IMU plus a compass or magnetometer, 10 DOF adds a barometer and 11 DOF implies the usage of a GPS receiver. All the raw data obtained by a sensor must be fused in order to obtain a good estimation of a particular state of the quadcopter, such as its attitude or position.

Autopilot

The autopilot or flight stack is the UAV software that uses the data collected by the sensors and applies a proper control strategy in order to send the appropriate commands to the motors, ensuring the flight stability of the drone and the execution of the necessary tasks and duties.
The most common control algorithm is the PID.
Most of the available flight stack provides open source software. This characteristic implies that the the software is not only accessible to everyone, but it is also possible to change it or customize it for executing specific tasks.
In order to have a stable UAV vehicle the autopilot software is required to be real-time because the system has to have a rapid response to the changing sensor data. As a matter of fact a drone is a system that is intrinsically unstable. Errors or delays in the system could easily lead the system to become unstable, causing the fall and the crash of the drone.

Communication

Most of UAV vehicles are equipped with software and hardware used for the communication tasks, usually controlled by a ground control station (GCS). There are different kind of communication link that can implemented. In general the control link is used to send commands to the drone in order to manage its movements. It is normally implemented with a radio link. The control link must be reliable and with low delays. A secondary control link can be implemented using a satellite communication that is less reliable respect to the radio link. It is important because it can be used if the primary control link fails. In general the ground control station implements also a data link, which can be used for the exchange of telemetry data containing important informations about the condition of the UAV vehicle, video streaming and other functional data.

1.2 Objective

Continuing the job done and described in the thesis “Implementation of an autopilot for UAV/UGV system based on Android smartphone”[3], the principal objectives of this thesis are:

- analyze the Android operating system with a particular focus on the sensor data acquisition. Android is not a real time operating system, so in theory it would not be suitable for accomplish a drone autopilot tasks. As a result, it is necessary to discover if the Android operating system injects some delays, trying to reduce them or mitigate their effects in order to have a working autopilot flight software running on an Android application. Moreover most of control algorithms are based on the fact that the sampling rate is constant, so it would be appropriate to investigate if it is possible to achieve a constant sampling rate on Android
- improve the autopilot Android application, reducing the delays and discovering possible bugs
- improve the Arduino software that operates as an interface between the Android application, motors and the human operator. In particular it is necessary to find an optimal way to handle all the necessary operations without slowing down the system

All these steps must be taken with the main intent of obtaining the flying stability of a UAV quadcopter controlled by an autopilot software running on an Android application.
In addition it is necessary to improve the communication link with the ground control station and the quality of video streaming and photos without causing problems to the autopilot tasks. The final goal would be to completely replace the usual autopilot software and hardware normally placed on a drone with only an Android smartphone and all the other components necessary for the actuation of the motors and the communication with a human operator.

1.3 Summary

The main goal of this thesis project is to realize a drone whose autopilot software runs on an Android smartphone.
The first chapter contains an overview of the Android operating system, with a description of all the software layers that, put all together, form the Android operating system. Android runs on many different devices and one of its main intents is to virtualize the underlying hardware components. Using this approach the Android application developing becomes easier and application developers do not have to worry about the hardware and internal system components and architecture. In addition Android operating system implements internally a lot of functionalities and makes them available to the developer as services. For example, Android implements internally the appropriate computations needed for directly providing the attitude measurements of the device. In general Android makes available a lot of classes and functions that can be used to structure an application at a high level, but this approach has the drawback of making difficult to access some low level system components. Anyway it is possible to access some low level software functionalities writing an application in native code, but they are mainly related to multimedia, graphical and pure computational tasks.
In the second chapter the main focus are the sensors, how Android provides sensor data, the sensors sampling rate and sensors latency. Considering that Android is not a real time operating system and sensor data are made available by system services, the sensor sampling rate is not perfectly constant and sensor latency is not zero. A new sensor sampling modality Android has just introduced

is described, even if it cannot be currently implemented. This new modality should provide faster sampling rate and lower latencies.
The following chapter contains a description of the Android application implemented. The application is divided in many different Java classes, each one focused on a specific functionality. On the other side it is possible to see the application as a group of different threads that run in parallel, where the most important one is responsible of implementing the control algorithm in order to obtain the appropriate motor commands and stabilyze the flight.
In the fourth chapter the main focus is the Arduino board. A description of all its funtionalities and how they are implemented is provided. Arduino board represents an interface between the Android smartphone, ESCs circuits that control the motors and the radio controller that is used for sending the control commands to the drone. The SBUS protocol is used for receiving commands from the radio link and the I2C protocol is used for sending the appropriate motor commands to the ESCs. On the other side the communication with the Android smartphone is realized through a USB connection.
In the fifth chapter the communication between the Android application and a GCS(Ground Control Station) is described. In this project the GCS is a computer running QGroundControl, a software that is usually adopted for implementing the communication between a GCS and different autopilot softwares normally used on drones. The communication channel is implemented using the MavLink communication protocol working over a TCP connection. It is mainly used for sending telemetry data to the GCS and implementing a mission planner. In addition it is also possible to set or change some parameters of the autopilot software from the GCS. A video-streaming and a photography functionality are implemented too. The communication link can work over a WiFi network or a mobile network.
In the following chapter the hardware used in order to realize a working prototype of the drone is shown. In the end the experimental results and the conclusions are presented.

Chapter 2

Android OS

Figure 2.1: Current Android logo

Android [4] is a mobile operating system, mainly designed for touchscreen devices, like smartphone and tablet. Android is developed by a consortium of developers called Open Handset Alliance and it is commercially sponsored by Google. It is free and open-source software, even if most of devices contain additional proprietary software pre-installed, like the well-known Google Play Store. From the Google Play Store users can download software packages, commonly called applications, and install them on their own Android devices. Android was presented in November 2007 and the first commercial Android devices was launched in September 2008. It was an HTC Dream device.

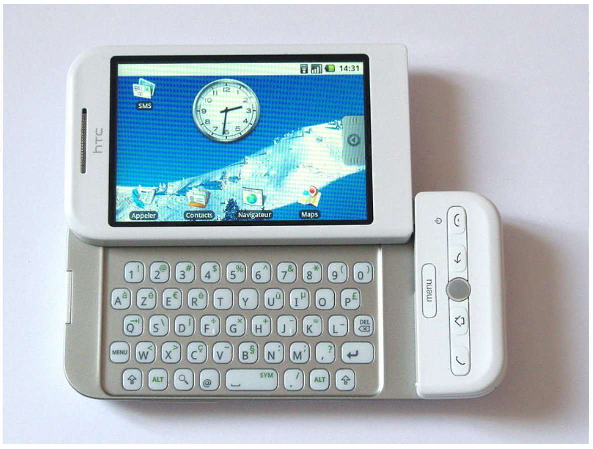


Figure 2.2: HTC Dream

Several variants of Android have been developed for different types of devices, including, for example, Android TV for televisions and Wear OS for wearable devices such as wrist watches. Anyway Android operating system is mainly installed on smartphone and today is the most used operating system on phone.

2.1 Android architecture

Android system is composed of many different parts [5]. Different components belong to different layers of the system, going from the upper application framework layer to the lower Linux kernel layer. A lower layer is more related to the device's hardware components, while the upper layers are more related to the Android applications used by the final user.

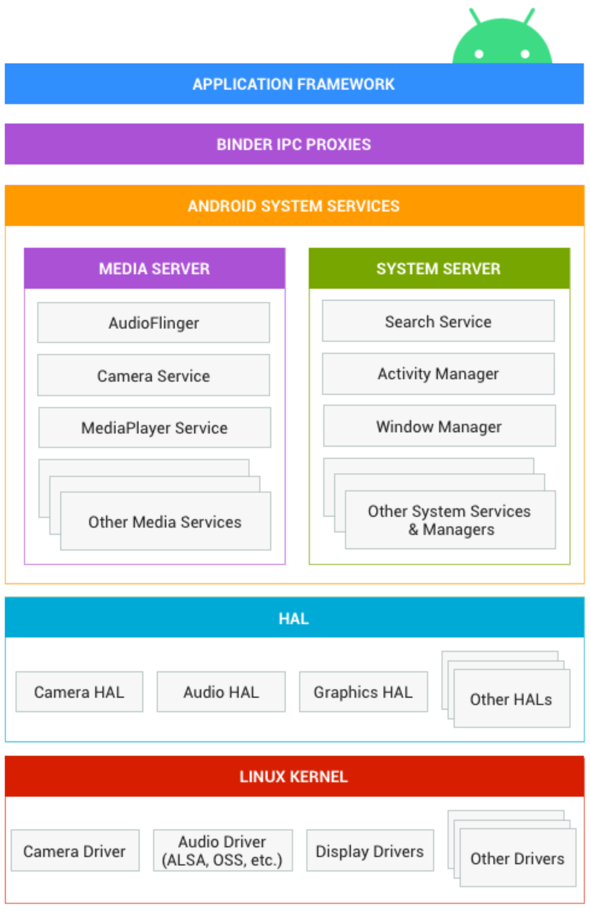


FIGURE 2.3: Android system architecture

2.1.1 Application framework

Application framework is formed by Java APIs, which are the building blocks needed to create an Android application [6], that can be written using Java, C++ or Kotlin programming languages. Android developers have full access to these APIs through the software development kit (SDK). The Android SDK compiles the code along with all the needed data and files, generating an APK (Android Package) file. The APK file is an archive file containing all the resources an Android specific device needs in order to install the application.
Each Android application lives in an isolated enviroment. The Android operating system is a multi-user Linux system and each application is a different user, which normally can access only its private resources.

An Android application can be composed of 4 different components:

- Activity
- Service

- Broadcast receiver
- Content provider

Each component has a different function and behaves in a different way respect to the others.

Activity

An activity [7] represents a single screen with a user interface and it is the main responsible of the interaction between the Android application and the user. In the mobile Android enviroment an application does not have one single entry point. In other words there are many different ways for launching an application. For example an application can be launched manually by the user clicking on its icon or it can also be launched by another application. The activity component is built in order to facilitate this kind of behaviour. Anyway an activity is not intended to continuosly provide data because it can be easily replaced by another activity when the screen is changed. After this thing happens, it can be easily sent to sleep by the operating system.
As the user navigates through, back to or out of an application, the activity components pass through several states during their lifecycles [8]. Every time the activity's state changes, a specific callback function is called. The callbacks must be implemented by the developer.

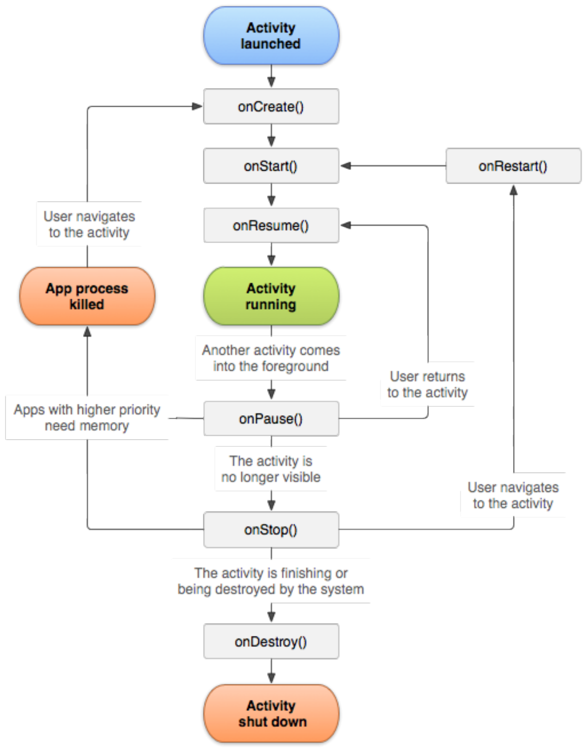

Activity launched
onCreate()
onStart()
onRestart()
User navigates to the activity
onResume()
App process killed
Activity running
Another activity comes into the foreground
User returns to the activity
Apps with higher priority need memory
onPause()
The activity is no longer visible
User navigates to the activity
onStop()
The activity is finishing or being destroyed by the system
onDestroy()
Activity shut down

FIGURE 2.4: Activity lifecycle

- Oncreate callback is called when the activity is launched and the system creates it. In this callback it is necessary to perform some basic startup operations that will be executed only once for the entire activity's lifecycle. It is also possible to frozen the state of an activity before putting the activity in the background. In this callback the frozen state can be restored.
- OnStart callback is called when the activity starts becoming visible to the user. Inside this callback the code responsible for the user interface is initialized.
- OnResume callback is called when the activity comes to the foreground and starts interacting with the user. The app stays in this condition until something takes the focus away, like, for example, another activity coming in the foreground.

- OnPause callback is called when the activity starts leaving the foreground, being replaced by another one. Inside this callback some operations that do not have to continue executing can be paused.
- OnStop callback is called when the activity is no longer visible in the foreground. In this callback some operations that do not have to run while the activity is in background could be stopped.
- OnDestroy is the final callback before the activity is destroyed. It happens when the activity finishes its task or when the system choose to close it in order to free memory space, but also when the device changes the configuration. This last thing happens, for example, when the orientation of the Android device changes. This callback is usually implemented in order to release all the resources used by the activity.

In general the operating system kills a process if it needs to free RAM memory. Concerning an activity, its probability of being killed depends on it current state. A destroyed or a background activity can be killed by the system more easily respect to an activity in the foreground.

Service

For performing long duration operations running in the background Android offers the Service component. Another component can open a connection to a service and performs interprocess communication. There are 3 different kind of services:

- Foreground services continuosly perform some operations even if the user is not interacting with the application, but their activity must be noticeable to the user displaying a notification
- Background services perform tasks that are not directly visible to the user
- Bound services are used in order to provide a client-server interface that allow different components to interact with the service exchanging some data or requests. The components may be in the same process, but they can also belong to different processes.

As for the activity component, the service lifecycle can be controlled by implementing some callback functions that are called when the service changes its state.

Content provider

Service are used in order to perform long duration tasks, but they are not suitable for hold persistent data, like storage files. Content provider components solves this problem, helping the developer in the data management part. Content providers are the standard interface used to share persistent data between processes, providing data security mechanism.

Broadcast receiver

Android application can send or receive broadcast messages sent by the Android operating system or other applications when some events occur. An application can subscribe to a specific broadcast, choosing to receive it. Then the application will be able to receive only the type of broadcast messages it subscribed to. The Android system automatically sends broadcast message when certain type of event occurs, such as when the device starts charging or the screen is turned on.

Intents

Different components of Android applications can exchange data and messages using a mechanism called Intent [9]. Intents are a form of interprocess communication and can be used to request an action from another application component. They are mainly used for 3 different thing: starting an activity, starting a service or sending broadcast messages. An intent contains a dataframe which provides informations about the components that the intent should receive , the action to be performed and secondary informations about how to carry out the requested action.

2.1.2 Binder

Binder [10] Inter Process Communication mechanism allows the application framework to cross process boundaries and call into Android system service code. In Android one process cannot normally access the memory of another process. Every application runs in its own process and is not allowed to directly access the memory of another application or system components. As a result, appropriate inter process communication procedures has been implemented in order to access the system services lower layer. This communication between the framework and the system services is hidden from the developer. However developers can implement a Binder inter process mechanism if they have to develop an application that must implement an exchange of data between a client and a service belonging to different high level applications. Binder is based on a client-server model. A client

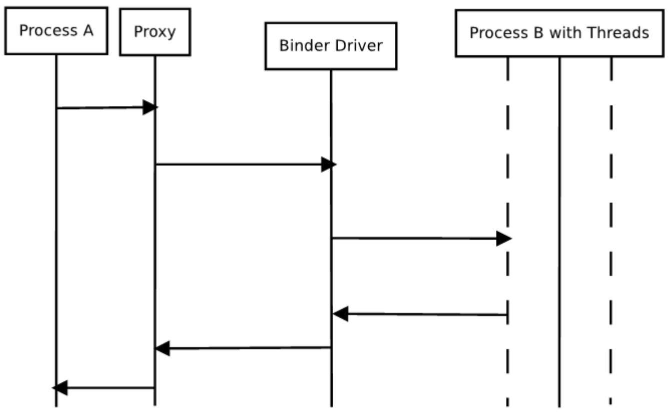


FIGURE 2.5: Binder communication

can start a communication with the server and wait for a response. The client uses a proxy for communicating with the Binder driver, which belongs to the kernel. The server has to implement a proper interface in order to accept call from the clients and implements different threads in order to handle different communications at the same time. In this way the Binder mechanism allows to call a remote method that belongs to a different process as if it would be a local method belonging to the same process[11]. In general a process act as both a server and a client.
Android Interface Definition Language (AIDL) is used for implementing interfaces. It is used for writing .aidl files that a build system translates into the Java or C++ source files that implements the classes responsible for Binder communication.

2.1.3 System services

System Services are modular components. Each one is focused on specific functionalities. The application framework gains access to system services thanks to Binder inter process communication mechanism. In this way the system services make the functionalities brought by the underlying hardware available to the application framework.
Many core Android system components and services are built with native code that required native libraries written in C and C++. The Android framework provides Java API to expose the functionalities of some of these native library. If it is necessary to develop an application using C or C++, it is possible to use Android NDK (Native Development Kit) instead of the SDK.

NDK

The NDK[12] allows to access some of the native library directly with C or C++ code. Writing applications using C or C++ through NDK platform could be useful if it is necessary to run computationally intensive operations or to achieve low latency. In general it could be used to improve the performance of an Android application.
An Android application written in C or C++ code is called native. It means that it is compiled for running on a particular hardware platform. On the other hand a program written in Java language run on a virtual machine that accepts only Java bytecode. When it is necessary to access some low

level operating system mechanisms from the Java code the Java Native Interface (JNI) must be used. Using JNI it is possible to call code written in native language from the Java code.

2.1.4 Hardware abtraction layer

Hardware Abstraction Layer (HAL) is a standard interface that hardware vendors must implement. It provides an abstraction of the lower lever driver implementation and allows Android to be agnostic respect to the them. HAL implementations are packaged into modules and loaded by the system at the appropriate time.

Treble project

With the release of Android 8, the architecture of the Android operating system has been modified carrying out a project called "Treble". In particular, the HAL Interface Definition Language (HIDL) has been introduced. HIDL separates the Android framework and the system services from the lower level software implemented by the hardware vendors. Before the release of Android 8, the interface did not provide a clear separation between the framework and the low level vendor implementation. When the Android operating system had to be updated, the vendor implementation required to be modified too. After the release of Android 8 it is not necessary to change the vendor implementation and only the Android operating system must be updated.

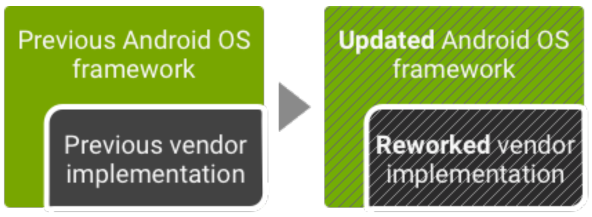

Previous Android OS framework
Previous vendor implementation
Updated Android OS framework
Reworked vendor implementation

FIGURE 2.6: Android update before version 8

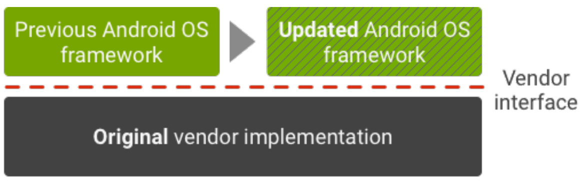

Previous Android OS framework
Updated Android OS framework
Vendor interface
Original vendor implementation

FIGURE 2.7: Android update after version 8

2.1.5 Kernel

Android uses a version of the Linux kernel with a few additions such as Low Memory Killer (a memory management system that is more aggressive in preserving memory), Wake Lock (a power managing software mechanism, used to wake up the device when it is needed), the Binder IPC mechanism and other features important for mobile embedded platform. These additions are related to system functionalities and does not affect drivers development. For this reason the driver development on Android is very similar to driver development on Linux.
Concerning an Android smartphone, the drivers are almost always realized by the hardware manufacturer and proprietary. There are some Android distributions that are almost completely open-source, but the drivers usually are proprietary. There is one distribution called Replicant that is completely open-source, but it is supported by few old devices and it has a lot of bugs and problems.
The Android operaing system is based on a Linux kernel, that was created in the 1991. Along all these years the Linux kernel has been used by a lot of users. It has been constantly improved and fixed by thousands of developers and today it is considered a trusted kernel by many corporations and security professional. As a result Android has inherited a lot of security key features from the Linux kernel.

2.2 Garbage collector

In the Android operating system the garbage collector is responsible of the dynamic memory management[13]. It keeps track of every memory allocation and reclaims the memory resources that are no longer needed because they cannot be accessed anymore.
The objects are allocated in the memory heap, which is divided into different regions. Every object belongs to a region depending on its expected life and size. For example, recently allocated objects belong to the young generation group. When an object stays active long enough, it is promoted to an older generation. Every generation region has its own upper limit on the amount of memory that can be occupied. When a generation region starts to fill up, the system runs the garbage collection operation and free up the unusued memory.
The garbage collection operations can affect the application performance. Generally it is impossible to know when a garbage collection occurs and how much time it takes to run. The system determines when the garbage collector must perform its job relying on a internal set of criteria. When criteria are satisfied, the system blocks the current process execution and starts the garbage collection. The executing Android application is literally stopped, causing delays.

2.3 Scheduler

The scheduler used by the Android operating system is the Completely Fair Scheduler(CFS). CFS has the objective of providing balance between tasks assigned to the processor and aims to maximize the overall CPU utilization, maximizing the interactive performances.
The CFS scheduler tries to guarantee a fairness in the execution time dedicated to each process. In order to accomplish this task, the scheduler needs to keeps track of the execution time dedicated to each process.
In general the priority an application gains depends on where the application is in the app lifecycle and if the application is in background or in foreground. An application in foreground gets about 95 percent of the total execution time of the device.

Chapter 3

Android sensor

Most of Android smartphone have many different built-in sensors[14]. Some of them are hardware-based and some are software-based.
Hardware-based sensors are physical component placed inside the device. They are MEMs (Micro-Electro-Mechanical system) chips and provides raw data measurements, but sometimes they can provide some logic in order to perform usual computation. In many cases different kind of sensors are present on the same chip.
Software-based sensor are not physical component, but they provides data derived from raw measurements coming from others hardware-based sensors. They can also be called virtual or synthetic sensors.
Android supports different kind of sensors, but not every Android-powered devices has every type of sensor. Each Android sensor has an official type defined in a file sensors.h in the Android source code. The type of a sensor describes how the sensors behaves and what data it provides. Most of sensors placed inside Android-powered devices are official, so they are tested and there is an official documentation.
Regarding physical sensors they can be accelerometers, gyroscopes, magnetometers, barometer, humidity, pressure, light, proximity and heart rate sensors. Also camera, fingerprint sensor, microphone and touch screen could be considered as physical sensors, but they are accessed with different specific mechanisms.

3.1 Sensor stack

Android applications can easily gain access to sensor data thanks to the Android sensor stack[15], which provides an abstraction of the sensor hardware and of the low level software necessary to use and control the hardware. The sensor stack is composed of different layers and each layer can communicate with the layer immediately above or below it.

3.1.1 Application sensor framework

An application can access sensor using SDK (Software Development Kit) APIs. They contain functions that allow a developer to discover the sensors available and acquire data from them. These functions belong to the Android sensor framework that is composed of various classes and interfaces.

- The class SensorManager[16] can be used to create an instance of the sensor service. It contains different methods used to list available sensors and register or unregister sensor event listeners. In order to obtain data from a specific sensor, it is necessary to register to a sensor event listener. The class SensorManager contains constants that can be used to set sensor parameters, such as sensor accuracy or sensor acquisition rate.
- The SensorEventListener[17] is an interface which must contain the implementation of two functions: `onSensorChanged` and `onAccuracyChanged`.
  The first is very important because it is a callback that is called every time a new sensor event containing sensor data is available. Inside the implementation of this functions, the developer should implement the code responsible of reading the sensor data. If it is necessary, it is possible

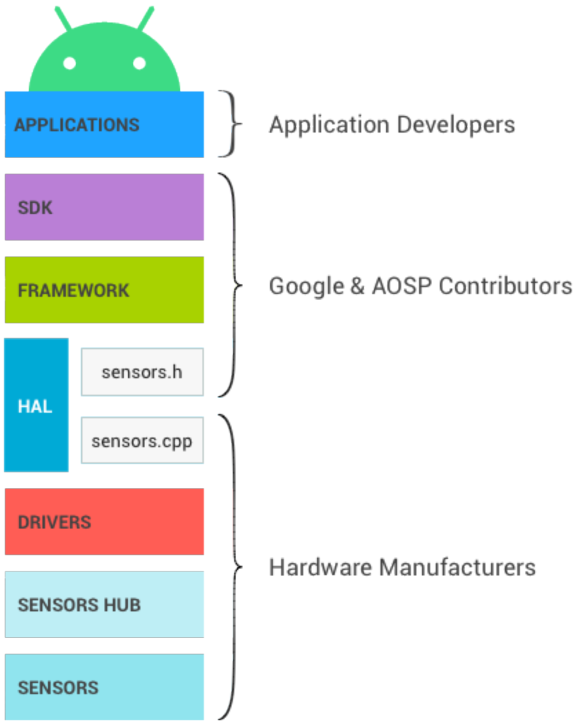


FIGURE 3.1: Android sensor stack

to carry out some operations or calculations on the sensor data just read, but it is not advisable to perform long and computationally intensive operations inside the callback. OnsensorChanged callback could be called quite often and should not be blocked in order to avoid losing of data or long delays.
OnAccuracyChanged is called every time the accuracy of the sensor changes.

- The class Sensor[18] can be used to obtain an instance of a specific sensor. This class has various methods that can be used for obtaining informations or set some parameters about a specific sensor.
- The class SensorEvent[19] is used to create an instance of a sensor event object, which contains the sensor data and information about the accuracy of the data, the type of sensor that has generated the data and the timestamp of the event.

The framework layer allows different application to gain access to the same sensor. Without it, only one single application could use a specific type of sensor.
It is important to notice that Android makes available a system sensor service the developer can register a listener to. The sensor service resides in a isolated process, so it is impossible to directly access the sensor data inside the sensor service. The sensor data can be accessed by the developer only when a specific callback function is called.

3.1.2 Sensor HAL

The sensor HAL (Hardware Abstraction Layer) is the layer between the hardware drivers and the framework. It define the functions that must be implemented to allow the framework to access sensors. The implementation is realized by the hardware manufacturer. Different versions of sensor HAL can be implemented.
With the release of Android 10, sensor HAL 2.0[20] has been introduced. It is based on the previous sensor HAL 1.0, but there are some key differences that prevent sensor HAL 2.0 from being backward compatible. One of the main differences between sensor HAL 1.0 and sensor HAL 2.0 is the different type of IPC (Inter-Process Communication) mechanism that is used in order to exchange data between the HAL layer and the framework layer. Sensor HAL 1.0 uses the mechanism called Binder that requires kernel operations and can also involves the scheduler. On the other hand, sensor HAL 2.0

uses an inter-process communication mechanism called Fast Message Queue (FMQ) that does not involve kernel and causes less overhead. In order to send data from the HAL 2.0 to the framework, a Event FMQ is created. Sensor data can be written to the Event FMQ, but they can be read only at the framework level. This imply that it is not possible to read sensor data directly from the HAL layer. The exchange of sensor data is synchronized with the use of proper functions and flags.
It is also possible to run two different kind of sensor HALs at the same time using a framework called Sensor Multi-HAL. A sensor HAL can be stored inside a device as a shared library that can be dynamically loaded during execution by the sensor Multi-HAL framework.
At the HAL level sensors can generate events in different ways called reporting modes[21]. Each sensor type has only one reporting mode associate with it. Four reporting modes exist.

- In continuous mode events are generated at a constant rate defined by a `sampling_period_ns` parameter passed to a function. For example accelerometers and gyroscopes use this reporting mode.
- In on-change mode events are generated only if the measured value has changed. When a on-changed sensor is activated, an event is immediately generated. Example of sensors using the on-change reporting mode are the step counter, proximity, and heart rate sensor types. In this case the `sampling_period_ns` parameter indicates the minimum amount of time between two consecutive events.
- In one-shot mode the sensor detects an event, deactivate itself and then send a single event through the HAL. No other events are sent until the sensor is reactivated. For example significant motion sensor belongs to this kind of sensors.
- Special mode reporting mode exists too. Step detector and tilt detector sensor use this type of mode. Respectively an event is generated each time a step is taken by the user and each time a tilt movement is detected.

3.1.3 Sensor drivers

Below the HAL layer there are the drivers that interacts with the physical devices. Drivers are proprietary and the source code is not available.
In some cases it is possible to have a sensor hub, used to perform some low-level computations at low power in order to reduce the power consumption and to avoid draining device's battery, allowing the main SoC (system on a chip) to stay in a low power mode. The sensor hub can be a single separate chip or can be placed inside the main Soc. Some sensor hubs contain a microcontroller used to perform some computations.
A sensor hub can implement batching too. It consists of buffering sensor data in the sensor hub or in a hardware FIFO buffer and allows to save power by sending sensor data to upper stack layers only when there are many sensor data in the buffer.

3.2 Rotation vector sensor

In our application it is necessary to measure the attitude of the smartphone using the rotation vector sensor. This measurement is the most important one because it is used for implementing the attitude control algorithm that is the main responsible of the stabiliy of the drone.
It is a virtual sensor whose data are obtained using the raw data of the built-in accelerometer, magnetometer and gyroscope. The rotation vector sensor gives as output a vector composed of 4 elements that are equal to the components of a unit quaternion, that can be converted into the roll, pitch and yaw values using the functions `getRotationMatrixFromVector` and `getOrientation` made available by the Android sensor framework. In this way it is possible to measure the orientation of the device respect to a coordinate reference system. The coordinate reference system used by the rotation vector sensor has the following characteristics:

- Z axis points toward the sky and is perpendicular to the ground plane
- Y axis points toward the geomagnetic North pole and it is tangential to the ground

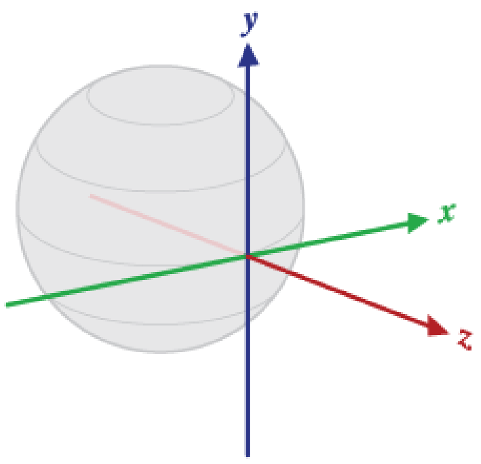


FIGURE 3.2: Rotation vector sensor reference system

- X axis is defined as the vector product Y x Z. It is tangential to the ground

Android sensor service implements internally a sensor fusion algorithm in order to compute the attitude of the smartphone using the data coming from accelerometer, magnetometer and gyroscope. The sensor fusion algorithm consists of a Kalman filter.

3.2.1 Sensor sampling rate

One of the main objective is to discover what is the maximum sensor sampling rate allowed in the Android system and if the sampling rate is constant or not.
The easiest way to find the maximum sampling rate of a specific sensor is to use the `getMinDelay` function of the Sensor class. In this case we will focus on the rotation vector sampling rate. This function returns the minimum time interval in microseconds a sensor can use to acquire a data, whose reciprocal is the maximum sampling rate. The sampling period can be set by the developer changing the corresponding parameter in the function `registerListener`. Anyway the sampling period set is only an hint and Android system or application can alter this period.
The real sampling time can be found using two timestamp values located inside two SensorEvent objects containing two consecutive sensor data acquisition. Setting a desired sampling frequency of 200 Hz we can see that the real sampling frequency is about 203 Hz. There are some sensor acquisitions whose sampling frequency drops to 199 Hz. Moreover the sampling frequency is not perfectly constant and there are oscillations between 202.4 Hz and 203.6 Hz. This behavior coincides with what is written in the Android documentation. Inside the documentation it is written that the system tries to held a sampling frequency a little higher than the one set, but it is not guaranteed that the sampling frequency will be always equal or higher respect to the one requested.

3.2.2 Sensor latency

Using the `registerListener` function it is possible to set also the desired latency, that is the delay between the sensor data acquisition and the time when sensor data are available. By default the latency is set to zero. The timestamp value of a sensor acquisition is synchronized with a clock signal with a nanosecond time base. The same clock can be retrieved using the function `elapsedRealTimeNanos`[22]. In order to measure the real sensor data latency of the rotation vector sensor it is needed to call the `elapsedRealTimeNanos` function inside the callback function where we can access the sensor data. Computing the difference respect to the timestamp value of a sensor data acquisition, it is possible to see from figure 3.4 that a delay of about 6 ms is present, even if the latency set is equal to zero. In addition the delay is not constant. As we can see from figure 3.5, the delays can also have spikes, reaching higher values. The `onSensorChanged` callback where it is possible to access sensor data is called automatically by the system and it is not possible to decide when to call it.
It could be useful to write an application in C++ using the Native Development Kit(NDK) in order to improve the responsiveness of the system and reduce the sensor latency. The NDK includes some native libraries: one of these library called libandroid contains a module[23] that can be used to access

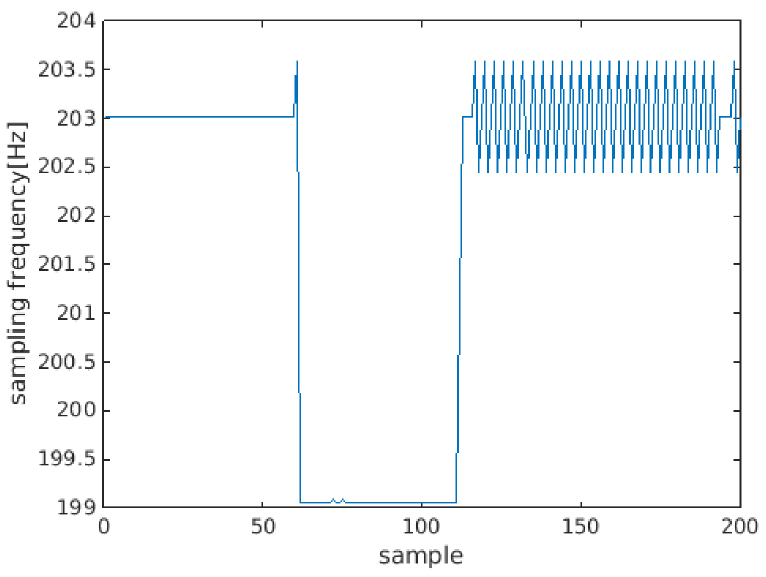


FIGURE 3.3: Sampling frequency of 200 samples

sensor data using C++ code. Anyway no latency improvement has been achieved. This means that the NDK API used to access the sensor with C++ code has the only function of mimic the same API used to access sensor data using Java code.

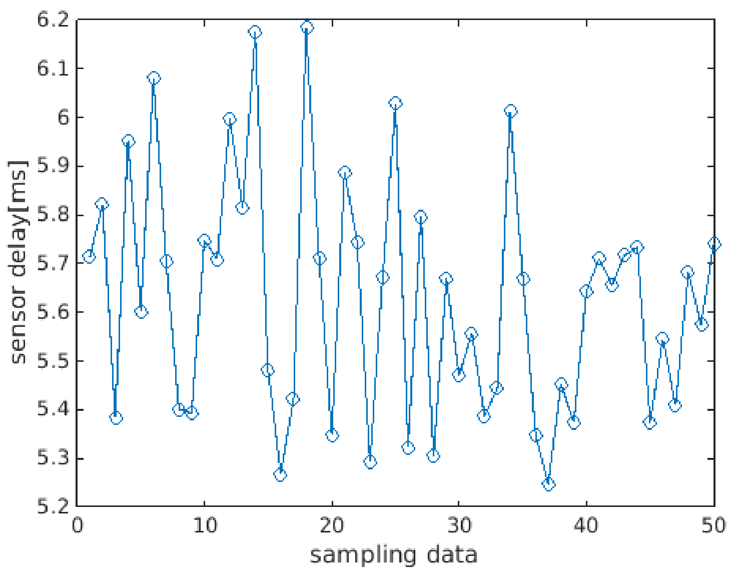


FIGURE 3.4: Sensor delays of 50 samples

3.3 Pressure sensor

The pressure sensor is used for measuring the altitude of the drone in order to stabilize the drone around a target altitude position using a PID controller. This sensor gives as output the atmospheric pressure in millibar. In order to convert the pressure measurement to an altitude measurement it is necessary to call the `getAltitude` function made available by Android sensor framework. The function takes as input parameters the pressure measurement and a parameter representing the pressure at sea level. It is possible to manually set the pressure at sea level or use an average standard value provided by the Android system. The standard value represents an approximation of the real pressure

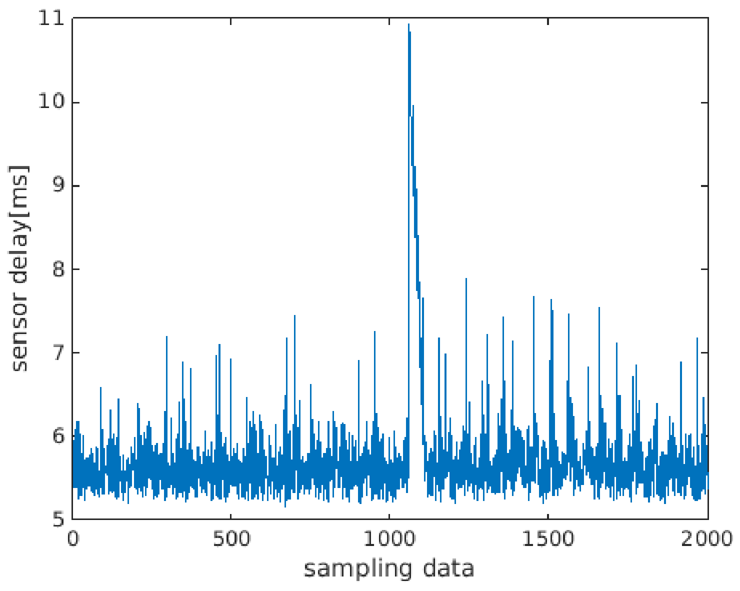


Figure 3.5: Sensor delays of 2000 samples

at sea level, so the absolute altitude measurement will be inaccurate. Considering that it is mainly needed to compute the altitude differences between 2 points, the error caused by the approximation of the pressure at the sea level is negligible.

3.4 GPS

Android provides GPS measurements that can be used for location purposes. In this project the GPS is needed for implementing the position control, realized using 2 PID controller.
In order to access location services provided by Android operating system, it is necessary to use the class LocationManager provided by the Android framework. The LocationManager[24] class provides the `requestLocationUpdates` method that can be used for requesting location data.
It is not possible to set a specific sampling frequency for the GPS measurements. The `requestLocationUpdates` function accepts as input parameters a time value representing the minimum time interval between 2 consecutive location samples and a distance value indicating the minimum travelled distance between 2 consecutive location samples.
When the location data are available, a specific callback is called and it is possible to access a Location object containing the latitude and longitude coordinates of the current position.

3.5 Sensor direct channel

Android operating system makes available a sensor sampling modality which should reach very high sampling rate (about 800 Hz), providing sensor data with low latency. It is called sensor direct channel[25].
This modality involves the use of a memory buffer that can be accessed by different processes and various hardware units such as sensors, GPU (Graphical Processor Unit) or other auxiliary processing unit. In this case sensor data are saved inside the shared memory region and can be accessed and read by the GPU.
Firstly it is necessary to set up the sensor direct channel modality. In particular the memory buffer has to be allocated in the proper way in order to accept sensor data. Once the sensor sampling is configured, it is necessary to read the sensor samples from the shared memory region.
Sensor events will be added into a queue formed by the shared memory region. Each element of the queue has size of 104 bytes and represents a sensor event. The data structure of an element (all fields in little-endian) is:

Offset	Type	Name
0x0000	`int32_t`	Size (always 104)
0x0004	`int32_t`	Sensor report token
0x0008	`int32_t`	Type
0x000C	`uint32_t`	Atomic counter
0x0010	`int64_t`	Timestamp
0x0018	`float`[16]/`int64_t`[8]	Data (data type depends on sensor type)
0x0058	`int32_t`[4]	Reserved (set to zero)

The sequence of new sensor events is determined by the atomic counter, which counts from 1 after creation of direct channel and increments 1 for each new event. Atomic counter will wrap back to 1 after it reaches `UINT32_MAX`, skipping value 0 to avoid confusion with uninitialized memory. The writer in sensor system will wrap around from the start of shared memory region when it reaches the end. If size of memory region is not a multiple of size of element (104 bytes), the residual is not used at the end. Function returns a positive sensor report token on success. This token can be used to differentiate sensor events from multiple sensor of the same type.
In order to access sensor data using the GPU, the OpenGL API(Application Programming Interface) has to be employ.
OpenGL is an API for rendering 2D and 3D graphics. It is typically used for interacting with the GPU in order to achieve hardware-accelerated rendering of images. On smartphones and other embedded devices a subset of Open GL API, called OpenGL ES (Open GL for Embedded System), is normally adopted. In addition it is necessary to employ the EGL API, which is an interface between the OpenGL ES rendering API and the underlying native platform window system. In general the APIs just mentioned are used for performing graphical tasks employing the GPU's computational power. In this case they have to be used in order to read sensor data from the shared memory region using the GPU.

OpenGL ES and EGL APIs contains several funtions that can be divided into 2 main groups: core functions and extension functions. Given a specific OpenGL or EGL version, core functions are normally available to the developer. On the other hand extension functions are not guaranteed to be usable, because the implementation of an extension function could not exist for a specific hardware GPU present on a device.
In order to read sensor data from the shared memory region it is needed to use 2 extension functions. At this stage an important issues arises: on the smartphone used for this project there are problems in opening a required extension function. This could means that the extension function is not implemented on the GPU present inside the smartphone device.

Chapter 4

Smartphone Application

The Android application represents the core of this thesis project. Its main function is to implement the autopilot software, allowing the quadcopter to have a stable flight. Moreover the application has to handle the communication with the Arduino board and with the GCS computer.
The application is composed of 3 principal thread:

- A Sensor thread is used for obtaining the sensor data. The most important data are the measurement of roll, pitch and yaw values that are sampled at a frequency of about 200 Hz. In addition the altitude measurements are sampled at about 25 Hz, while the GPS position points are obtained at about 1 Hz frequency. On this thread the callback function where it is possible to read the data coming from the USB connection is called too. The sensor thread coincides with the Android application main thread, so it is also responsible of most of initialization procedures during the application start-up.
- An Autopilot thread implements the control algorithm. Starting from the sensor measurements and the target commands received, it computes the motors values and send them to the Arduino board through the USB communication. The inner control loop is the most important because it is used to stabilize the attitude. Moreover it is present a control loop for controlling the altitude and one for controlling the position based on GPS measurements. The control algorithm used is a PID, so it is necessary to perform a tuning of the control paramemters.
- A Connection thread is responsible of the communication with the ground control station. In addition some of the computations needed for implementing the position control based on GPS measurements are performed inside this thread.

Two additional threads are called, respectively, when video streaming starts and when a photography is shot.

The Android application is organized in 2 package: androidAutoPilot and androidGroundControl.
The androidAutoPilot package contains the following Java classes that perform tasks related to the control part:

- AdkCommunicator
- Autopilot
- MySensors
- PidAngleRegulator

The androidGroundControl package contains the following Java classes mainly used for the communication with the GCS:

- Connection
- ConnectionVideoToGCS
- Photo
- SendDataToGCS

4.0.1 AdkCommunicator

This class is responsible of the communication with the Arduino board through the USB connection. In this class the callback where it is possible to read data coming from the Arduino board is implemented. The data are encoded into 4 bytes, surrounded by a start byte and an end byte that are used for reading data in the proper order. Depending on the fly mode selected on the radio controller, the start and the end byte change, allowing the autopilot to adopt the right control algorithm.
The callback is also used for indicating the autopilot thread to start. Using this approach, the autopilot thread starts running only if the radio controller is switched on and is sending commands to the drone.
The method `setPowers` is used for sending the motor values computed by the the control algorithm to the Arduino board. Every motor command is encoded in one byte. A start byte is inserted, allowing Arduino board to read data in the correct order.

4.0.2 MySensors

This class implements the callback where sensor data are read, sent to the GCS and saved in the variables that will be read in the autopilot thread for implemeting the proper control strategy. Inside the callback some computation are done in order to get the right measurements.

4.0.3 Autopilot

This class implements the autopilot thread which runs the control algorithm.
At the start of autopilot thread a 4 seconds delay is injected in order to avoid malfunctions for the Arduino board.
The Autopilot thread is composed of a infinite loop which continuosly read sensor data and target commands for computing the motor values to be sent to the Arduino board. While sensor data always come from the sensor thread, the target commands depends on the fly mode the drone is using. The 3 flying modes implemented are:

- Manual
- Altitude regulated
- Automatic

In manual mode and altitude regulated mode the roll, pitch and yaw angles are controlled by using 3 different PID regulators and it is possible to change their target values using the radio controller. In order to control the drone vertical position it is necessary to vary the velocity of all the 4 rotors. In manual mode this component is direcly regulated by the human operator using a stick of the radio controller.
In altitude regulated mode the vertical position is controlled using an additional PID regulator, which uses the altitude measurements obtained by the pressure sensor, allowing the human operator to select the target altitude the drone should reach.
In automatic mode the target altitude, the target roll and the target pitch are chosen according to the commands received by the GCS. The target yaw remains costant.

4.0.4 PidAngleRegulator

PidAngleRegulator class contains the method `getInput` that implements the PID control algorithm. Several objects of this class are instantiated and each one is dedicated to the control of a particular quantity.
Totally 6 PidAngleRegulator objects are present in the Android application. They are:

- rollRegulator for controlling the roll angle
- pitchRegulator for controlling the pitch angle
- yawRegulator for controlling the yaw angle
- altitudeRegulator for controlling the altitude

- latitudeRegulator for controlling the pitch angle according to the GPS position
- longitudeRegulator for controlling the roll angle according to the GPS position

Every PidAngleRegulator object has 3 main parameters that it is necessary to set: the proportional, the integral and the derivative gains. In addition it is possible to set a saturation limit for the integral term and a coefficient for regulating the cut off frequency of a low pass filter used for smoothing the derivative term.

4.0.5 Connection

This class contains the connection thread that is used to handle the communication with the GCS and to perform some computations that are necessary for implementing the position control based on GPS measurements.
The connection thread starts creating a TCP server socket and listening for a connection request on a particular port. After a timeout of 150 ms, the GPS position coordinates are read and the target pitch and roll values are computed by the PID regulators and stored into variables that will be read by the autopilot thread. Using this approach it is possible to stabilize the drone around its current position if the automatic flying mode is running. After these calculations the TCP server socket restarts waiting for a connection request and the loop continues endlessly. If a request arrives from the GCS and a connection is successfully established, the smartphone starts waiting for incoming messages and commands and starts sending the telemetry data and an hearthbeat message at 1 Hz frequency. At the same time the position control continues to run.
In order to receive commands or parameters from the GCS, the appropriate microservices are implemented. Concerning the position control, it is possible to send to the smartphone a path composed of different waypoints. The first received waypoint becomes the current target position that must be reached, while eventual other waypoints are saved into a list. When the the current position measured by the GPS is near the target position, the target position is updated with next waypoint of the list. If the flying mode is changed from automatic to altitude regulated, the drone immediately stops moving because the target position becomes equal to the current position. In addition the list containing waypoints received by the GCS is emptyed.

4.0.6 ConnectionVideoToGCS

This class is used for handling the video streaming. Inside this class a ffmpeg command is called.
Ffmpeg is a free open-source framework which contains a large set of libraries and tools for handling video, audio and other multimedia files and stream. In this case a ffmpeg commands is executed for opening the Android device back camera, encoding the video and sending it to a particular IP address using the RTP protocol. The video format adopted is the H264 or AVC(Advanced Video Encoding), which is the most commonly used video format for the recording, compression and the distribution of video content.
Ffmpeg library provides some presets that can be used for tuning the ratio between encoding speed and compression. The videos must be encoded in the fastest way for minimizing the latency, even if this implies to have less compression and more data to be sent. It is also possible to use the zerolatency tuning option, which changes the encoding setting in order to optimize the encoding process for a low latency streaming.
Moreover it is possible to record the video streaming on the device.

4.0.7 Photo

This class implements the procedures that must be carried out in order to open the camera, shoot a photo and save it.
In particular a CameraManager object is created for detecting, configuring and opening the back camera of the Android smartphone, while a Imagereader object is created for accessing the image shot by the camera. When the camera device is opened, a particular callback is called. Inside the callback a new camera capture session is created, indicating a list of output Surface objects. A Surface is a raw buffer the camera device will render images to. Once the capture session is configured, another callback is called and a capture request is created. The capture request defines a set of parameters the camera device will use for shooting the photography. In this case the capture request requires

the camera to shoot a still image, trying to maximize the quality of the photography. In addition the target Surface the camera will render the image to is indicated. When the image is available a final callback is called and the image is saved.
During the photography shoot process the video streaming is blocked. For this reason, once the photography is saved, the video streaming thread is restarted.

4.0.8 SendDataToGCS

This class contains methods that are used for bulding and sending some important Mavlink messages that are continuosly sent by the smartphone to the GCS. This set of message includes the heartbeat message and the other messages responsible of sending sensor data to the GCS.

Chapter 5

Arduino board

Arduino Mega 2560 board[26] is used in order to handle different operations. In particular it controls the radio communication part and the actuation of motors.
The board is mainly powered by the smartphone thanks to the usb communication.

5.1 Radio communication

The drone is controlled by a human operator on the ground. The human operator sends commands to the drone thanks to a radio transmitter, linked with a radio receiver positioned on the UAV vehicle. The radio receiver sends data to the Arduino board and this communication is achieved using the SBUS protocol.

5.1.1 SBUS protocol

The SBUS protocol is a bus protocol that can be used for receiving signal, but also for sending commands to actuators. Due to the fact that SBUS implements a digital communication, it has been adopted in this project because of its capacity of rejecting electro-magnetic noise caused by the motors actuators.

Initially a radio receiver based on a PWM communication has been used. The PWM radio receiver gives as output a PWM signal with a period of 20 ms. The information brought by this kind of signal is related to its duty cycle, that is the fraction of one period in which the signal stays at high level .
Along the 20 ms period, the PWM signal stays at high level for a time between 1 ms and 2 ms. Then the signal goes low level until the end of the 20 ms period.
In the Arduino code an approach based on interrupt routines has been used in order to read the value of the PWM signal coming from the radio receiver. Anyway the signal received was too noisy.
Initially a combination of a median filter and a low pass filter was implemented on the Android application in order to clean the signal received, but the two filters introduce a significant delay on the signal, so the SBUS protocol based on a digital communication was eventually adopted.

The SBUS protocol uses an inverted serial logic with a baudrate of 100000. Every SBUS packet is 25 bytes long and 16 different channels can be used. The packet is composed of:

- byte [0] represents the SBUS header
- byte [1-22] contains the 16 channels with the data bits
- byte [23] contains 2 digital channels, frame lost bit and failsafe bit
- byte [24] is the SBUS footer or end bit

The frame lost bit is activated when a frame is lost during the transmission, while filesafe bit is activated when different frames are lost in a row and indicates that the receiver is entered in a mode called failsafe mode. Every packet is sent about every 10 ms. Every packet contains 16 channels, each one composed of 11 bit, so the 16 channels are encoded into 22byte.

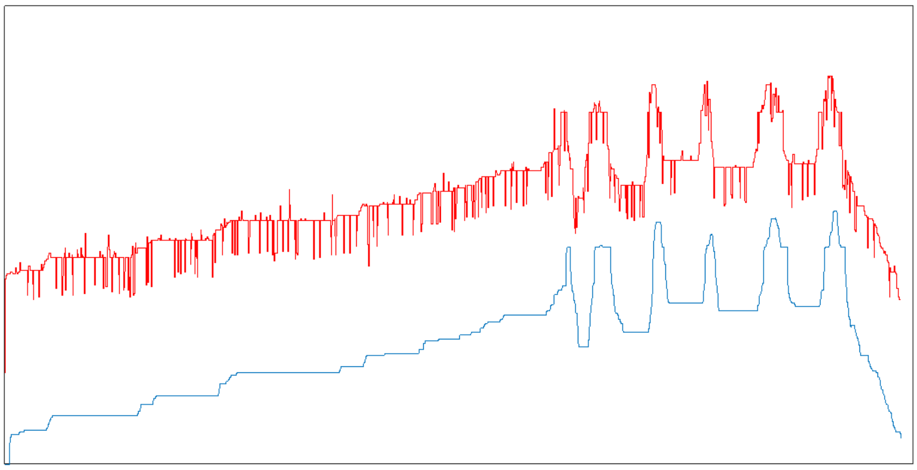


FIGURE 5.1: Noisy and filtered signal using PWM communication

The single signal line coming out from the SBUS receiver is connected to a serial port of Arduino board. Due to the fact that SBUS protocol uses an inverted signal logic that is not supported by Arduino Mega 2560, it is necessary to use a simple inverter circuit based on a NPN transistor.
The data read by the SBUS receiver are sent to the smartphone through the USB port. In particular there are four values representing altitude, roll, pitch and yaw signals. They are remapped into 4 byte values before sending them to the smartphone. In addition a start byte and an end byte are inserted in order to detect possible communication errors, obtaining totally 6 byte.

5.2 Motors actuation

Once the smartphone has run the control algorithm using the sensor signal and the target signal coming from the SBUS radio receiver, the output control data must be sent to the motors. The smartphone sends the output data to Arduino board through the USB port and they are immediately sent to the 4 ESCs (electronic speed controllers). The communication between Arduino and the ESCs is realized with the use of the I2C communication bus.

5.2.1 I2C

The I2C[27] (Inter Integrated Circuit) is a popular serial communication bus invented in 1982 by Philips Semiconductor (now NXP semiconductors). It is normally used for a communication between a master and one or many slave devices, but it can also be implemented using more than one master. One of the main characteristic of the I2C communication bus is that it needs only 2 wire in order to communicate with many different peripherals: the SDA (serial data line) and the SCL (serial clock line). Both lines must be connected to a power supply voltage through a pull-up resistor.
The I2C implements a bidirectional communication, even if a slave cannot transmit data unless it has been authorized by a master. Anyway in our project a mono-directional communication is implemented because it only needed to send some values to 4 slaves, that are the 4 ESC. Every slaves has its own address.

The general procedure for a master to send a data to a slave is:

- the master sends a START condition
- the master sends the slave address
- the master sends the R/W bit
- the slaves sends the ACK signal

- the master sends the register address it wants to write to
- the slaves sends another ACK signal
- the master send the data bits
- the slaves sends another ACK signal and the master sends a STOP signal

A START condition is achieved during a high to low transition on the SDA line while the SCL line is at high level. A STOP condition is achieved during a low to high transition on the SDA line while the SCL line is at high level.
The R/W indicates a write operation if it is set to 0 and a read operation if it is set to 1.
The ACK (acknowledge) bit is sent by the receiver in order to communicate to the transmitter that the transmission has been received and other bits can be sent. Before the receiver is able to send the ACK bit, the transmitter must release the SDA line, leaving the line to a high level. To send an ACK bit the receiver must pull down the SDA line during the low phase of the SCL line in a specific clock period. If the SDA line is pull to the high level during that specific clock period, an NACK bit is sent. A NACK singal indicates that the receiver cannot receive data or the data just received cannot be understood because they contains errors.
It is important to say that a slave usually has different registers in order to memorize different data, but a slave can also have one single register. In this case the data bits are sent immediately after the slave address and the R/W bit.
During the data transmission one bit is transferred during each clock pulse on the SCL line. Data on the SDA line must remain stable during the high phase of the clock signal on the SCL line because a level change when the SCL line is low is interpreted as a control commands (START or STOP).

5.3 Timing

One of the main problem related to the software code running on the Arduino mega 2560 board is that the signals coming from the SBUS radio receiver must be read without interfering with the signals related to the motor actuation. In other words the commands reading must not be a blocking operation and must not cause additional delays in the motor actuation.
This result has been achieved. The commands from the receiver are read at about 100 Hz frequency, while the motors commands are read from the smartphone and written to the ESCs at about 100 Hz, the same frequency at which the data are sent from the smartphone to Arduino. In addition the micro-controller loop runs faster than the operations just mentioned because it has a frequency of about 900 Hz.

5.4 Power supply

The Arduino board is mainly powered by the smartphone thanks to the USB communication. This means that the board switches off if the USB cable accidentally disconnects, causing the motors to stop spinning immediately. In order to avoid this condition another power supply source has been added. A DC-DC converter has been used in order to obtain a stable 5V tension from the 15V LiPo battery. Moreover the code running on the Arduino board has been modified.
If the usb cable disconnects, the smartphone cannot communicate anymore with the Arduino board, so the feedback control loop stops running. When Arduino board stops receiving data from the smartphone, it continues to actuate the motors using the target commands coming from the radio control link, without using any feedback loop. Obviously this working condition cannot lead to a stable flight, but it could be used in emergency situations in order to mitigate the damages when the USB cable accidentally disconnects.

5.5 Code implementation

The code running on the Arduino board is mainly composed of 2 different function: setup() and loop().
When Arduino board is switched on, the setup function is called. It contains the code related to the

initialization preocedures that must carried out before starting the real working program.
Once the setup function has ended, the software enters in the loop function. This function is continuosly called, so the software always pass through the same portion of code contained in the loop function.
This endless loop can only be stopped by an interrupt, which is an event that causes the execution of a particular portion of code called interrupt service routine(ISR). When the ISR is terminated, the software restarts from the point where it was interrupted.

5.5.1 setup

At the start of the setup function the SBUS communication, the I2C communication and the USB serial communication are prepared and started. An external library is used for handling the SBUS communication. The library is added to the project including the "SBUS.h" file and allows us to define the receiver object, which is used for read data encoded using the SBUS protocol. Instead the "Wire.h" file is included in order to manage the I2C communication with the ESCs. Then all the necessary variables are initialized to a known initial value.
In the end some registers are set in order to start a timer, make it count in a particular modality and enable an interrupt received when the timer reaches its maximum values and restarts from zero. In particular the CS51 bit of the TCCRB register is set to 1 in order to have the prescaler at 8. This means that the timer runs at the main microcontroller clock frequency (16 MHz) divided by 8, that is 2 MHz frequency. In addition the timer value (TCNT5) is initialized to zero and the bit TOIE5 of the register TIMSK5 is set, enabling the timer overflow interrupt.
Considering a 16 bit timer, the maximum value that can be reached is 65535. Due to the fact that the timer is incremented at 2 MHz frequency, it takes about 32 ms for reaching its maximum value and launching the appropriate ISR. In this case the ISR is called when the Arduino board does not receive data from the smartphone through the USB communication for more than 32 ms, causing the board to change its behaviour and actuate the motors without any feedback loop. This event should happen only in a emergency situation when, for example, the USB cable is accidentally disconnected during the flight.

5.5.2 loop

In the first part of the loop function the presence of data from the radio controller is checked. If data are available, the 4 target commands are read from their specific communication channel, remapped and stored inside some byte variables: `target_roll`, `target_pitch`, `target_yaw` and `target_altitude`. If the `target_altitude` and `target_pitch` are zero at the same time, a `flagStart` bool variable, which is used as a starting condition, is set to 1. If the radio controller sticks do not pass through the right configuration, the `flagStart` variable remains at zero and the motors are not actuated. Then the presence of smartphone is checked using a `flagNoSmartphone` bool variable, which is set to 1 if the Arduino board does not receive data from smartphone for more than 32 ms. If the smartphone is present and is sending data to the Arduino board, the target commands just received from the radio controller are sent to the smartphone, adding a start and an end byte. The target commands are 4 and every one is encoded as a byte, so totally 6 bytes are sent. Depending on a value sent by the radio controller on a particular channel, the start and the end byte change. In this way 3 different flight mode can be implemented on the smartphone application.
The next portion of code is responsible of reading the motor commands coming from the smartphone. When a data is available, the `flagNoSmartphone` bool variable is set to zero and the 16 bit timer is reset, preventing it to reach its maximum value and launch the timer overflow ISR. Due to the fact that an end byte is inserted at the end of the motor commands frame sent by the smartphone, the data are stored in the appropriate variables only when the end byte is read and if the correct number of byte is arrived. The end byte is inserted in order to read the motor commands in the appropriate order.
If the `flagNoSmartphone` is equal to 1, the motor values are directly computed using the target commands received by the radio controller because it is impossible to receive data from the smarthphone. The motors values, before sending it to the ESCs, can be set to zero if the right stick of the radio controller is below a certain level and if the `flagStart` bool variable is equal to zero.
In the end the motors values are sent to the 4 motors using the I2C communication protocol. It is

necessary to use the `beginTransmission` function of the Wire library for indicating the address of a specific motor, writing the correct value with the `write` function and close the transmission for that specific motor with the `endTransmission` function. This portion of code must be repeated for all the 4 motors.

Chapter 6

Ground control station (GCS)

A Ground Control Station[28] is the control center that provides facilities for establishing a link with a UAV vehicle. The GCS includes both the hardware and the software systems needed for implementing the communication and control tasks.
In general it is possible to divide Ground Control Stations into 2 category: fixed and portable.
Fixed GCS are used especially for large military UAVs. The pilot or the operator usually sits in front of many different screens showing the view from the UAV, a map and aircraft instrumentation. Control is realized through a conventional aircraft-style joystick and throttle. In addition a long range or satellite communication link is implemented.
On the other hand smaller UAVs can be controlled by a traditional "twin-stick" style transmitter. This setup can be extended with a laptop or a tablet computer, creating what is effectvely named as a Ground Control Station.

In our project a portable GCS has been used. Along with the control link realized with a radio trasmission, an additional laptop computer has been introduced in order to establish another communication link, directly connecting the Android smartphone to the laptop. A Wi-Fi based network connection is used for creating a short range communication link, while for a long range communication link an approach based on the 4G mobile network has been adopted.
In order to establish the communication link, the laptop computer runs a software called QGroundControl, which uses a specific communication protocol called Mavlink. On the other side the Android application running on the smartphone has to implement the proper MavLink software interface.
It is appropriate to observe that QGroundControl software can also run on a smartphone, so it is possible to use another smartphone as a Ground Control Station.

6.1 Mavlink

MAVLink[29] or Micro Air Vehicle Link is a very lightweight messaging protocol released in 2009. It is designed for resource-constrained systems and bandwidth-constrained links. It is mainly used for communication between a Ground Control Station and a drone or between different onboard drone components.
MAVLink follows a modern hybrid publish-subscribe and point-to-point design pattern: data streams are sent or published as topics while configuration sub-protocols that involve changes in system configurations are point-to-point with retransmission.
MavLink protocol has been deployed in 2 version: MavLink v1.0 and Mavlink v2.0.
MAVLink 2 is a backward-compatible update to the MAVLink protocol that has been designed to bring more flexibility and security to MAVLink communication. Today MavLink v2.0 is the recommended version and it is used in this project.

6.1.1 Messages

A MavLink message is a stream of bytes encoded into a particular data structure.
In MavLink v2.0[30] the packet structure is the following:

Byte Index	Field name	Description
0	Start-of-frame	Denotes the start of frame transmission.
1	Payload length	Indicates the length of the message (payload).
2	incompatibility flags	Indicate features that a MAVLink library must support in order to be able to handle the packet.
3	compatibility flags	Indicate features will not prevent a MAVLink library from handling the packet.
4	Packet sequence	Each component counts up their send sequence. Allows for detection of packet loss.
5	System ID	Identification of the sending system. Allows to differentiate different systems on the same network.
6	Component ID	Identification of the sending component. Allows to differentiate different components of the same system.
7 to 9	Message ID	Identification of the message - the id defines what the payload “means” and how it should be correctly decoded
10 to (n+10)	Payload	The data into the message, depends on the message id.
(n+11) to (n+12)	CRC	Used to check message integrity and to ensure the sender and receiver both agree in the message that is being transferred.
(n+13) to (n+25)	Signature	Signature to verify that messages originate from a trusted source. (optional)

The protocol defines a large set of messages. Functionalities that can be considered useful for most of autopilot and ground control stations can be implemented using the common message set. If it necessary to extend the common message set it is possible to include some MavLink dialects, obtaining additional message types.

6.1.2 Microservices

The MAVLink microservices define higher-level protocols that MAVLink systems can adopt in order to better inter-operate. The microservices are used to exchange many types of data, including: parameters, missions, images and other files. If the data can be far larger than can be fit into a single message, services will define how the data is split and re-assembled, and how to ensure that any lost data is re-transmitted. Other services provide command acknowledgment and error reporting.
Most services use the client-server pattern, such that the GCS initiates a request and the vehicle responds with data.

6.2 QGroundControl software

QGroundControl[31] is a ground control station software that uses MavLink protocol. It provides a full flight control and mission planning for drones that uses MavLink protocol. It has a GUI (Graphical User Interface) for displaying in a intuitive and clear way some telemetry data such as the current

roll, pitch, yaw and altitude values. The software also shows a map where it is indicated the current position of the drone. In order to display all these values on the GUI, QGroundControl must receive the appropriate set of MavLink messages.
QGroundControl also provides useful GUI elements for sending specific commands or parameters to the drone. In most of the cases a specific microservices must be implemented in order to check if the right commands or parameters have been received by the drone.

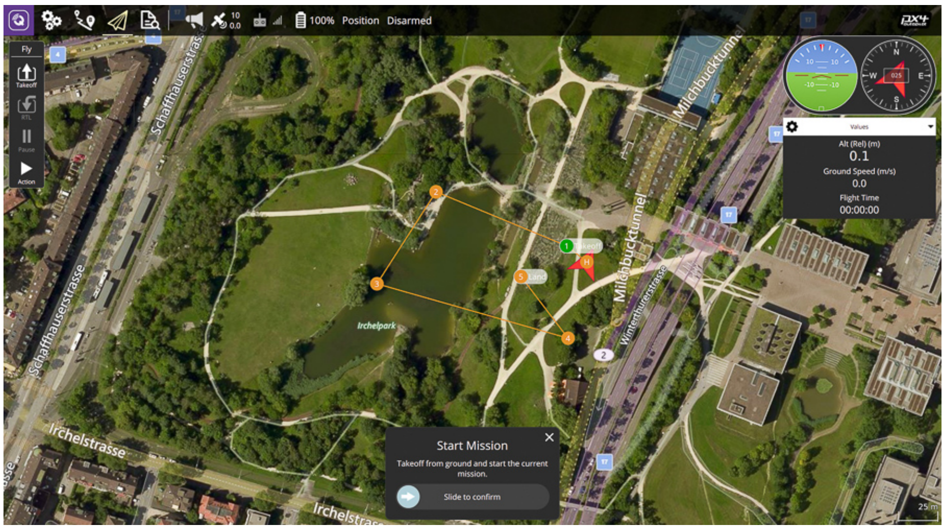


FIGURE 6.1: QGroundControl main window

6.3 Network

The MavLink messaging protocol can works over almost every serial connection and does not depend on the underlying technology.
In this project MavLink connection runs over a TCP socket that can be created over a WiFi network or the 4G mobile network. The Android smartphone is the server, while the GCS computer is the client. Using a TCP socket, this is the only possible configuration because QGroundControl software can behave only as a TCP client and cannot be treated as a TCP server.
The TCP server has to have a reachable IP address in order to be able to receive a connection request by the client. This is a problem if it is needed to work in a 4G mobile network, while it is not a problem using only a WiFi network.

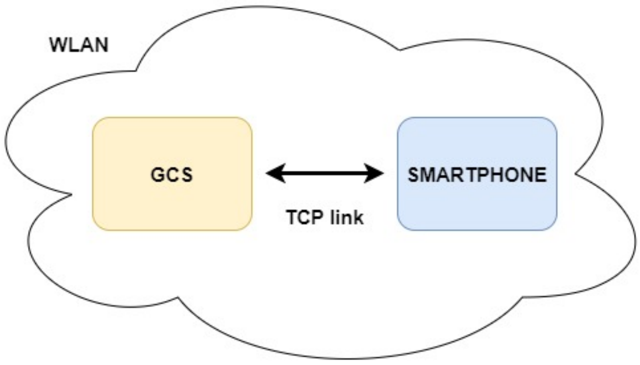


FIGURE 6.2: WiFi network configuration

WiFi is a family of wireless network protocols commonly used for providing internet access and establishing a WLAN (Wireless Local Area Network), allowing nearby devices to exchange data between them. A WLAN is a particular kind of LAN (Local Area Network) that uses a wireless technology.

Inside the same LAN network a private IP address is assigned to each device. The private IP addresses are required for enabling the communication between different devices inside the same network.
Inside the same private WLAN network it is possible to achieve a TCP connection between the ground control station and the smarthphone. The ground control station needs only to know the IP address of the smartphone in order to send the connection request.

In order to create a communication between a device inside a LAN network and the external internet network it is usually present a router implementing a NAT(Network Address Translator).
The router is a networking device that forwards data packet between computer networks. During the routing process, the NAT modifies the destination or the source IP address of data packets, allowing the communication between devices belonging to different networks. In general a device with a private IP address cannot be directly reached by an external connection because it is "hidden" behind the NAT, but it is often possible to make it available using a process called Port Forwarding. It constist of remapping the destination IP address of incoming data packet, allowing to send the connection request to the private IP address placed inside a LAN network.
In a mobile network the IP address of a device is private and it is isolated from the public internet network by a special kind of NAT, called CGNAT. The CGNAT is not available to the end users and it is impossible to directly make use of the Port Forwarding method, but it is possible to obtain the same result using a VPN (Virtual Private Network). The private IP addresses of devices connected to a mobile network belongs to the Shared Address Space.

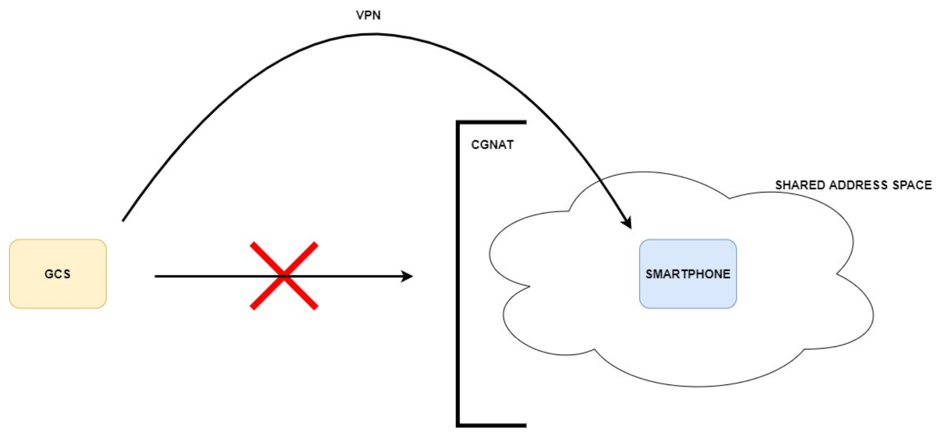


FIGURE 6.3: Mobile network configuration

In this project the video-streaming functionality is not based on the MavLink protocol, but it is implemented using RTP protocol (Real-time Transport Protocol) working over an UDP connection. If the smartphone is connected to internet using the mobile network and the ground control station is inside a WLAN network behind a NAT, it is necessary to use the port forwarding method in order to reach the GCS from the smartphone.
RTP protocol[32] is mainly designed for application involving audio and video streaming over in-

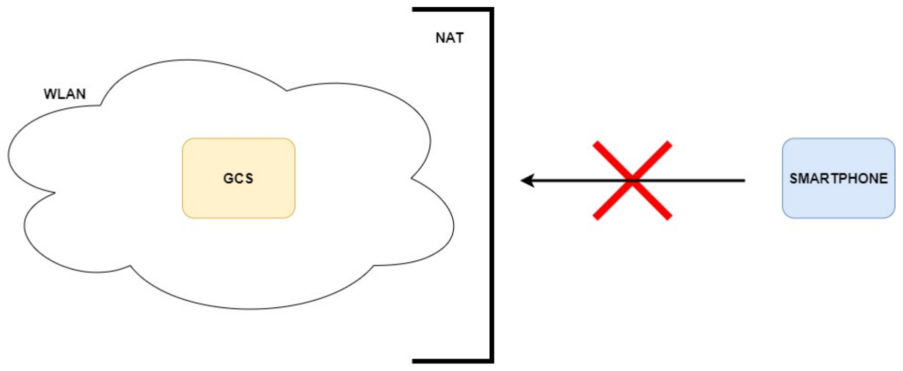


FIGURE 6.4: GCS "hidden" behind a NAT

ternet network. The protocol provides facilities for jitter compensation, detection of packet loss and out-of-order delivery, which are common especially during UDP transmissions. Real-time multimedia streaming applications require timely delivery of information and often can tolerate some packet loss to achieve this goal. The TCP protocol, although standardized for RTP use, is not normally used in RTP applications because TCP favors reliability over timeliness. Instead the majority of the RTP implementations are built on the UDP protocol. RTP also allows data transfer to multiple destinations at the same time and it is designed to carry a multitude of multimedia formats. For each class of application, RTP defines a profile and an associated payload formats. The profile defines the codecs used to encode the payload data and how the payload data has to be mapped into RTP packets.

Chapter 7

Hardware used

A prototype of a quadcopter drone has been built in order to test the autopilot Android application and the code running on the Arduino board. In particular the following hardware has been used.

- Smartphone Google Pixel 4
- Arduino Mega 2560
- 4-in-1 ESC and power board
- Motors MK3638
- FrSky S8R receiver
- Taranis X9D Plus transmitter
- USB-A to USB-B Cable with a USB-B/USB-C adapter
- 4S Battery
- Propellers

7.1 Smartphone

FIGURE 7.1: Google Pixel 4

The smartphone is the core of this project. It run the the Android application that works as an drone autopilot. The smartphone used is a Google Pixel 4. Its main specifications are:

- **Chipset:** Qualcomm SM8150 Snapdragon 855
- **Processor:** Octa-core (1x2.84 GHz Kryo 485 + 3x2.42 GHz Kryo 485 + 4x1.78 GHz Kryo 485)
- **GPU:** Adreno 640
- **RAM:** 6 GB
- **Camera:** 12,2 Mp + 16 Mp + 8 Mp
- **Video:** 4K at 30 fps, 1080p at 120 fps
- **LTE**
- **Wi-Fi:** 802.11 a/b/g/n/ac
- **GPS:** A-GPS/GLONASS/BeiDou/Galileo Dual-frequency GPS
- **Sensors:** Accelerometer, magnetometer, gyroscope, proximity
- **Battery:** Li-Po 3700 mAh
- **USB:** Type-C 3.1

The Google Pixel 4 is an high end smartphone and it has been chosen in order to not have problems related to computational performances. In addition it provides a dual-frequency GPS which can bring a significant improvement in the GPS locationing accuracy.

7.2 Arduino Mega 2560

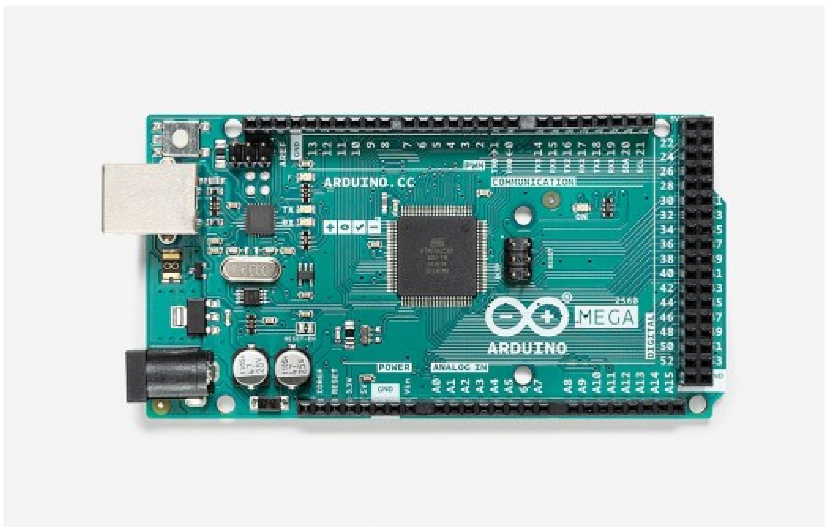
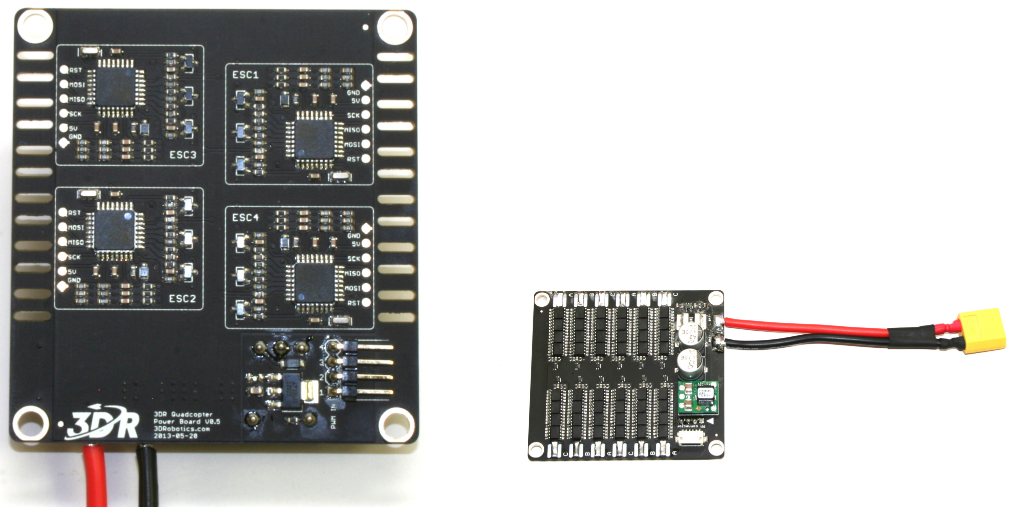


FIGURE 7.2: Arduino Mega 2560

Arduino Mega 2560 is a board based on the ATmega2560 microcontroller. Among many different features this board implements, some of the most interesting characteristics are that it has 6 pins that can be configured to be triggered by external interrupts, 4 serial lines and supports I2C communication protocol.
The USB communication port is controlled by the ATmega16U2 microcontroller. One of the 4 serial line is linked to the ATmega16U2 and it is channeled through the USB port in order to provide a communication between the ATmega2560 microcontroller and an external PC. The USB communication is usually used for programming the board. In our case it is also used for the communication with the Android smartphone.
The DTR line of the ATmega16U2 is connected to the reset pin of the ATmega 2560 via a 100 nF capacitor. When the DTR line is taken down at low level, the reset line drops and resets the main microcontroller. This feature is used in order to reset the microcontroller before an upload of new

firmware by the Arduino IDE running on a PC. After the reset the bootloader starts running and it will intercept the data coming from the USB port. Even if the bootloader is programmed for ignoring malformed data beside the uploaded firmware code, it is necessary to wait a second before sending data to the board after opening the connection. This problem is handled in the Android application code injecting a delay before starting sending data to the Arduino board through the USB line. Without the additional delay the Arduino board immediately stops working.

7.3 ESCs



(A) Front. (B) Back

FIGURE 7.3: ESCs front/back board and power distribution

The electronic speed controllers (ESCs) accomplish the task related to the actuation of the motors. They are responsible of sending the correct signals in order to control the speed of the motors.
The ESCs usually accept as input PWM signal at a frequency of 50 Hz. Along the 20 ms period, the signal stay high for a time between 1 ms and 2 ms. The velocity of the motors depends on the time the PWM signal stays at high level. The more it stays high, the more the velocity is high. Anyway the ESCs used support also I2C communication protocol, that allows to have as input an higher frequency signal. Using the I2C protocol every ESC is indicated by a specific address and accepts as input a digital data that indicates the motor velocity. In our project the I2C protocol is used.

7.4 Motors

The main features of the engines are:

- **Motor type:** Brushless
- **Diameter:** 35 mm
- **Shaft diameter:** 4 mm
- **Weight:** 125 g
- **RPM/V:** 760 KV
- **Battery:** 3-6 S LiPo
- **Power:** 350 W
- **Number magnetic poles:** 14

A set of 11 inches propellers has been used.

7.5 RC

(A) Transmitter

(B) Receiver

Figure 7.4: RC

The radio control part is managed using Taranis X9D Plus radio transmitter and FrSky S8R receiver. The main specifications of the transmitter are:

- **Channels:** 16 (up to 32)
- **Band:** 2.4 GHz
- **Modulation:** ACCST
- **Operating voltage range:**6/15 V
- **Operating current:** 270 mA maximum
- **Operating temperature range:** -10/60 C

The main specifications of the receiver are :

- **Channels:** 16 using SBUS protocol
- **Band:** 2.4 GHz
- **Modulation:** ACCST
- **Operating voltage range** 4/10 V
- **Operating Current:** 120 mA at 5 V
- **Dimension:** 46.47×26.78×14.12 mm
- **Weight:** 14 g

The receiver can generate a RSSI (received signal strengh indicator) PWM output signal. RSSI is a measurement of the power of the received radio signal. It can be used when the receiver is losing transmitter signal.

Chapter 8

Experimental results

8.1 System overview

In this project the main focus are the Android smartphone, the Arduino board and the GCS. The overall system is also composed by the rotors, the drone frame, the ESCs and the battery. All these component just mentioned form a drone which is able to establish a communication link with a GCS. The core functionalities of the overall system resides inside the Android smartphone and the Arduino board. The Arduino board receives target commands from the radio communication link or from the GCS and sends them to the smartphone, which samples sensor data and implements the control algorithm. The motor commands are computed and sent to the Arduino board. In the end the Arduino board sends the motor commands to the ESCs in order to actuate the motor.
Three flight mode are implemented:

- Manual
- Altitude regulated
- Automatic

The flight mode can be selected moving a switch on the radio controller.

8.1.1 Manual mode

In manual flight mode the drone attitude is regulated by the PID control algorithm running on the smartphone, while vertical position is controlled directly varying all rotor 4 velocities. The human operator can make the drone move up increasing the rotors velocities or move down decreasing the rotor velocities, but these operations are not controlled using a feedback loop.

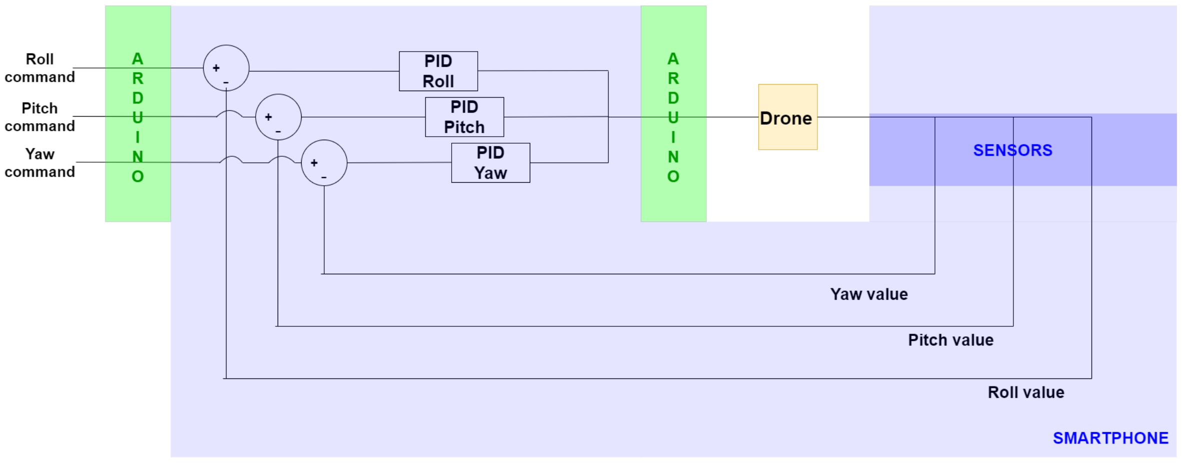


FIGURE 8.1: Manual mode control scheme

8.1.2 Altitude regulated

In this flight mode the altitude is regulated by a PID control algorithm running on the smartphone too. Using the radio controller, the human operator sends to the drone a target altitude that should be reached.

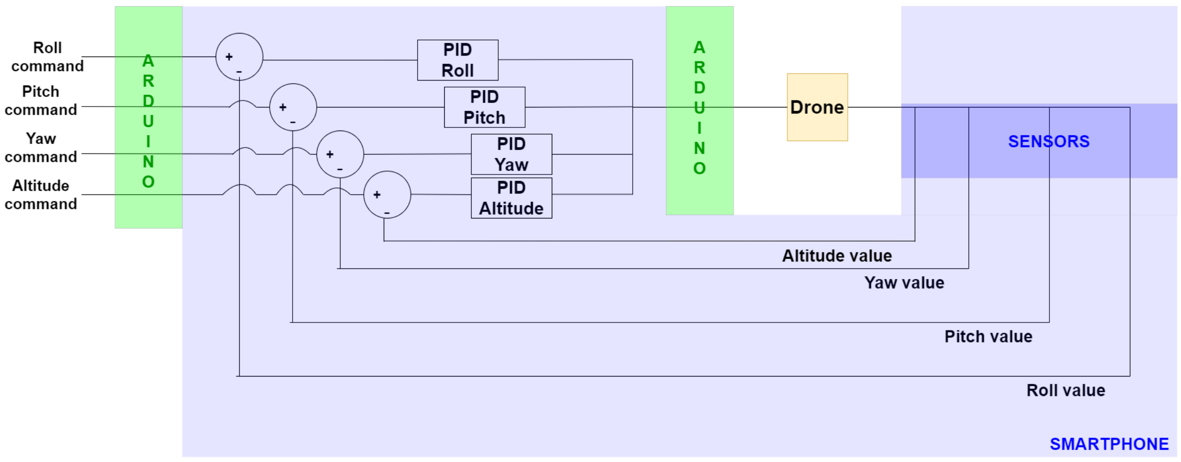


FIGURE 8.2: Altitude regulated mode control scheme

8.1.3 Automatic

In automatic flight mode a position control is implemented. The drone receives some GPS waypoints from the GCS and has to reach them. If no waypoints are sent to the drone, it maintains its current position and altitude. A latitude position error drives a roll movement, while a longitude position error drives a pitch movement. The yaw value is needed in order to know the orientation of the drone.

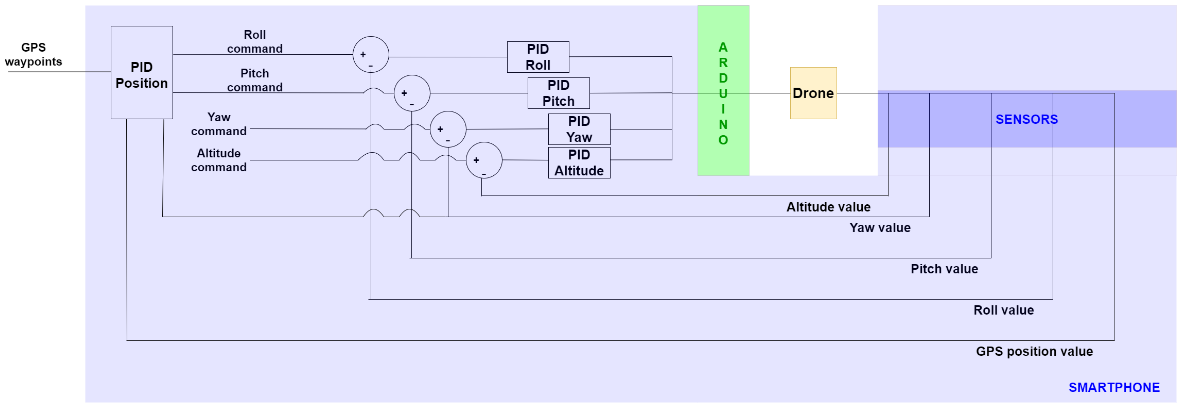


FIGURE 8.3: Automatic mode control scheme

8.2 Manual mode test

The manual flight mode has been tested. In this flight mode the drone is able to fly and can be controlled by human operator using the radio controller.
The Android autopilot application allows to save important data inside log files that can be analyze after the flight.
In the following graphs a sample of 5000 data representing informations about the attitude of the drone is shown. Considering that the sensor sampling rate is about 200 Hz, 5000 data represent about 25 seconds of flight.

8.2.1 Roll

From the following figures it is possible to see that the real roll value does not follow perfectly the target value, but the error between them remains almost always below 2-3 degrees. In particular it is noticeable that in most of the cases the real roll value initially follows pretty well the target commands, but it is always present a slight overshoot when the target commands returns to the zero value. In some cases the error increases, but it does not exceed 6 degrees.

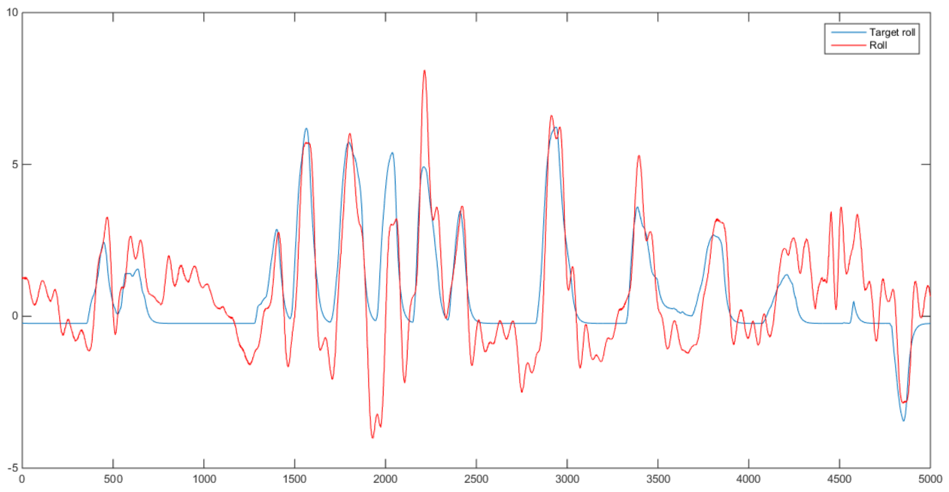


FIGURE 8.4: Roll response

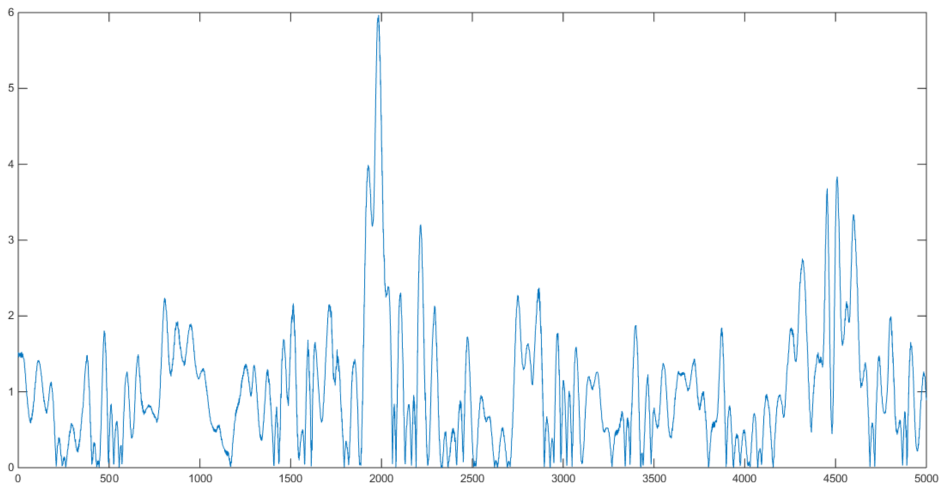


FIGURE 8.5: Roll error

8.2.2 Pitch

The pitch has a behaviour very similar to the roll's one. Also in this case it is always present a little overshoot and the real pitch values does not perfectly follow the target commands. The error between the target pitch and the real pitch values presents an average value greater respect to the roll error's one, but the error never exceed 5 degrees.

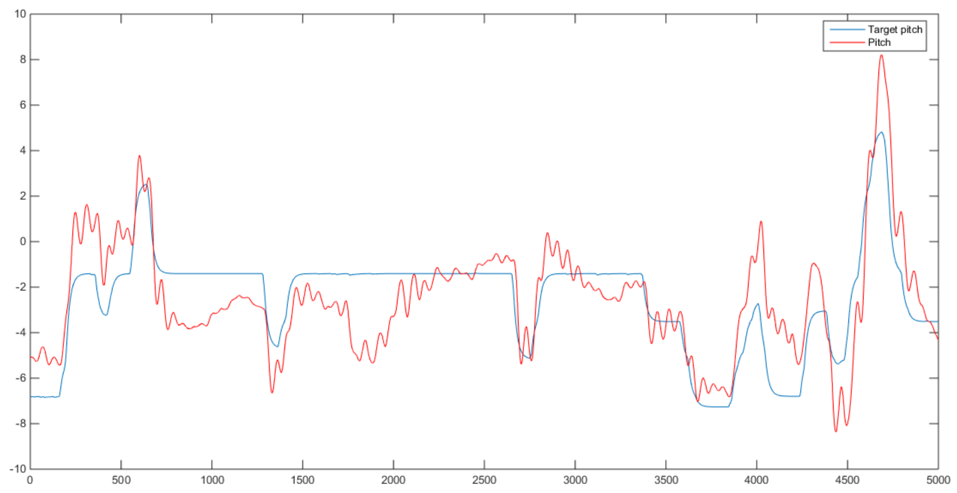


FIGURE 8.6: Pitch response

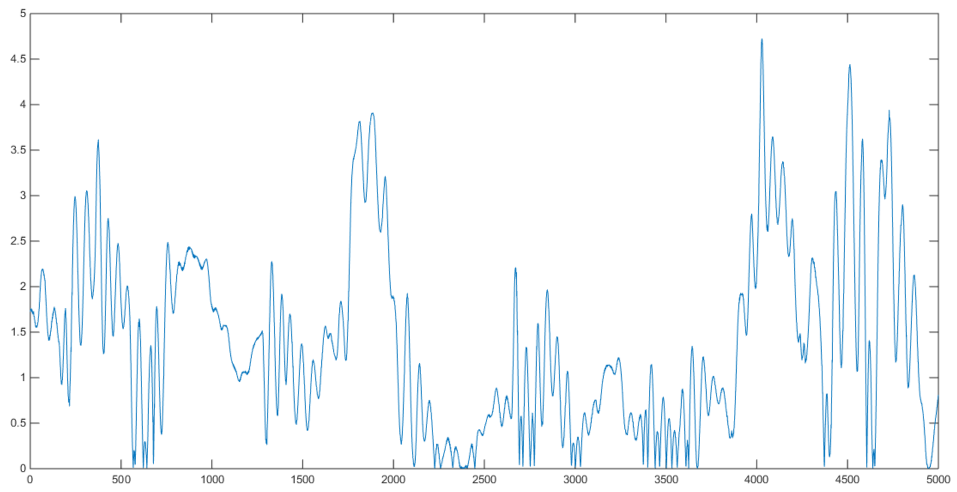


FIGURE 8.7: Pitch error

8.2.3 Yaw

The yaw response presents poorer performances respect to the roll and pitch responses. The error reaches higher values and the response is slower. In addition the error exceeds 10 degree even if the target command remains almost constant.
It is necessary to say that in order to obtain the flight stability of a quadcopter the roll and pitch movements are the most important, while the yaw movement does not need to have very high performances. Anyway the yaw control could be improved setting the PID control parameters in a better way.

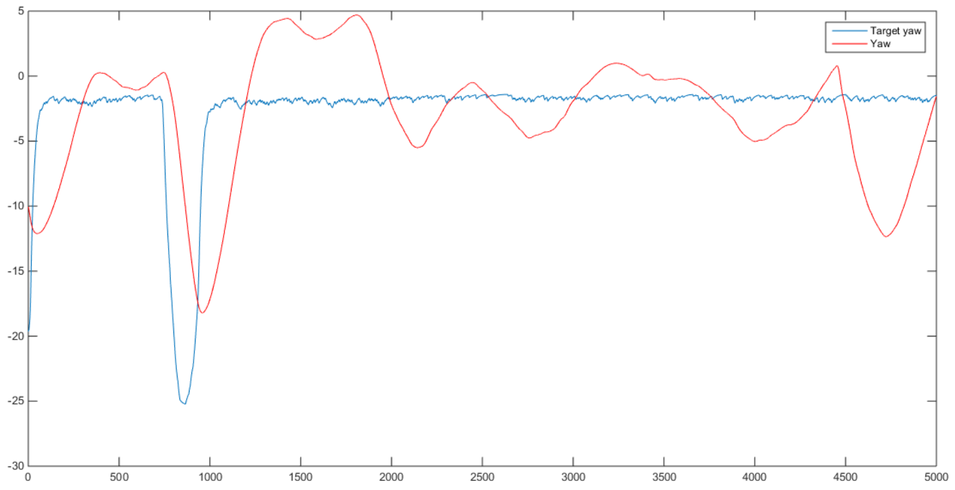


FIGURE 8.8: Yaw response

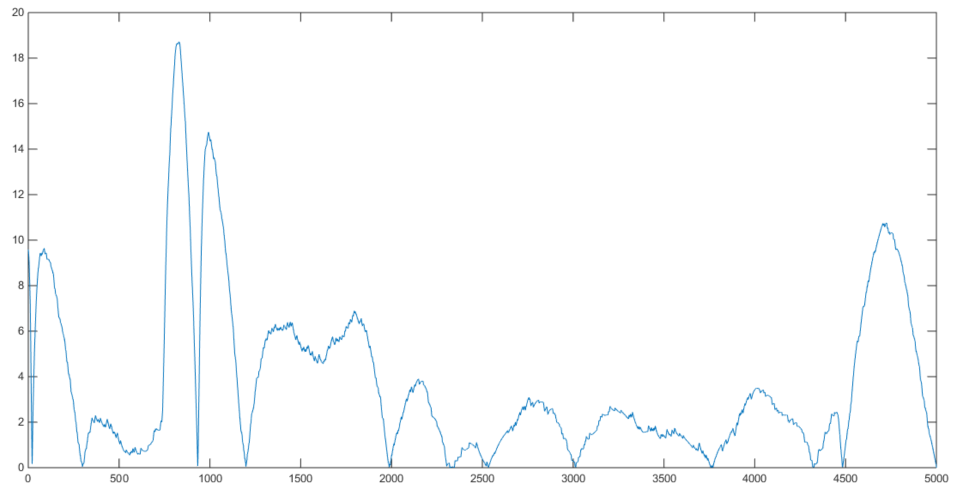


FIGURE 8.9: Yaw error

8.3 Mavlink connection and video streaming test

The activation of the Mavlink connection and the video streaming funtionality has an impact on the performances of the Android autopilot task. Without connecting the Android smarthphone to the Arduino board, some tests have been performed. During the test a collection of 1000 samples about the rotation vector sensor sampling period and rotation vector sensor latency is collected. For each quantity an average value is computed.
It is necessary to say that it is always possible that some sensor sample data are lost. In other words it means that the sensor sampling period sometimes could pass from a normal value of about 5 ms to 10 ms or 15 ms. Activating additional funtionalities, the number of lost samples increases and causes a bigger impact on the computed average sampling period.
Running only the control algorithm, the average sampling period is 4.93 ms. It can be considered as the nominal sampling period that is not influenced by the loss of some samples. It is possible to see that the video streaming functionality has an huge impact on the performances, while the Mavlink connection causes fewer delays. In particular the delays increase dramatically setting an higher video resolution.

Active functionalities	Average rotation sensor sampling period [ms]	Average rotation sensor latency [ms]
None(only autopilot)	4.93	5.73
Mavlink connection to GCS	4.94	5.79
Video streaming (320x240 resolution)	4.99	5.94
Mavlink connection to GCS + video streaming (320x240 resolution)	5.01	5.95
Mavlink connection to GCS + HD video streaming (1280x720 resolution)	5.15	6.10

Chapter 9

Conclusions

In this master thesis project a prototype of a quadcopter drone has been realized. The core of the drone consists of an Android smartphone running the control algorithm, while the Arduino board can be considered as an interface between the motors, the human operator and the smartphone.
The main objective of this thesis is accomplished: the drone flies and it is stable.
This result has been achieved even if Android is not a real time operating system and injects some delays into the system.
The main sources of delays are the sensors latency and the garbage collection. The sensors latency is always present if it is needed to read sensor data, while the garbage collection causes important delays only if additional funtionalities that require a lot of memory are activated. It could be possible to achieve real time performances on Android, but it would imply modifing some core parts of the operating system[33]. In this case an Android application has been realized and it can be installed on many different Android smartphones equipped with the Android OS.

9.0.1 Future improvements

The flight stability has been obtained implementing an attitude controller. In addition an altitude and a position controller has been implemented, but they have not been tested. In particular it is necessary to tune the altitude controller and the position controller. It is not only needed to choose the right value for the proportional, integral and derivative gains but it could also be necessary to change the integrator maximum value and tune the filter of the derivative term. Regarding the altitude control, it could also be necessary to tune the filter of the pressure sensor. On the other side it could be appropriate to filter the roll and pitch commands values computed by the position controller.
One major improvement could be achieved implementing the sensor direct channel sampling modality which should provides sensor data at a frequency of about 800 Hz, 4 times higher the sampling frequency it is currently possible to use.
Eventually the Android application can be optimized, especially the parts related to the communication with the GCS that do not have a primary importance and do not need to run at high frequencies.
This system architecure based on an Android smartphone and an Arduino board could be extended to different kind of UAV vehicles and to UGV vehicles too.

Bibliography

[1] *Unmanned aerial vehicle.* URL: https://en.wikipedia.org/wiki/Unmanned_aerial_vehicle.

[2] *Quadcopter.* URL: https://en.wikipedia.org/wiki/Quadcopter.

[3] Eros Giorgi. *"Implementation of an autopilot for UAV/UGV system based on Android smartphone.* 2019. URL: https://webthesis.biblio.polito.it/10946/1/tesi.pdf.

[4] *Android.* URL: https://en.wikipedia.org/wiki/Android_(operating_system).

[5] *Android Architecture.* URL: https://source.android.com/devices/architecture?hl=en.

[6] *Application Fundamentals.* URL: https://developer.android.com/guide/components/fundamentals.

[7] *Introduction to Activities.* URL: https://developer.android.com/guide/components/activities/intro-activities.

[8] *Understand the Activity Lifecycle.* URL: https://developer.android.com/guide/components/activities/activity-lifecycle.

[9] *Intents and intent filters.* URL: https://developer.android.com/guide/components/intents-filters.

[10] *Binder.* URL: https://developer.android.com/reference/android/os/Binder.

[11] Thorsten Schreiber. *Android Binder.* 2011. URL: https://www.nds.ruhr-uni-bochum.de/media/attachments/files/2011/10/main.pdf.

[12] *Ndk.* URL: https://developer.android.com/ndk/guides.

[13] *Overview of memory managment.* URL: https://developer.android.com/topic/performance/memory-overview.

[14] *Sensor overview.* URL: https://developer.android.com/guide/topics/sensors/sensors_overview.

[15] *Sensor stack.* URL: https://source.android.com/devices/sensors/sensor-stack?hl=en.

[16] *SensorManager.* URL: https://developer.android.com/reference/android/hardware/SensorManager.

[17] *SensorListener.* URL: https://developer.android.com/reference/android/hardware/SensorListener.

[18] *Sensor.* URL: https://developer.android.com/reference/android/hardware/Sensor.

[19] *SensorEvent.* URL: https://developer.android.com/reference/android/hardware/SensorEvent.

[20] *Sensor HAL 2.0.* URL: https://source.android.com/devices/sensors/sensors-hal2?hl=en.

[21] *Reporting modes.* URL: https://source.android.com/devices/sensors/report-modes?hl=en.

[22] *System clock.* URL: https://developer.android.com/reference/android/os/SystemClock.

[23] *Ndk sensor module.* URL: https://developer.android.com/ndk/reference/group/sensor.

[24] *Location manager.* URL: https://developer.android.com/reference/android/location/LocationManager.

[25] *Sensor direct channel.* URL: https://developer.android.com/reference/android/hardware/SensorDirectChannel.

[26] *Arduino Mega 2560 website.* URL: https://store.arduino.cc/arduino-mega-2560-rev3.

[27] Jared Becker Jonathan Valdez. *Understanding I2C.* URL: https://www.ti.com/lit/an/slva704/slva704.pdf?ts=1626092822053&ref_url=https%253A%252F%252Fwww.google.com%252F.

[28] *Ground control station.* URL: https://en.wikipedia.org/wiki/UAV_ground_control_station.

[29] *MAVLink developer guide.* URL: https://mavlink.io/en/.

[30] *MAVLink.* URL: https://en.wikipedia.org/wiki/MAVLink.

[31] *QGroundControl website.* URL: http://qgroundcontrol.com/.

[32] *RTP.* URL: https://en.wikipedia.org/wiki/Real-time_Transport_Protocol.

[33] Cláudio Maia Luis Miguel Nogueira Luis Miguel Pinho. *Evaluating Android OS for Embedded Real-Time Systems.* 2010. URL: http://www.cister.isep.ipp.pt/docs/evaluating_android_os_for_embedded_real_time_systems/569/view.pdf.

Appendix A

JavaCode

A.1 androidAutopilot Package

A.1.1 AdkCommunicator

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()) {
                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");
            } else {
                Log.d("SERIAL", "PORT NOT OPEN");
            }
        } else {
            Log.d("SERIAL", "PORT IS NULL");
        }
    }
```

onReceiveData

```
UsbSerialInterface.UsbReadCallback usbReadCallback=
                new UsbSerialInterface.UsbReadCallback() {
        @Override
        public synchronized void onReceivedData(byte[] arg0) {
            try {
                 if (arg0[0]==0 && arg0[5]==0)
                 {
                     receivedData.roll =  arg0[1] & 0xFF;
                     receivedData.pitch = arg0[2] & 0xFF;
                     receivedData.yaw = arg0[3] & 0xFF;
                     receivedData.altitude = arg0[4] & 0xFF;
                     autoPilot.mode=0;
                 }
                 else if (arg0[0]==1 && arg0[5]==1)
                 {
                     receivedData.roll =  arg0[1] & 0xFF;
                     receivedData.pitch = arg0[2] & 0xFF;
                     receivedData.yaw = arg0[3] & 0xFF;
                     receivedData.altitude = arg0[4] & 0xFF;
                     autoPilot.mode=1;
                 }
                 else if (arg0[0]==2 && arg0[5]==2)
                 {
                     receivedData.roll =  arg0[1] & 0xFF;
                     receivedData.pitch = arg0[2] & 0xFF;
```

```
                    receivedData.yaw = arg0[3] & 0xFF;
                    receivedData.altitude = arg0[4] & 0xFF;
                    autoPilot.mode=2;
                }
              if(!autoPilot.getPidCommandThread().isAlive() &&
                            serialPort != null) {
                   autoPilot.startPIDThread();
               }
           } catch (Exception e) { }
       }
   };
```

setPowers

```
public void setPowers(MotorsPowers powers) {
        txBuffer[0] = (byte) powers.sw;
        txBuffer[1] = (byte) powers.se;
        txBuffer[2] = (byte) powers.ne;
        txBuffer[3] = (byte) powers.nw;
        txBuffer[4] = (byte) 0;
        if (serialPort!=null) {
           serialPort.write(txBuffer);
        }
    }
```

A.1.2 Autopilot

Autopilot thread

```
private Runnable PIDCommand = new Runnable() {
        @Override
        public void run() {
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            double tempPowerNW = 0, tempPowerNE = 0, tempPowerSE = 0, tempPowerSW = 0;
            float currentRoll=0, targetAngleRoll=0, currentPitch=0, targetAnglePitch=0;
            float currentYaw=0, targetAngleYaw=0,targetAltitude=0,currentAltitude=0;
            float zero_altitude=0;
            float meanTargetAngleRoll=0;
            float meanTargetAnglePitch=0;
            float dt;
            float dt_pressure;
            mode=0;
            sensorsData = mySensors.getSensorsData();
            targetAngleYaw=sensorsData.yaw;
            boolean flagModeTwo=false;
            int contFlagModeTwo=0;
            zero_altitude=sensorsData.zero_altitude;
            while (true) {
                sensorsData = mySensors.getSensorsData();
                receivedData = adkCommunicator.getReceivedData();
                currentRoll = sensorsData.roll;
                currentPitch = sensorsData.pitch;
                currentYaw = sensorsData.yaw;
                currentAltitude=sensorsData.altitude-zero_altitude;
                dt = ((float) (sensorsData.time - sensorsData.previousTime));
                if (acquisition_time != sensorsData.time && dt>2500000) {
                    if(mode==2)
                    {
                        sendHoldingToConnection(true);
                        targetAngleRoll=getTargetRollFromConnection();
                        targetAnglePitch= (float) (getTargetPitchFromConnection()-1.4);
                        targetAltitude=getTargetAltitudeFromConnection()-zero_altitude;
                    }
                    else
                    {
                        sendHoldingToConnection(false);
                        targetAngleRoll=(receivedData.roll-128)/128*30;
                        targetAnglePitch=(receivedData.pitch-128)/128*30;
                        if (mode==1)
                        {
                            if (receivedData.altitude<5)
                            {
                                altitudeRegulator.integrator=0;
                            }
                            else if (abs(receivedData.altitude-127)>5)
                            {
                                targetAltitude=targetAltitude
                                        +(receivedData.altitude - 127)/20000;
                                if (targetAltitude>7)
                                {
                                    targetAltitude=7;
                                }
                            }
                        }
                        else
                        {
                            targetAltitude=(receivedData.altitude - 20);
                        }
                        if (abs(receivedData.yaw-126.5)>5)
                        {
                            targetAngleYaw=(float)
                                    (targetAngleYaw-((receivedData.yaw-126.5)/640));
                            while(targetAngleYaw < -180.0f)
                            {
                                targetAngleYaw += 360.0f;
                            }
                            while(targetAngleYaw > 180.0f)
                            {
                                targetAngleYaw -= 360.0f;
                            }
                        }
                    }
        meanTargetAngleRoll= (float) (meanTargetAngleRoll*0.95+targetAngleRoll*0.05);
        meanTargetAnglePitch= (float) (meanTargetAnglePitch*0.95+targetAnglePitch*0.05);
        targetAngleRoll=meanTargetAngleRoll;
        targetAnglePitch=meanTargetAnglePitch;
        rollForce = rollRegulator.getInput(targetAngleRoll, currentRoll, dt,true);
        pitchForce = pitchRegulator.getInput(-targetAnglePitch, currentPitch, dt,true);
        yawForce = yawRegulator.getInput(targetAngleYaw,currentYaw,dt,true);
        dt_pressure= ((float)
                (sensorsData.pressure_time - sensorsData.previous_pressure_time));
        if (dt_pressure>20000000 && mode!=0)
        {
                altitudeRegulator.setKp(getAltitudeRegulatorKp());
                altitudeRegulator.setKd(getAltitudeRegulatorKd());
                altitudeRegulator.setKi(getAltitudeRegulatorKi());
                altitudeForce=altitudeRegulator.getInput
                        (targetAltitude, currentAltitude , dt_pressure,false);
        }
        if (mode==0)
        {
            altitudeForce=targetAltitude;
        }
        tempPowerNW = altitudeForce;
        tempPowerNE = altitudeForce;
```

```
        tempPowerSE = altitudeForce;
        tempPowerSW = altitudeForce;
        tempPowerNW += rollForce;
        tempPowerNE -= rollForce;
        tempPowerSE -= rollForce;
        tempPowerSW += rollForce;
        tempPowerNW += pitchForce;
        tempPowerNE += pitchForce;
        tempPowerSE -= pitchForce;
        tempPowerSW -= pitchForce;
        tempPowerNW += yawForce;
        tempPowerNE -= yawForce;
        tempPowerSE += yawForce;
        tempPowerSW -= yawForce;
        motorsPowers.nw = motorSaturation(tempPowerNW);
        motorsPowers.ne = motorSaturation(tempPowerNE);
        motorsPowers.se = motorSaturation(tempPowerSE);
        motorsPowers.sw = motorSaturation(tempPowerSW);
        long usb_t = SystemClock.elapsedRealtimeNanos();
        adkCommunicator.setPowers(motorsPowers);
        acquisition_time = sensorsData.time;
        delay_sensor=sensorsData.delay;
       if (outStream!=null) {
       try {
       outStream.write((
       acquisition_time + ";" + currentRoll + ";" + currentPitch + ";" +
        motorsPowers.nw + ";" + motorsPowers.ne + ";" + motorsPowers.sw +
         ";" + motorsPowers.se + ";" + altitudeForce+";"+
        receivedData.altitude + ";"+usb_t +
       ";"+rollRegulator.integrator+";"+rollRegulator.derivative+
        ";"+rollRegulator.kp+";"+
       rollRegulator.ki+";"+rollRegulator.kd+";"+
        targetAngleRoll+";"+receivedData.roll+";"
       +pitchRegulator.integrator+";"+pitchRegulator.derivative+
        ";"+pitchRegulator.kp+";"+
       pitchRegulator.ki+";"+pitchRegulator.kd+";"+
        targetAnglePitch+";"+receivedData.pitch+";"+
       yawRegulator.integrator+";"+yawRegulator.derivative+
        ";"+yawRegulator.kp+";"+yawRegulator.ki+
       ";"+yawRegulator.kd+";"+targetAngleYaw+";"+receivedData.yaw
        +";"+currentYaw+";"+
        currentAltitude+";"+delay_sensor+";"+
        targetAltitude+";"+
       ";"+rollForce+";"+pitchForce+";"+yawForce+";"+
        (SystemClock.elapsedRealtimeNanos()/1000000)
        +";\n").getBytes());
       } catch (IOException e) {
               e.printStackTrace();
                        }
        }
       }
       }
     }
   }
;
```

A.1.3 MySensors

onSensorChanged

```
@Override
public void onSensorChanged(SensorEvent event) {
if(event.sensor.getType() == Sensor.TYPE_ROTATION_VECTOR){
        System.arraycopy(event.values, 0, rotationVec, 0, 3);
        SensorManager.getRotationMatrixFromVector(rotationMatrix, rotationVec);
        SensorManager.getOrientation(rotationMatrix, yawPitchRollVec);
        sensorsData.yaw = getMainAngle(-(yawPitchRollVec[0]) * RAD_TO_DEG);
        sensorsData.pitch = getMainAngle(-(yawPitchRollVec[1]) * RAD_TO_DEG);
        sensorsData.roll = getMainAngle((yawPitchRollVec[2]) * RAD_TO_DEG);
        sensorsData.previousTime = sensorsData.time;
        sensorsData.time = event.timestamp;
        delay=SystemClock.elapsedRealtimeNanos()-event.timestamp;
        sensorsData.delay=delay;
        this.connectionMavlink.setCurrentYaw(sensorsData.yaw);
}
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));
        if (pressure_count>150 && pressure_count<=200)
        {
                sensorsData.altitudeTemp=sensorsData.altitudeTemp+rawAltitudeUnsmoothed;
                if (pressure_count==200)
                {
                sensorsData.zero_altitude=sensorsData.altitudeTemp/(pressure_count-150);
                this.connectionMavlink.setAltitudeZero(sensorsData.zero_altitude);
                }
        }
        pressure_count=pressure_count+1;
        sensorsData.altitude=absoluteElevation;
        sensorsData.previous_pressure_time= sensorsData.pressure_time;
        sensorsData.pressure_time=event.timestamp;
        sendAltitudeToConnectionMavlink();
        this.connectionMavlink.sendDataToGCS.sendSensorsData(sensorsData);
        }
}
```

locationListener

```
private LocationListener locationListener = new LocationListener() {
        @Override
        public void onLocationChanged(Location location) {
            Log.d("Location Change", "Send new location from GPS module");

            double latitude=location.getLatitude();
            double longitude=location.getLongitude();
            gpsdata=gpsdata+1;
            String gpsdatastring=Long.toString(gpsdata);
            String gpslatitudestring=Double.toString(latitude);
            String gpslongitudestring=Double.toString(longitude);
            String t= Long.toString(SystemClock.elapsedRealtimeNanos()/1000000);
            connectionMavLink.sendGpsData(location);
            connectionMavLink.setActualLocation(location);
            if (autoPilot.gpsStream!=null)
            {
                try {
                    autoPilot.gpsStream.write((gpsdatastring+","+t+","+
                        gpslatitudestring+","+gpslongitudestring+"\n").getBytes());
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
        @Override
        public void onStatusChanged(String provider, int status, Bundle extras) {
        }
        @Override
        public void onProviderEnabled(String provider) {
        }
        @Override
        public void onProviderDisabled(String provider) {
        }
    };
```

A.1.4 PidAngleRegulator

getInput

```
public float getInput(float targetAngle, float currentAngle, float dt, boolean angle)
    {
        float rawDifference = targetAngle - currentAngle;
        if (angle)
        {
            difference = getMainAngle(rawDifference);
        }
        else {
            difference=rawDifference;
        }
        boolean differenceJump = (difference != rawDifference);
        float input = 0.0f;
        input += difference * kp;
        integrator += difference * dt * ki / 1000000000;
        if(integrator > MAX_INTEGRATOR)
           integrator = MAX_INTEGRATOR;
        else if(integrator < - MAX_INTEGRATOR)
            integrator = -MAX_INTEGRATOR;
        input += integrator;
        if(!differenceJump)
        {
            differencesMean = differencesMean * smoothingStrength
                                + difference * (1 - smoothingStrength);
            derivative = (differencesMean - previousDifference) / dt * 1000000000;
            previousDifference = differencesMean;
            input += derivative * kd;
        }
        else
        {
            differencesMean = 0.0f;
        }
        return input;
    }
```

getMainAngle

```
public static float getMainAngle(float angle) {
        while(angle < -180.0f)
        {
            angle += 360.0f;
        }
        while(angle > 180.0f)
        {
            angle -= 360.0f;
        }
        return angle;
    }
```

A.2 androidGroundControl Package

A.2.1 Connection

Connection thread

```
@RequiresApi(api = Build.VERSION_CODES.N)
    @Override
    public void run() {
        while(true) {
            try (ServerSocket serverSocket = new ServerSocket(5760)) {
        serverSocket.setSoTimeout(150);
        int systemId = 2;
        int componentId = 1;
        sendDataToGCS = new SendDataToGCS(systemId, componentId,singleConnection);
        singleSocket = serverSocket.accept();
        singleConnection = MavlinkConnection.create(
                singleSocket.getInputStream(),
                singleSocket.getOutputStream());
        sendDataToGCS.setConnection(singleConnection);
        sendDataToGCS.sendBatteryStatus(batteryLevel);
        sendDataToGCS.sendGpsData(actualLocation);
                new Thread(new Runnable() {
                    @Override
                    public void run() {
                        Log.d("HeartBeat", "send");
                        sendDataToGCS.sendHeartbeat();
                    }
                }).start();
                int missionItemNumber=0;
                int missionItemCounter=0;
                while (!singleSocket.isClosed()) {
                Log.d("Connection","Waiting for messages");
                MavlinkMessage m;
                int count = 0;
```

```
while ((m = singleConnection.next()) != null) {
if (m.getPayload() instanceof ParamRequestList)
{
        Log.d("Connection","Params request received");
        ParamValue paramValue=ParamValue.builder()
        .paramId("Altitude Kp")
        .paramValue(altitudeKp)
        .paramType(MavParamType.MAV_PARAM_TYPE_REAL32)
        .paramIndex(0)
        .paramCount(9)
        .build();
        singleConnection.send2(systemId, componentId, paramValue);
        ParamValue paramValue1=ParamValue.builder()
        .paramId("Altitude Ki")
        .paramValue(altitudeKi)
        .paramType(MavParamType.MAV_PARAM_TYPE_REAL32)
        .paramIndex(1)
        .paramCount(9)
        .build();
        singleConnection.send2(systemId, componentId, paramValue1);
        ParamValue paramValue2=ParamValue.builder()
        .paramId("Altitude Kd")
        .paramValue(altitudeKd)
        .paramType(MavParamType.MAV_PARAM_TYPE_REAL32)
        .paramIndex(2)
        .paramCount(9)
        .build();
        singleConnection.send2(systemId, componentId, paramValue2);
        ParamValue paramValue3=ParamValue.builder()
        .paramId("Latitude Kp")
        .paramValue(latitudeRegulator.getKp()*1000)
        .paramType(MavParamType.MAV_PARAM_TYPE_REAL32)
        .paramIndex(3)
        .paramCount(9)
        .build();
        singleConnection.send2(systemId, componentId, paramValue3);
        ParamValue paramValue4=ParamValue.builder()
        .paramId("Latitude Ki")
        .paramValue(latitudeRegulator.getKi()*1000)
        .paramType(MavParamType.MAV_PARAM_TYPE_REAL32)
        .paramIndex(4)
        .paramCount(9)
        .build();
        singleConnection.send2(systemId, componentId, paramValue4);
        ParamValue paramValue5=ParamValue.builder()
        .paramId("Latitude Kd")
        .paramValue(latitudeRegulator.getKd()*1000)
        .paramType(MavParamType.MAV_PARAM_TYPE_REAL32)
        .paramIndex(5)
        .paramCount(9)
        .build();
        singleConnection.send2(systemId, componentId, paramValue5);
        ParamValue paramValue6=ParamValue.builder()
        .paramId("Longitude Kp")
        .paramValue(longitudeRegulator.getKp()*1000)
        .paramType(MavParamType.MAV_PARAM_TYPE_REAL32)
        .paramIndex(6)
        .paramCount(9)
        .build();
        singleConnection.send2(systemId, componentId, paramValue6);
        ParamValue paramValue7=ParamValue.builder()
        .paramId("Longitude Ki")
        .paramValue(longitudeRegulator.getKi()*1000)
        .paramType(MavParamType.MAV_PARAM_TYPE_REAL32)
        .paramIndex(7)
        .paramCount(9)
        .build();
        singleConnection.send2(systemId, componentId, paramValue7);
        ParamValue paramValue8=ParamValue.builder()
        .paramId("Longitude Kd")
        .paramValue(longitudeRegulator.getKd()*1000)
        .paramType(MavParamType.MAV_PARAM_TYPE_REAL32)
        .paramIndex(8)
        .paramCount(9)
        .build();
        singleConnection.send2(systemId, componentId, paramValue8);
```

```
}
else if (m.getPayload() instanceof ParamSet)
{
         ParamSet ps =(ParamSet) m.getPayload();
         if (ps.paramId().equals("Altitude Kp"))
         {
                 altitudeKp=ps.paramValue();
                 Log.d("Connection",Float.toString(altitudeKp));
                 ParamValue paramValue=ParamValue.builder()
                 .paramId("Altitude Kp")
                 .paramValue(altitudeKp)
                 .paramType(MavParamType.MAV_PARAM_TYPE_REAL32)
                 .paramIndex(0)
                 .paramCount(9)
                 .build();
                 singleConnection.send2(systemId,componentId,paramValue);
         }
else if (ps.paramId().equals("Altitude Ki"))
{
         altitudeKi=ps.paramValue();
         Log.d("Connection",Float.toString(altitudeKi));
         ParamValue paramValue=ParamValue.builder()
         .paramId("Altitude Ki")
         .paramValue(altitudeKi)
         .paramType(MavParamType.MAV_PARAM_TYPE_REAL32)
         .paramIndex(1)
         .paramCount(9)
         .build();
         singleConnection.send2(systemId,componentId,paramValue);
}
else if (ps.paramId().equals("Altitude Kd"))
{
         altitudeKd=ps.paramValue();
         Log.d("Connection",Float.toString(altitudeKd));
         ParamValue paramValue=ParamValue.builder()
         .paramId("Altitude Kd")
         .paramValue(altitudeKd)
         .paramType(MavParamType.MAV_PARAM_TYPE_REAL32)
         .paramIndex(2)
         .paramCount(9)
         .build();
         singleConnection.send2(systemId,componentId,paramValue);
}
else if (ps.paramId().equals("Latitude Kp"))
{
         latitudeRegulator.setKp(ps.paramValue()/1000);
         ParamValue paramValue=ParamValue.builder()
         .paramId("Latitude Kp")
         .paramValue(latitudeRegulator.getKp()*1000)
         .paramType(MavParamType.MAV_PARAM_TYPE_REAL32)
         .paramIndex(3)
         .paramCount(9)
         .build();
         singleConnection.send2(systemId,componentId,paramValue);
}
else if (ps.paramId().equals("Latitude Ki"))
{
l        atitudeRegulator.setKi(ps.paramValue()/1000);
         ParamValue paramValue=ParamValue.builder()
         .paramId("Latitude Ki")
         .paramValue(latitudeRegulator.getKi()*1000)
         .paramType(MavParamType.MAV_PARAM_TYPE_REAL32)
         .paramIndex(4)
         .paramCount(9)
         .build();
         singleConnection.send2(systemId,componentId,paramValue);
}
else if (ps.paramId().equals("Latitude Kd"))
{
         latitudeRegulator.setKd(ps.paramValue()/1000);
         ParamValue paramValue=ParamValue.builder()
         .paramId("Latitude Kd")
         .paramValue(latitudeRegulator.getKd()*1000)
         .paramType(MavParamType.MAV_PARAM_TYPE_REAL32)
         .paramIndex(5)
         .paramCount(9)
         .build();
         singleConnection.send2(systemId,componentId,paramValue);
}
else if (ps.paramId().equals("Longitude Kp"))
{
         longitudeRegulator.setKp(ps.paramValue()/1000);
         ParamValue paramValue=ParamValue.builder()
         .paramId("Longitude Kp")
         .paramValue(longitudeRegulator.getKp()*1000)
         .paramType(MavParamType.MAV_PARAM_TYPE_REAL32)
         .paramIndex(6)
         .paramCount(9)
         .build();
         singleConnection.send2(systemId,componentId,paramValue);
}
else if (ps.paramId().equals("Longitude Ki"))
{
         longitudeRegulator.setKi(ps.paramValue()/1000);
         ParamValue paramValue=ParamValue.builder()
         .paramId("Longitude Ki")
         .paramValue(longitudeRegulator.getKi()*1000)
         .paramType(MavParamType.MAV_PARAM_TYPE_REAL32)
         .paramIndex(7)
         .paramCount(9)
         .build();
         singleConnection.send2(systemId,componentId,paramValue);
}
else if (ps.paramId().equals("Longitude Kd"))
{
         longitudeRegulator.setKd(ps.paramValue()/1000);
         ParamValue paramValue=ParamValue.builder()
         .paramId("Longitude Kd")
         .paramValue(longitudeRegulator.getKd()*1000)
         .paramType(MavParamType.MAV_PARAM_TYPE_REAL32)
         .paramIndex(8)
         .paramCount(9)
         .build();
         singleConnection.send2(systemId,componentId,paramValue);
}
}
else if (m.getPayload() instanceof CommandLong) {
CommandLong c = (CommandLong) m.getPayload();
if (c.command().entry() == MavCmd.MAV_CMD_DO_DIGICAM_CONTROL) {
Log.d("Message received", "MAV_CMD_DO_DIGICAM_CONTROL " + c.toString());
if (c.param5() == 1.0 || c.param7() == 1.0) {
         Log.d("Param", "Shooting Command");
         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();
         Photo photo=new Photo(mainContext);
         try {
                 photo.openCameraThread();
         }
         catch (CameraAccessException e)
         {
                 e.printStackTrace();
         }
singleConnection.send2(systemId, componentId, commandLong);
singleConnection.send2(systemId, componentId, commandAck);
```

```java
singleConnection.send2(systemId, componentId, cameraTrigger);
}
} else if (c.command().entry().equals(MavCmd.MAV_CMD_REQUEST_MESSAGE)) {
         Log.d("Message received", c.toString());
         AutopilotVersion autopilotVersion = AutopilotVersion.builder()
         .capabilities
         (MavProtocolCapability.
         MAV_PROTOCOL_CAPABILITY_FLIGHT_INFORMATION)
         .capabilities
         (MavProtocolCapability.
         MAV_PROTOCOL_CAPABILITY_SET_POSITION_TARGET_GLOBAL_INT)
         .capabilities
         (MavProtocolCapability.MAV_PROTOCOL_CAPABILITY_TERRAIN)
         .capabilities
         (MavProtocolCapability.MAV_PROTOCOL_CAPABILITY_MAVLINK2)
         .capabilities
         (MavProtocolCapability.MAV_PROTOCOL_CAPABILITY_COMMAND_INT)
         .osSwVersion(222)
         .flightSwVersion(111)
         .boardVersion(333)
         .productId(1)
         .build();
singleConnection.send2(systemId, componentId, autopilotVersion);
}
}
else if (m.getPayload() instanceof MissionCount)
{
         MissionCount ms= (MissionCount) m.getPayload();
         if (ms.targetComponent()==1 &&
         ms.targetSystem()==2 &&
         ms.missionType().entry().
         equals( MavMissionType.MAV_MISSION_TYPE_MISSION))
{
missionItemNumber=ms.count();
missionItemCounter=0;
MissionRequestInt missionRequestInt=MissionRequestInt.builder()
.targetSystem(255)
.targetComponent(244)
.seq(0)
.missionType(MavMissionType.MAV_MISSION_TYPE_MISSION)
.build();
singleConnection.send2(systemId,componentId,missionRequestInt);
}
}
else if (m.getPayload() instanceof MissionItemInt)
{
  MissionItemInt msItemInt=(MissionItemInt)m.getPayload();
  if (msItemInt.targetSystem()==2 && msItemInt.targetComponent()==1)
  {
  if (msItemInt.frame().entry().equals
               (MavFrame.MAV_FRAME_GLOBAL_RELATIVE_ALT))
  {
  if (msItemInt.command().entry().equals(MavCmd.MAV_CMD_NAV_TAKEOFF))
  {
         MissionStep missionStep= new MissionStep();
         missionStep.altitude=msItemInt.z();
         missionStep.what=What.TAKE_OFF;
         missionStepList.add(missionStep);
 }
else if (msItemInt.command().entry().equals(MavCmd.MAV_CMD_NAV_WAYPOINT))
{
         MissionStep missionStep=new MissionStep();
         missionStep.latitude=msItemInt.x();
         missionStep.longitude=msItemInt.y();
         missionStep.what=What.WAYPOINT;
         missionStepList.add(missionStep);
}
else if (msItemInt.command().entry().equals(MavCmd.MAV_CMD_NAV_LAND))
{
         MissionStep missionStep=new MissionStep();
         missionStep.what=What.LAND;
         missionStepList.add(missionStep);
}
}
if (msItemInt.seq()==missionItemNumber-1)
{
         MissionAck missionAck=MissionAck.builder()
```

```
               .targetSystem(255)
               .targetComponent(244)
               .type(MavMissionResult.MAV_MISSION_ACCEPTED)
               .missionType(MavMissionType.MAV_MISSION_TYPE_MISSION)
               .build();
               singleConnection.send2(systemId,componentId,missionAck);
               MissionStep ms=missionStepList.removeFirst();
               What w=ms.what;
               if (w==What.TAKE_OFF)
               {
                       Log.d("Connection","First take off discarded!!!");
                        ms=missionStepList.removeFirst();
                       w=ms.what;
               if (w==What.WAYPOINT)
               {
                       targetAltitude=altitude;
                       targetLatitude = ms.latitude;
                       targetLongitude = ms.longitude;
               }
               }
               else if (w==What.WAYPOINT)
               {
                       targetAltitude=altitude;
                       targetLatitude = ms.latitude;
                       targetLongitude = ms.longitude;
               }
               else
               {
                       Log.d("Connection","initial step wrong!");
               }
       }
      else if (missionItemCounter==msItemInt.seq())
      {
               missionItemCounter++;
               MissionRequestInt missionRequestInt=MissionRequestInt.builder()
               .targetSystem(255)
               .targetComponent(244)
               .seq(missionItemCounter)
               .missionType(MavMissionType.MAV_MISSION_TYPE_MISSION)
               .build();
               singleConnection.send2(systemId,componentId,missionRequestInt);
      }
   }
}
if (!holdingFromAutopilot)
{
      targetAltitude = altitude;
      targetLongitude = longitude;
      targetLatitude = latitude;
      missionStepList.clear();
}
if (targetAltitude < altitude + 0.1 &&
               targetAltitude > altitude - 0.1 &&
               targetLatitude < latitude + 5 &&
               targetLatitude > latitude - 5 &&
               targetLongitude < longitude + 5 &&
               targetLongitude > longitude - 5) {
       if (!missionStepList.isEmpty()) {
               MissionStep ms = missionStepList.removeFirst();
               What w = ms.what;
               if (w == What.WAYPOINT) {
                       targetAltitude = altitude;
                       targetLatitude = ms.latitude;
                       targetLongitude = ms.longitude;
                       Log.d("Connection", "WayPoint set");
               } else if (w == What.LAND) {
                       targetLatitude = latitude;
                       targetLongitude = longitude;
                       targetAltitude = altitudeZero;
               }
       }
 }
long dtGps=gpsTime-gpsAcquisitionTime;
if (dtGps!=0 && targetLatitude!=0 && targetLongitude!=0)
{
       float tempTargetPitch=latitudeRegulator.getInput
                               (targetLatitude,latitude,dtGps,false);
```

```
                float tempTargetRoll=longitudeRegulator.getInput
                                        (targetLongitude,longitude,dtGps,false);
                medianTargetRoll= (float)
                        (tempTargetRoll*Math.cos(Math.toRadians(-currentYaw))
                        -tempTargetPitch*Math.sin(Math.toRadians(-currentYaw)));
                medianTargetPitch=(float)
                        (tempTargetRoll*Math.sin(Math.toRadians(-currentYaw))
                        +tempTargetPitch*Math.cos(Math.toRadians(-currentYaw)));
         if (medianTargetPitch>ROLLPITCHMAX)
         {
                 gpsTargetPith=ROLLPITCHMAX;
         }
         else if (medianTargetPitch<-ROLLPITCHMAX)
         {
                 gpsTargetPith=-ROLLPITCHMAX;
         }
         else
         {
                 gpsTargetPith=medianTargetPitch;
         }
         if(medianTargetRoll>ROLLPITCHMAX)
         {
                 gpstargetRoll=ROLLPITCHMAX;
         }
         else if (medianTargetRoll<-ROLLPITCHMAX)
         {
                gpstargetRoll=-ROLLPITCHMAX;
          }
          else
          {
               gpstargetRoll=medianTargetRoll;
          }
          gpsAcquisitionTime=gpsTime;
        }
      }
}
} catch (EOFException eof) {
        Log.d("Connection","error1");
} catch (UnknownHostException e) {
        Log.d("Connection","error2");
        e.printStackTrace();
} catch (IOException e) {
        // Log.d("Connection","error timeout");
        if (!holdingFromAutopilot)
        {
                targetAltitude = altitude;
                targetLongitude = longitude;
                targetLatitude = latitude;
                missionStepList.clear();
        }
   long dtGps=gpsTime-gpsAcquisitionTime;
   if (dtGps!=0 && targetLatitude!=0 && targetLongitude!=0)
   {
        float tempTargetPitch=latitudeRegulator.getInput
                        (targetLatitude,latitude,dtGps,false);
        float tempTargetRoll=longitudeRegulator.getInput
                        (targetLongitude,longitude,dtGps,false);
        medianTargetRoll= (float) (tempTargetRoll*Math.cos(Math.toRadians(-currentYaw))
                                -tempTargetPitch*Math.sin(Math.toRadians(-currentYaw)));
        medianTargetPitch=(float) (tempTargetRoll*Math.sin(Math.toRadians(-currentYaw))
                                +tempTargetPitch*Math.cos(Math.toRadians(-currentYaw)));
        if (medianTargetPitch>ROLLPITCHMAX)
        {
                gpsTargetPith=ROLLPITCHMAX;
        }
        else if (medianTargetPitch<-ROLLPITCHMAX)
        {
                gpsTargetPith=-ROLLPITCHMAX;
        }
         else
        {
                gpsTargetPith=medianTargetPitch;
        }
        if(medianTargetRoll>ROLLPITCHMAX)
        {
                gpstargetRoll=ROLLPITCHMAX;
        }
        else if (medianTargetRoll<-ROLLPITCHMAX)
        {
                gpstargetRoll=-ROLLPITCHMAX;
        }
        else
        {
                gpstargetRoll=medianTargetRoll;
        }
        gpsAcquisitionTime=gpsTime;
     }
   e.printStackTrace();
   }
  }
}
```

A.2.2 SendDataToGCS

sendSensorsData

```
public void sendSensorsData(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-senso
        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( int level){
    BatteryStatus batteryStatus=BatteryStatus.builder().batteryRemaining(level).build();
    new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    if(connection!=null) {
                        connection.send2(systemId, componentId, batteryStatus);
                    }
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }).start();
    }
```

sendGpsData

```
public void sendGpsData( 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(){
        Heartbeat heartbeat;
        boolean connectionActive = true;
        while (connectionActive) {
            try {
                sleep(1000);
                heartbeat = Heartbeat.builder()
                        .type(MavType.MAV_TYPE_QUADROTOR)
                        .autopilot(MavAutopilot.MAV_AUTOPILOT_GENERIC)
                        .baseMode(MavModeFlag.MAV_MODE_FLAG_GUIDED_ENABLED)
                        .systemStatus(MavState.MAV_STATE_ACTIVE)
                        .mavlinkVersion(2)
                        .build();
                int systemId = 2;
                int componentId = 1;
                connection.send2(systemId, componentId, heartbeat);
            } catch (IOException e) {
                e.printStackTrace();
                connectionActive = false;
            } catch (InterruptedException e) {
                e.printStackTrace();
                connectionActive = false;
            }
        }
    }
```

A.2.3 Photo

Photo constructor

```
@RequiresApi(api = Build.VERSION_CODES.LOLLIPOP)
public Photo(Context context) {
this.context = context;
this.connectionVideoToGCS=new ConnectionVideoToGCS(context);
cameraManager = (CameraManager) context.getSystemService(Context.CAMERA_SERVICE);
try {
     if (cameraManager!=null) {
     cameraList = cameraManager.getCameraIdList();
      for (String cameraID : cameraList) {
      CameraCharacteristics cameraCharacteristics =
                        cameraManager.getCameraCharacteristics(cameraID);
      Integer facing = cameraCharacteristics.get(CameraCharacteristics.LENS_FACING);
      if (facing == CameraCharacteristics.LENS_FACING_BACK) {
           StreamConfigurationMap streamConfigurationMap =
                cameraCharacteristics.get
                        (CameraCharacteristics.SCALER_STREAM_CONFIGURATION_MAP);
                if (streamConfigurationMap.isOutputSupportedFor(ImageFormat.JPEG)) {
                    Size[] sizes =
                        streamConfigurationMap.getOutputSizes(ImageFormat.JPEG);
                    for (Size size : sizes) {
                           if (size.getHeight() > maxHeightPixels) {
                                   maxHeightPixels = size.getHeight();
                                   camID = cameraID;
                                }
                                if (size.getWidth() > maxWidthPixels) {
                                    maxWidthPixels = size.getHeight();
                                    camID = cameraID;
                                }
                            }
                        }
                    }
                }
            }
            imageReader = ImageReader.newInstance
                        (maxWidthPixels, maxHeightPixels, ImageFormat.JPEG, 3);
        } catch (CameraAccessException e) {
            e.printStackTrace();
        }
    }
```

openCameraThread

```
public void openCameraThread() throws CameraAccessException {
handlerThread=new HandlerThread("CameraThread");
handlerThread.start();
handler=new Handler(handlerThread.getLooper());
imageReader.setOnImageAvailableListener(imageAvailableListener,handler);
if (ContextCompat.checkSelfPermission
                (context, Manifest.permission.CAMERA) ==
                                PackageManager.PERMISSION_GRANTED) {
        cameraManager.openCamera(camID, stateCallback, handler);
}
else
{
    Log.d("Camera","no permission");
}
}
```

stateCallback

```
private final CameraDevice.StateCallback stateCallback=new CameraDevice.StateCallback() {
        @Override
        public void onOpened(@NonNull CameraDevice camera) {
            List<Surface> surfaceList=new ArrayList<>();
            surfaceList.add(imageReader.getSurface());
            cameraDevice=camera;
            try {
                camera.createCaptureSession(surfaceList,captureSessionCallback,null);
            } catch (CameraAccessException e) {
                e.printStackTrace();
            }
        }
        @Override
        public void onDisconnected(@NonNull CameraDevice camera) {
            handlerThread.quit();
        }
        @Override
        public void onError(@NonNull CameraDevice camera, int error) {
            Log.d("Camera","error opening camera");
        }
    };
```

captureSessionCallback

```
private final CameraCaptureSession.StateCallback captureSessionCallback=
new CameraCaptureSession.StateCallback() {
@Override
public void onConfigured(@NonNull CameraCaptureSession session) {
cameraCaptureSession=session;
try {
     captureRequestBuilder=cameraDevice.createCaptureRequest(
        CameraDevice.TEMPLATE_STILL_CAPTURE);
     captureRequestBuilder.addTarget(imageReader.getSurface());
     session.capture(captureRequestBuilder.build(),null,null);
     } catch (CameraAccessException e) {
                e.printStackTrace();
     }
}
        @Override
        public void onConfigureFailed(@NonNull CameraCaptureSession session) {
        }
    };
```

imageAvailableListener

```
 private final ImageReader.OnImageAvailableListener imageAvailableListener=
                                        new ImageReader.OnImageAvailableListener(){
        @Override
        public void onImageAvailable(ImageReader reader) {
            Image image=reader.acquireNextImage();
            ByteBuffer byteBuffer=image.getPlanes()[0].getBuffer();
            byte[] bytes=new byte[byteBuffer.remaining()];
            byteBuffer.get(bytes);
            File sd =
                new File(context.getExternalFilesDir
                (Environment.DIRECTORY_PICTURES),"PictureOnAir");
            if(!sd.exists()) {
                sd.mkdirs();
            }
            Calendar cal = Calendar.getInstance();
            SimpleDateFormat sdf = new SimpleDateFormat("yyyyMMdd_HHmmss");
            String tar = (sdf.format(cal.getTime()));
            try {
                FileOutputStream outStream = new FileOutputStream(sd+"/"+tar+".jpg");
                outStream.write(bytes);
            } catch (FileNotFoundException e) {
                e.printStackTrace();
            } catch (IOException e) {
                e.printStackTrace();
            }
            image.close();
            imageReader.close();
            cameraDevice.close();
            connectionVideoToGCS.start();
        }
    };
```

A.2.4 ConnectionVideoToGCS

ConnectionVideoToGCS thread

```
public void run() {
super.run();
File sd = new File(context.getExternalFilesDir
                      (Environment.DIRECTORY_PICTURES), "PictureOnAir");
if(!sd.exists()) {
     sd.mkdirs();
    Log.i("FO", "folder" + context.getExternalFilesDir(Environment.DIRECTORY_PICTURES));
    }
    Calendar cal = Calendar.getInstance();
    SimpleDateFormat sdf = new SimpleDateFormat("yyyyMMdd_HHmmss");
    String tar = (sdf.format(cal.getTime()));
    String path= sd.getAbsolutePath();
     int rc=FFmpeg.execute("-loglevel trace -f android_camera -camera_index 0
               -i 0:0 -pix_fmt yuv420p -s 320x240 -c:v libx264 " +
               "-preset ultrafast -tune zerolatency " +
               "-f rtp rtp://192.168.43.151:5600" //+
      // " -pix_fmt yuv420p -s 1280x720 -crf 19
     //      -preset ultrafast -c:v libx264 "+path+"/"+tar+".mkv"
     );
     if (rc!=RETURN_CODE_SUCCESS)
     {
         Log.d("video",Integer.toString(rc));
         Config.printLastCommandOutput(Log.INFO);
     }
    }
```

Appendix B

Arduino Code

```
#include "SBUS.h"
#include <Wire.h>
#define SBUS_MIN 172
#define SBUS_MAX 1815
uint8_t target_roll;
uint8_t target_pitch;
uint8_t target_yaw;
uint8_t target_altitude;
byte sw;
byte se;
byte ne;
byte nw;
int swNoSmartphone;
int nwNoSmartphone;
int seNoSmartphone;
int neNoSmartphone;
volatile bool flagNoSmartphone;
bool flagStart;
byte tx[6];
int lengthOfRx;
// a SBUS object, which is on hardware
// serial port 3
SBUS receiver(Serial3);
// channel, fail safe, and lost frames data
uint16_t channels[16];
bool failSafe;
bool lostFrame;
int nrx;
byte received;
void setup() {
  cli();
  receiver.begin();
  Wire.begin();
  Serial.begin(115200);
  target_roll = 0;
  target_pitch = 0;
  target_yaw = 0;
  target_altitude = 0;
  flagNoSmartphone=0;
  flagStart=0;
  sw = 0;
  se = 0;
  ne = 0;
  nw = 0;
swNoSmartphone=0;
nwNoSmartphone=0;
seNoSmartphone=0;
neNoSmartphone=0;
nrx=0;
received=0;
  lengthOfRx=0;
  pinMode(9, OUTPUT);
  pinMode(10, OUTPUT);
  pinMode(13,OUTPUT);
  TCCR5A = 0;
  TCCR5B = 0;
  TCCR5B |= (0 << CS52) | (1 << CS51) | (0 << CS50);
  TCNT5  = 0x0000;
  bitSet(TIMSK5, TOIE5);
  sei();
}
void loop() {
  // look for a good SBUS packet from the receiver
  if (receiver.read(&channels[0], &failSafe, &lostFrame)) {
    if (channels[8] >= SBUS_MIN && channels[8]  <= SBUS_MAX)
    {
      target_roll = map(channels[8] , SBUS_MIN, SBUS_MAX, 0, 255);
    }
    if (channels[2] >= SBUS_MIN && channels[2]  <= SBUS_MAX)
    {
      target_pitch = map(channels[2] , SBUS_MIN, SBUS_MAX, 0, 255);
    }
    if (channels[6] >= SBUS_MIN && channels[6]  <= SBUS_MAX)
    {
      target_altitude = map(channels[6] , SBUS_MIN, SBUS_MAX, 0, 255);
    }
    if (channels[7] >= SBUS_MIN && channels[7]  <= SBUS_MAX)
    {
      target_yaw = map(channels[7] , SBUS_MIN, SBUS_MAX, 0, 255);
    }
    if (target_pitch==0 && target_altitude==0)
      {
        flagStart=1;
      }
    if (flagNoSmartphone==0)
    {
      tx[1] = (byte) target_roll;
      tx[2] = (byte) target_pitch;
      tx[3] = (byte) target_yaw;
      tx[4] = (byte) target_altitude;
      if (channels[0]==1401)
      {
        tx[0] = (byte) 2;
        tx[5] = (byte) 2;
        Serial.write(tx, sizeof(tx));
      }
      else if (channels[0]==992)
      {
        tx[0] = (byte) 1;
        tx[5] = (byte) 1;
        Serial.write(tx, sizeof(tx));
      }
      else if (channels[0]==582)
      {
        tx[0] = (byte) 0;
        tx[5] = (byte) 0;
        Serial.write(tx, sizeof(tx));
      }
    }

  }
  byte rx[5];
  while(Serial.available() > 0 ){
    TCNT5 = 0;
    flagNoSmartphone=0;
    received=Serial.read();
    if (received==(byte)0)
    {
        if (nrx==4)
        {
          sw = (byte) rx[0];
          se = (byte) rx[1];
          ne = (byte) rx[2];
          nw = (byte) rx[3];
        }
        nrx=0;
    }
    else
    {
      rx[nrx]=received;
      nrx++;
    }
  }
  if (flagNoSmartphone==1)
  {
      nwNoSmartphone=(target_altitude-(target_pitch-128)/2+
                (target_roll-128)/2-(target_yaw-128)/2)-20-6;
      swNoSmartphone=(target_altitude+(target_pitch-128)/2+
                  (target_roll-128)/2+(target_yaw-128)/2)-20;
      neNoSmartphone=(target_altitude-(target_pitch-128)/2-
                  (target_roll-128)/2+(target_yaw-128)/2)-20-6;
      seNoSmartphone=(target_altitude+(target_pitch-128)/2-
                  (target_roll-128)/2-(target_yaw-128)/2)-20;
      if (nwNoSmartphone>255)
      {
        nwNoSmartphone=255;
      }
      else if(nwNoSmartphone<0)
      {
        nwNoSmartphone=0;
      }
      if (swNoSmartphone>255)
      {
        swNoSmartphone=255;
      }
      else if(swNoSmartphone<0)
      {
        swNoSmartphone=0;
      }
      if (neNoSmartphone>255)
      {
        neNoSmartphone=255;
      }
      else if(neNoSmartphone<0)
      {
        neNoSmartphone=0;
      }
      if (seNoSmartphone>255)
      {
        seNoSmartphone=255;
      }
      else if(seNoSmartphone<0)
      {
        seNoSmartphone=0;
      }
      nw=(byte)nwNoSmartphone;
      sw=(byte)swNoSmartphone;
      ne=(byte)neNoSmartphone;
      se=(byte)seNoSmartphone;
}
  if (target_altitude < 20 || flagStart==0)
  {
    sw = 0;
    se = 0;
    ne = 0;
    nw = 0;
  }

  Wire.beginTransmission(0x29);
  Wire.write(ne);
  Wire.endTransmission();
  Wire.beginTransmission(0x2a);
  Wire.write(sw);
  Wire.endTransmission();
  Wire.beginTransmission(0x2b);
  Wire.write(nw);
  Wire.endTransmission();
  Wire.beginTransmission(0x2c);
  Wire.write(se);
  Wire.endTransmission();

}
ISR(TIMER5_OVF_vect)
{
    cli();
    flagNoSmartphone=1;
    sei();
}
```