On an Android-based Arduino-governed unmanned Quadcopter platform: The CDIO Academy case

Thesis by: Carles Carruesco Picas

Advisors: Eduard Alarcón Cot
Elisenda Bou Balust

Escola Tècnica Superior d'Enginyeria de Telecomunicació de Barcelona
UNIVERSITAT POLITÈCNICA DE CATALUNYA

Barcelona

October 2014

Abstract

This thesis covers the design, implementation and application of a capstone project surrounding Air-Sensing Engines in Smart Cities. The task is to achieve autonomous control of an Unmanned Air Vehicle (a quadcopter; small four rotor helicopter) using an onboard smartphone and an Arduino board with air quality sensors.

It is divided in three main sections: design process, project platform and application.

Design process follows all the process from initial idea to final working prototype; choosing an aircraft and how to control it, how the software is designed to allow easy tinkering and adaptation and all the tests performed to ensure the proper functioning.

Project platform is a reference book for developers and anyone who wants to use this platform. It contains in depth descriptions of how each part works.

Application shows how this capstone project was successfully used in the CDIO Academy 2014 as a multidisciplinary challenge.

Acknowledgments

I want to first thank my colleague Gonzalo Martínez for all his work and dedication on the hardware side: choosing and calibrating sensors, prototyping with different configurations and building all the final units. This project would not have succeeded without him.

Also thank my thesis directors Eduard Alarcón and Elisenda Bou for their continued support and assistance throughout the project; especially Elisenda, who also developed and maintained the web server for the *GroundStation* and its web interface.

Finally, thank Carles Araguz and David Rodríguez for all their help organizing and running the CDIO Academy event.

Table of contents

List of figures

1: Introduction

1.1: Rationale

Unmanned Aerial Vehicles are a hot topic these days; companies like Amazon and Google are preparing to offer delivery services (Amazon Prime Air [1] and Google's Project Wing [2]), film producers are starting to use multirotor helicopters to capture aerial images [3] and, once regulations are finished, many more commercial applications will arise [4].

Since it is a relatively new field, there are many research possibilities. Also, although UAVs are often related to military uses, there are a lot of possibilities and civil applications waiting to be developed. For this reason, it is a very interesting topic to work on.

Another key aspect is the multidisciplinary nature of this topic. Designing multirotor helicopters requires mechanical engineers and aerodynamic engineers to properly stabilize it, this stabilization has then to be translated into software by computer science engineers.

All this just for flying, then comes application development which will need expertise in its field to properly incorporate it in a UAV. The possibilities for applications are truly endless: consumer oriented like fast package delivery, remote sensing with any sensor you can fit in the unit (irrigation control of large plantations, environmental studies, traffic monitoring, etc.) and emergency and disaster response like scouting for survivors, building temporary communication networks and navigating through hard terrain like mountains or debris.

This capstone project illustrates the design process and system integration with an air sensing application: designing the unit and its control systems as well as working with two important aspects of UAVs: autonomous navigation and working with sensors.

With this sample application, this capstone project portrays the possibilities of UAVs and provides an entry point to work and research within this topic.

1.2: Objectives

The aim is to create a generic UAV platform and use it in an air sensing application that navigates through a city and gets air quality and pollution readings.

This platform will have to be easily reproduced and thus, all the documentation will be public and freely available; the source code being already available in a public GitHub repository [5].

It is also important that in can be used for a different application so both hardware and software will have to be easily modifiable.

Finally, this project will also be used as a challenge in the CDIO Academy 2014, where teams of interdisciplinary students from around the world will compete to build the best implementation. The platform will have to be adapted to open some degrees of freedom and design choices for the challenge.

2: Design process

2.1: The goal

The goal of this capstone project is to have an autonomous quadcopter that follows a predefined set of waypoints, makes measurements of air quality, takes photos and uploads all the data to a public server.

There are many ways to achieve this goal. Our solution consist of an Android smartphone and an Arduino board; the smartphone handles navigation and makes decisions while the Arduino board houses all the required sensors and controls the quadcopter (according to the orders of the smartphone).

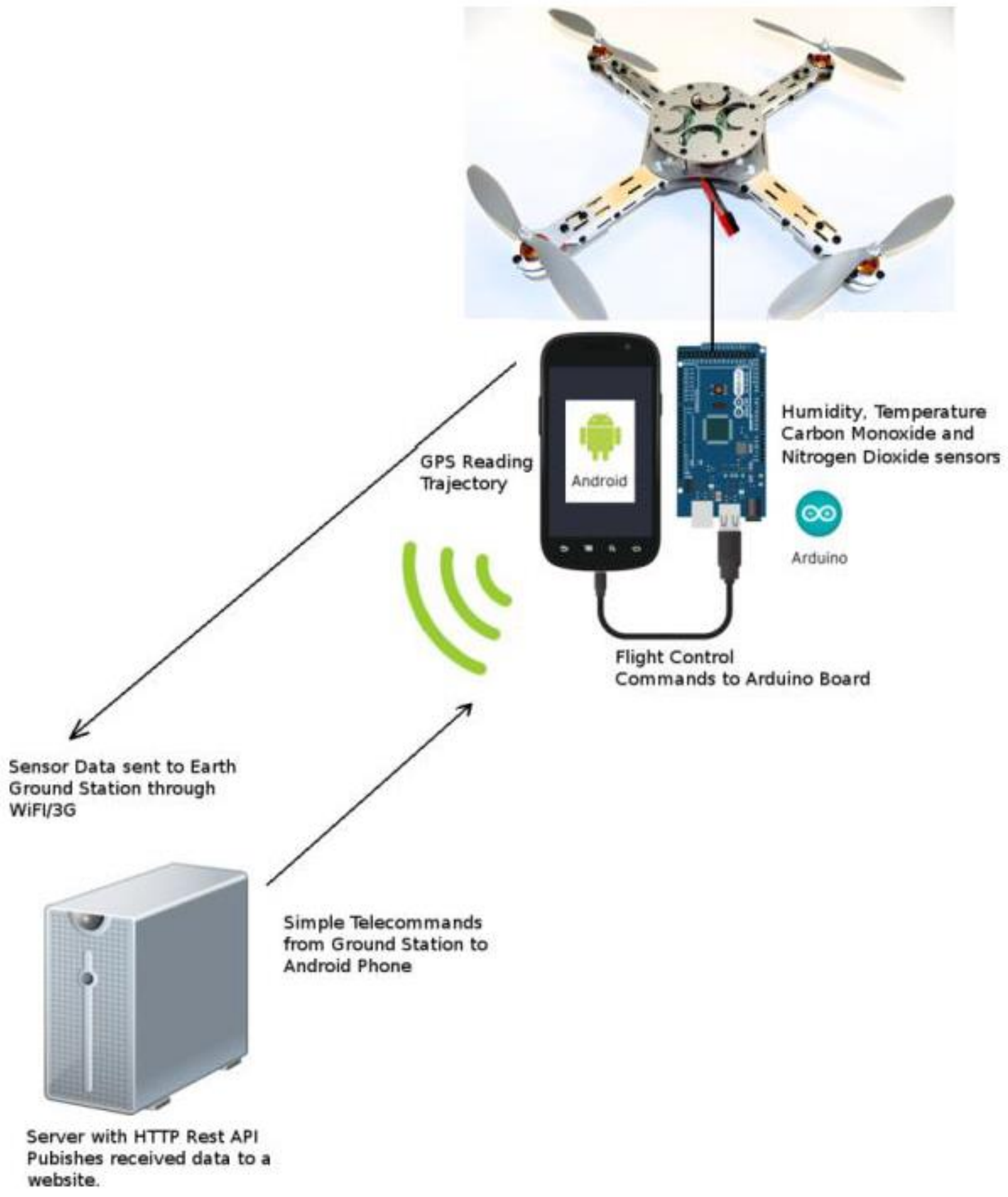


Figure 1: Overview of the goal

What follows is the design process and all the choices we have made to get to this solution.

2.2: How to control a quadcopter

First and most important step is how to communicate with the quadcopter. All of them, even DIY solutions (more on that later), have a flight control unit that takes care of stabilization; some also include assisted flight modes like locked altitude (usually called altitude hold).

Interacting with these flight control units requires firmware modifications and, while some of them are open source and thus it's possible to make said modifications, it would require a long time; time we didn't have. Instead, we went for a simpler approach; the smartphone would emulate a person controlling the quadcopter, in other words, it would emulate an RC. With this solution the firmware requires no modifications as it just sees a regular RC.

The Arduino board that hosts all the required sensors would also take care of RC emulation; more on this later.

2.3: Choosing the unit. Commercial versus DIY

The quadcopter can either be a commercial product or a DIY unit for the participants to build. This choice has a major impact on both time and skills required, as building it is no easy task.

Given the time constraints of the challenge, using a DIY solution was not an option. That said, a DIY solution can still be used for this capstone, if the application allows it, because its flight control unit is very similar, if not identical, to most commercial units.

2.3.1: Requirements

The unit, either commercial or DIY, will have to include a flight control unit with these capabilities:

- Automatic takeoff and landing
- Stabilized flight mode that allows easy horizontal and vertical movement
- Enough lift power to carry the payload (Arduino board, sensors and smartphone)
- (Not strictly required but strongly recommended) Safety features such as automatic landing when battery is low and ability to set a flight zone the unit can't leave.

2.3.2: 3DR Iris

The unit chosen for this project is the Iris from 3D Robotics [6][1], a Ready-To-Fly commercial solution.

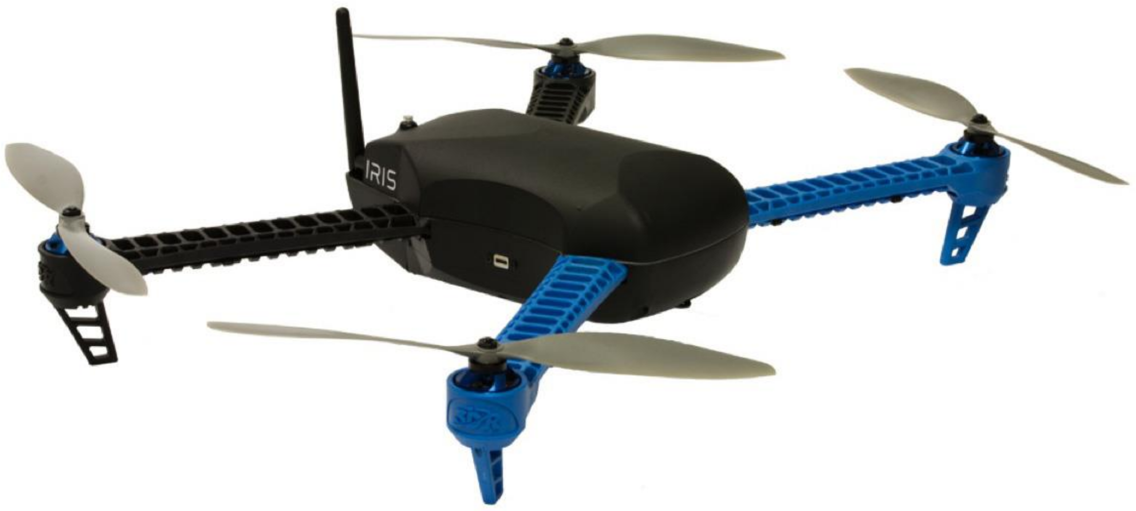


Figure 2: The 3DR Iris

Specifications:

- Motor to motor dimension: 550 mm
- Height: 100 mm
- Weight (with battery): 1282 g
- Average flight time: 9 - 14 minutes
- 400 g payload capacity
- Battery: 3-cell 11.1 V 3.5 Ah lithium polymer with XT-60 type connector. Weight: 262 g
- Propellers: (2) 10 x 4.7 normal-rotation, (2) 10 x 4.7 reverse-rotation
- Motors: AC 2830, 850 kV
- Telemetry/Control radios available in 915 MHz or 433 MHz
- 32-bit Pixhawk autopilot system with Cortex M4 processor (flight control unit)
- GPS receiver with integrated magnetometer

It fulfills all the requirements and includes the useful loiter flight mode that locks the unit in space. The 400 g payload capacity is enough for this project and the battery lasts for two missions.

2.4: Arduino RC emulation

RC receivers usually have PWM or PPM outputs; the main difference being PWM uses one output per channel whereas PPM multiplexes all the channels into a single output. The unit used for this project uses a PPM receiver. For more information on PPM see appendix *1: Pulse Position Modulation*.

The program developed for the Arduino board generates the PPM stream of the emulated RC and sends it to the Flight Control Unit as if it was a regular receiver. This, however, leaves all control to the smartphone and its autonomous flight program, which could be a huge problem if said program misbehaves. To solve this issue we have to be able to get manual control over the aircraft at any time.

[1] The Iris is no longer available; instead, 3DR offers the Iris+, an improved model.

2.4.1: Manual override

The manual override, or manual switch, is the security feature which guarantees that, no matter what the autonomous program is doing, we can always get manual control.

We connected the original RC receiver to one of the Arduino inputs and used a two position switch, in a non-used channel, to switch between manual and automatic, autonomous, mode.

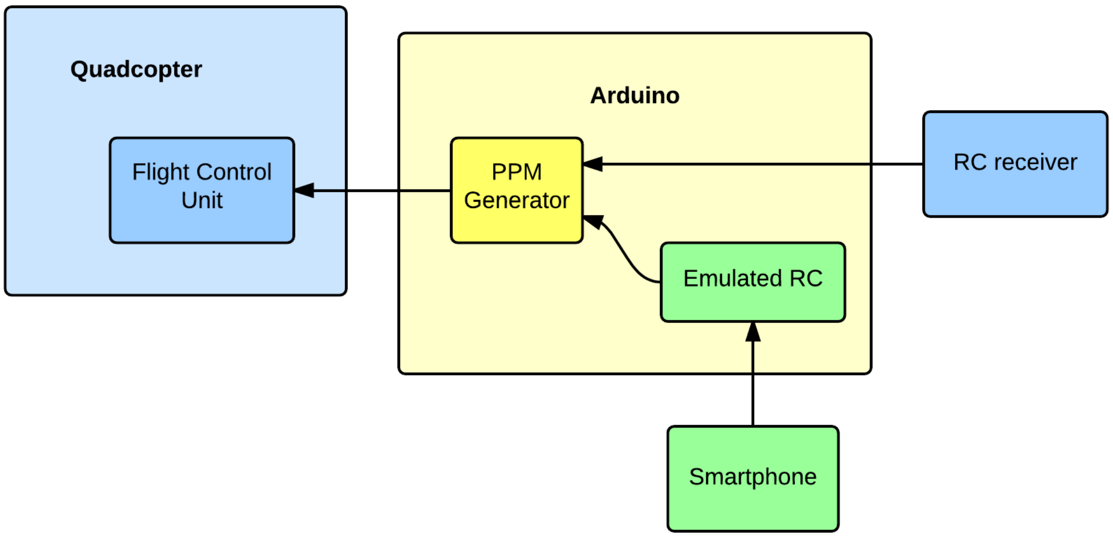


Figure 3: RC Emulation scheme

The Arduino is constantly reading form the real RC and generates an output stream accordingly. See *3.2.2.1.1: RC emulation* for more information on how this is accomplished.

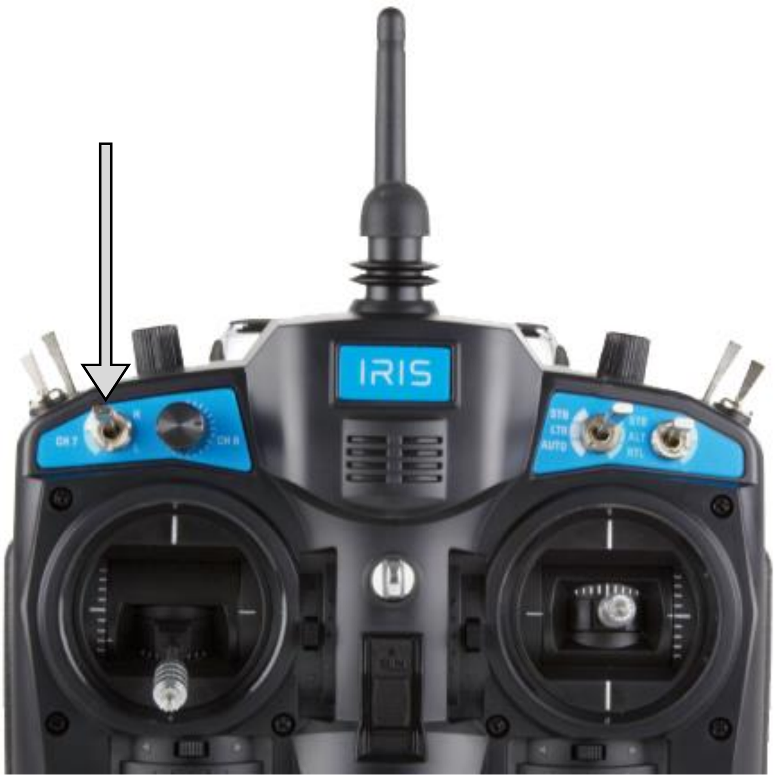


Figure 4: Manual override switch

2.5: Communicating with the Arduino board

The smartphone communicates with the Arduino board to “move” the virtual RC and perform measurements with the sensors. This communication narrows the smartphone platform choice down to Android; here is how: the other major mobile OSs, Apple’s iOS and Microsoft’s Windows Phone, can only talk with Arduino through a wireless link, usually Bluetooth or Bluetooth Low Energy, while Android also has the possibility to connect via USB; Arduino acting as a USB host.

Using a wired link offers two important advantages:

- More reliable than a wireless link in terms of error rate.
- All the system runs off a single battery. Arduino is powered by the quadcopter’s battery and also powers the smartphone through USB (because it is the host).

Google released its Accessory Development Kit (ADK) back at Google IO 2011 [7]; it was a modified Arduino board with sample source code to communicate with it. This is the framework we used to implement the communication link. For more information on Android’s ADK, see its developer guide [8] or consult Beginning Android ADK with Arduino [9], a much recommended book.

Note that the Arduino board must have a USB Host interface; you can use a regular Arduino with a USB Host shield or the Arduino MEGA ADK, based on the MEGA variant of Arduino with the USB Host interface built in. We used the latter for simplicity.

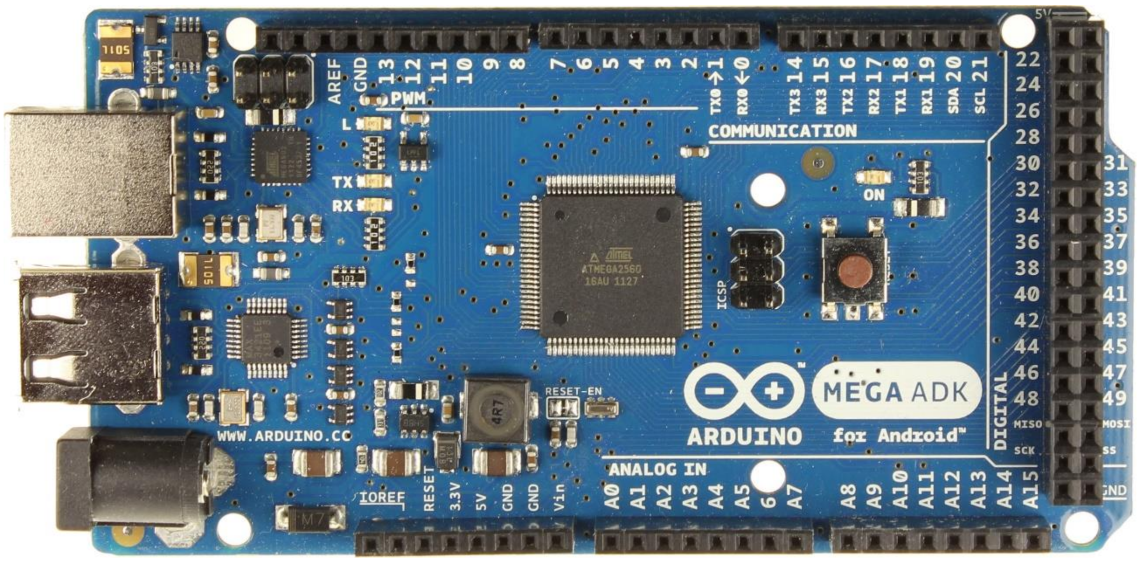


Figure 5: Arduino MEGA ADK board

It is also important to take into consideration the limited processing power of Arduino. To not waste resources, we designed a custom communication protocol focused on simplicity and short messages. See appendix *2: Arduino – Android communication protocol* for more information.

2.6: First tests

At this point we had the first prototype built and a preliminary version of the software, time for some tests.

With some simple flight mission we tested all the different systems: sensors, manual override, navigation algorithm and the quadcopter's own automatic modes[2]. These tests revealed several critical issues.

First, the material used to add housing space for the added components, polymethyl methacrylate, was not strong enough.

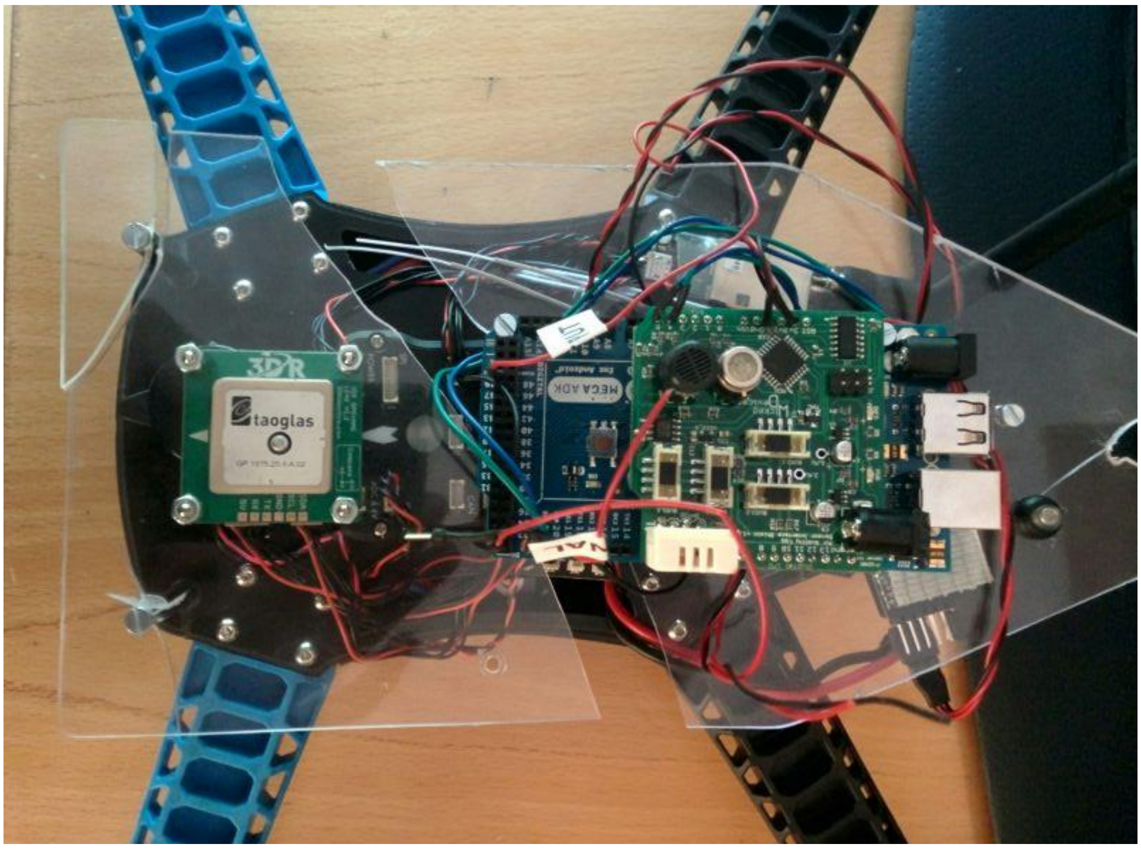


Figure 6: Broken support panel

[2] We use autonomous flight modes of the on board Flight Control Unit for automatic takeoff and landing; much safer than trying to do it with the smartphone.

We needed a more resistant panel, either the same material but thicker or another material; because of supplier problems, we opted for a different material, wood-like.

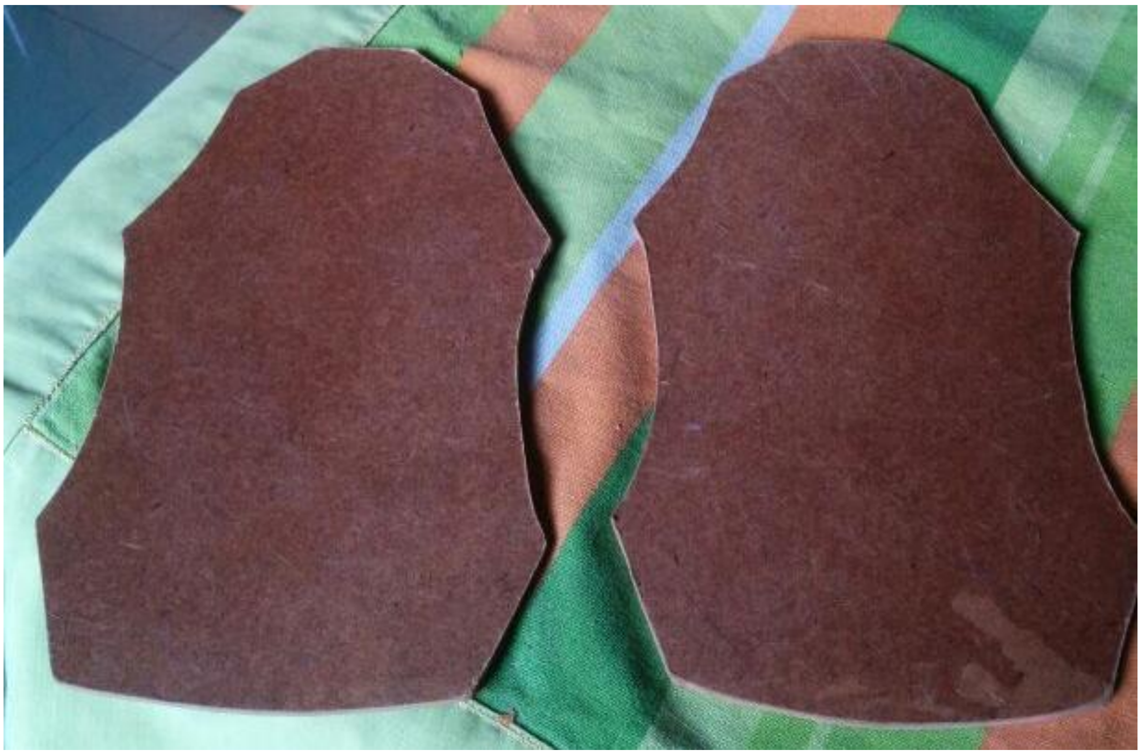


Figure 7: The new support panels

The second most important issue, smartphone's compass was not accurate enough and thus, could not be used for navigation. We found a workaround by relaying on the assisted flight modes of the quadcopter, which has a much better compass; this modes lock orientation so the unit always points in the same direction.

What we did was align the unit to North before every take off. This way latitude and longitude coordinates translate to forward/backward and left/right movements.

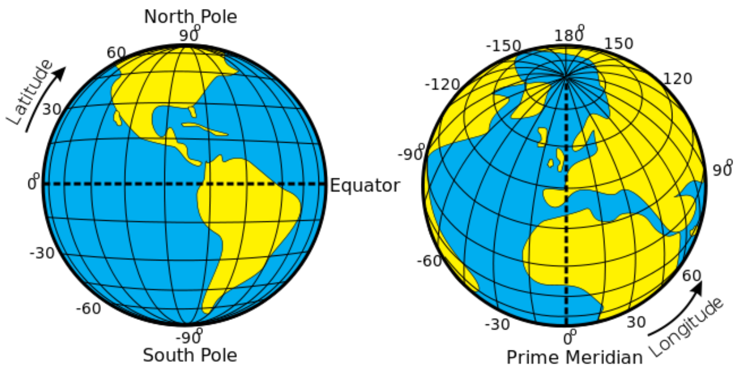


Figure 8: Latitude and longitude coordinates

Forward:	Latitude ↑	Right:	Longitude ↑
Backward:	Latitude ↓	Left:	Longitude ↓

Another problem was the bad reception of the unit's GPS receiver, especially with the new panels. Without a proper GPS lock, the automatic modes did not work.

To fix this issue we changed the GPS module position from under the panel to on top of it and elevated. See the following two images of before and after for comparison; the before image is an early prototype with the old panels (you can see how it was starting to break at the front), the after picture is from the final model.

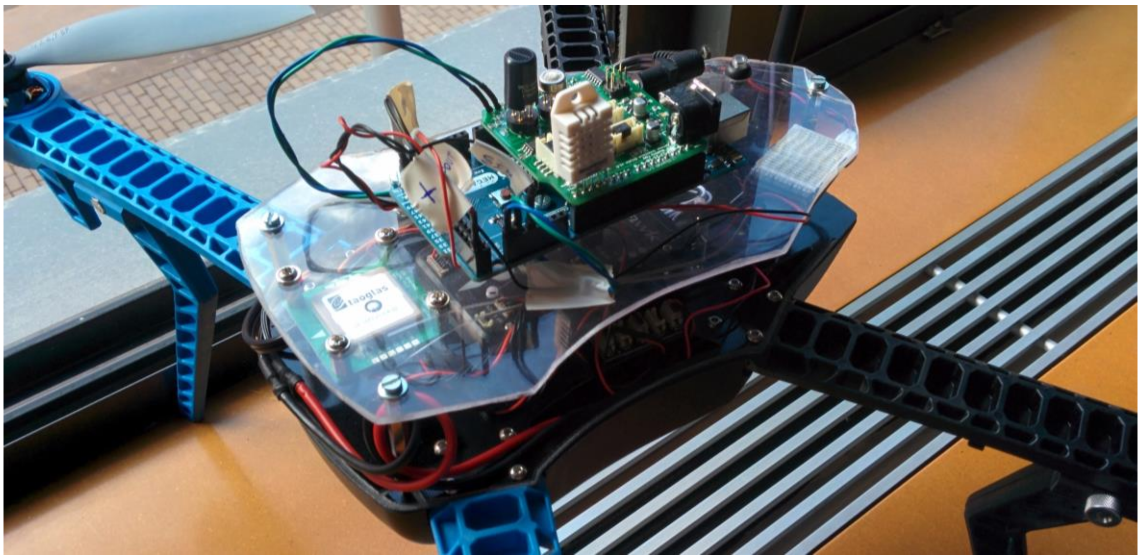


Figure 9: Early prototype with GPS module under the support panel

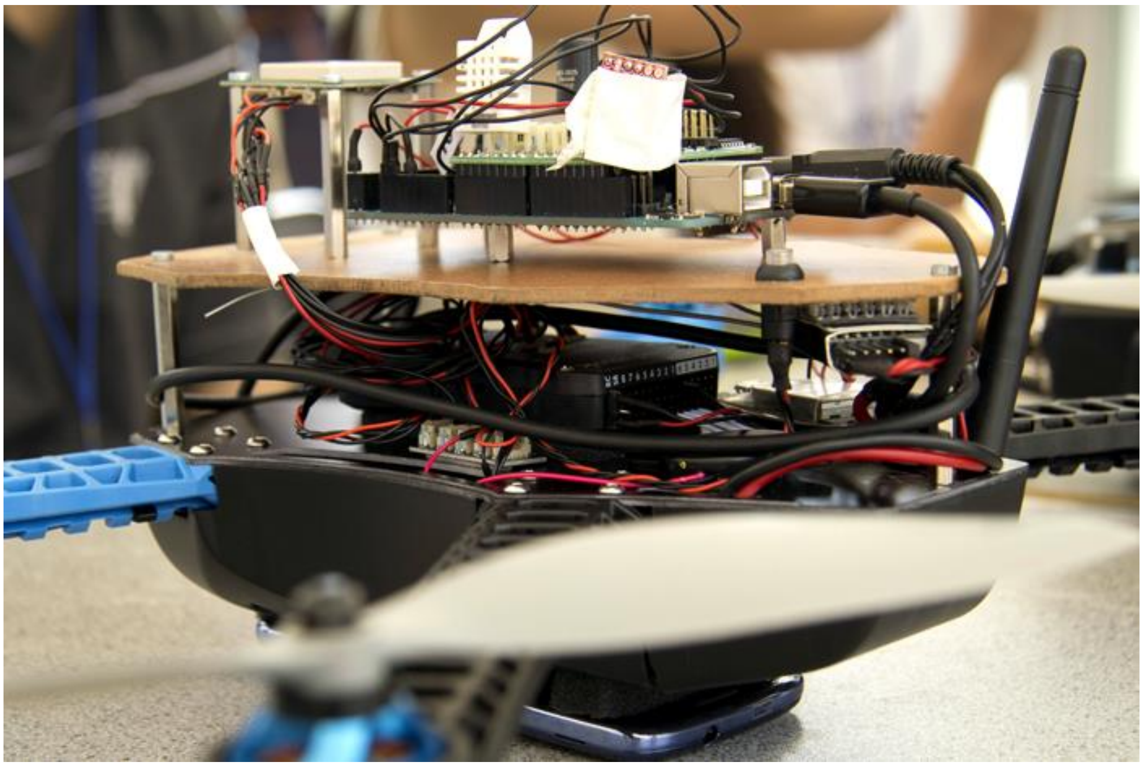


Figure 10: Final model with GPS module elevated

Next, we also identified a critical issue with the manual override: given how the Arduino program worked, if it was overloaded or stuck because of bad code the manual override would not work. Unacceptable; the manual override must work at all times under any circumstances. We remade from scratch how it reads and generates RC signals so that it ran with interruptions, immune to overload.

Another issue detected was a vibration of the smartphone that resulted in a wobbly effect in the photos. For these first tests we were using a rigid support to hold the smartphone in place. We fixed this problem by holding the smartphone with Velcro ties at the bottom of the unit; this change also allowed us to shorten the legs of the quadcopter, decreasing its center of mass and making it more stable, especially when landing.

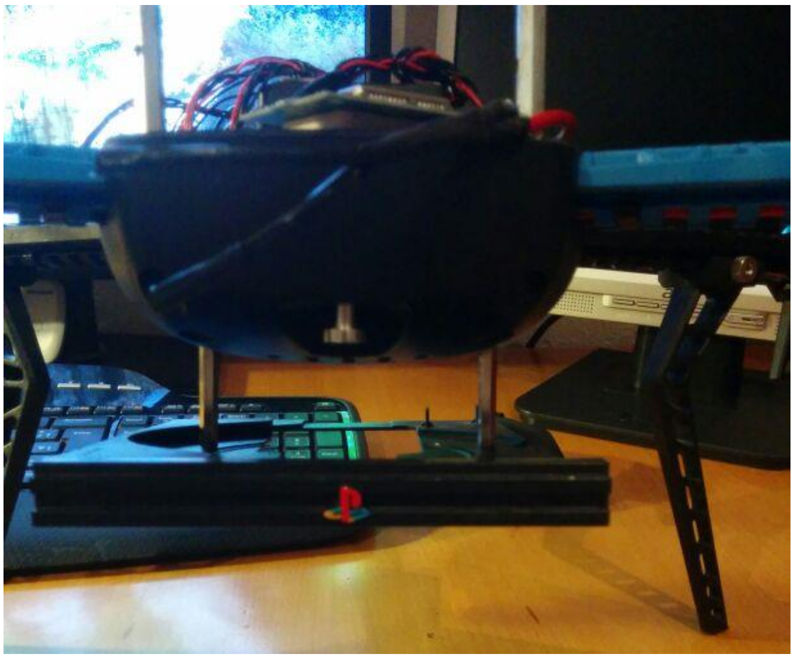


Figure 11: Very early prototype of smartphone support

These two photos show the difference between the old support and the Velcro ties, the lines should all be straight.

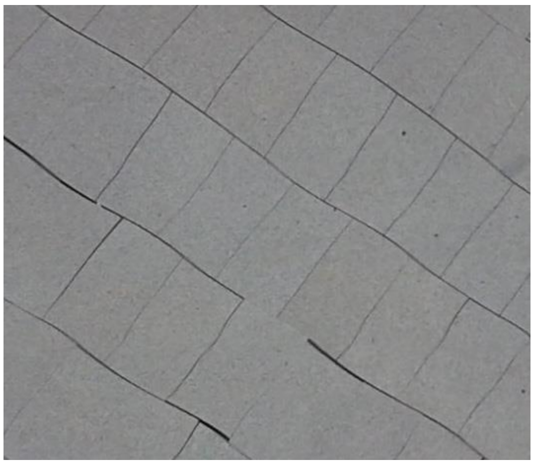


Figure 12: Wobbly effect with the old support

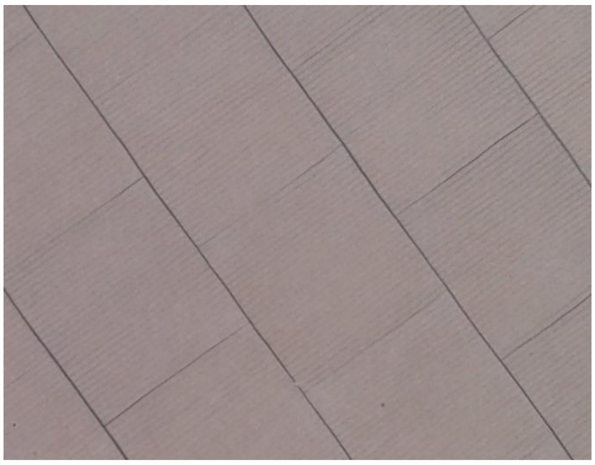


Figure 13: Wobbly effect with the new support

With the new support the effect is small enough to neglect it.

Another effect to take into account regarding the camera, if you take photos while moving the result will be distorted since the unit is tilted. To take good pictures you have to give it some time to stabilize.

Finally, at this point we were using a custom Java server with a rudimentary connection; during the tests it was made clear that it had to be scrapped as the connection would constantly drop. It was substituted with a web server and standard HTTP; much more reliable.

2.7: Sensors

Regarding the sensors, we needed air quality readings and Arduino compatibility. Since the Arduino platform is widely used, there are plenty of extension boards (also known as shields) that add functionality; we used an extension board called Egg Shield [10], it has sensors for carbon monoxide (CO), nitrogen dioxide (NO2), temperature and humidity.

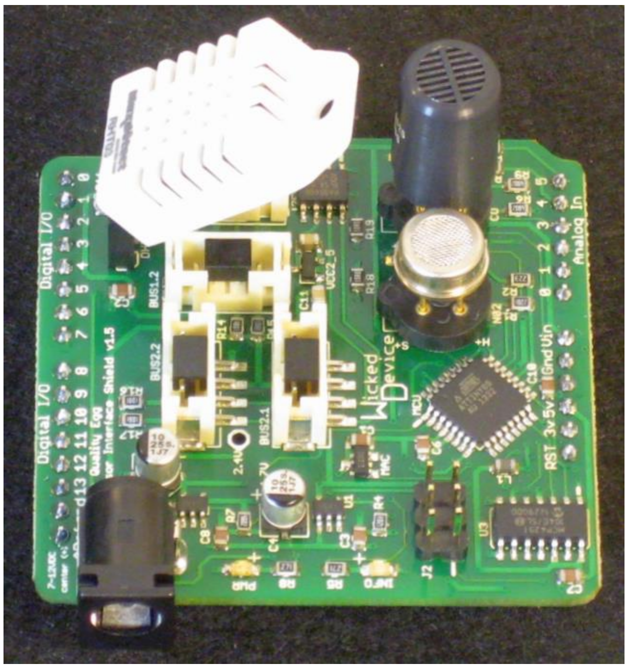


Figure 14: The Egg Shield board

It's also important to note that the carbon monoxide and nitrogen dioxide sensors have a settling time of about two minutes; measurements before they have settled are inaccurate.

The Arduino board also has a standard I2C bus for compatible sensors. This bus can be used to add other sensors and add functionality to the system.

Using this bus, we also added a barometric pressure sensor (MPL3115A2 [11]) to get altitude readings and allow vertical movement in the navigation algorithm.

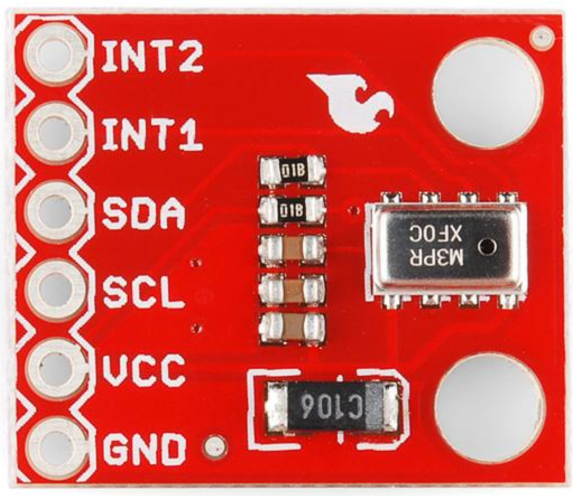


Figure 15: The MPL3115A2 sensor in a breakout board

Note that this sensor must be tightly covered with foam to offer proper readings; even then it may not be very precise. Our sensor had a small drift that resulted in a varying offset from the real value. It was still usable with temporally close relative measurements; in other words, it could still be used for things like increasing altitude by roughly 10 meters, not for keeping a record of flight altitude.

2.8: Android software. *QuadADK*

The Android program had to manage smartphone components (GPS and camera), handle the Arduino communication link and host the navigation algorithm. All these while being easy to modify, as the participants would have to develop their own navigation algorithms within the program.

This is why we went for a modular approach; each function is encapsulated in a module and they can all intercommunicate. See *3.2.2.2: QuadADK* and *3.2.3: Modules* for more information.

With this system, the navigation algorithm is encapsulated in the mission module and can be easily modified without knowledge of how the rest of the program works.

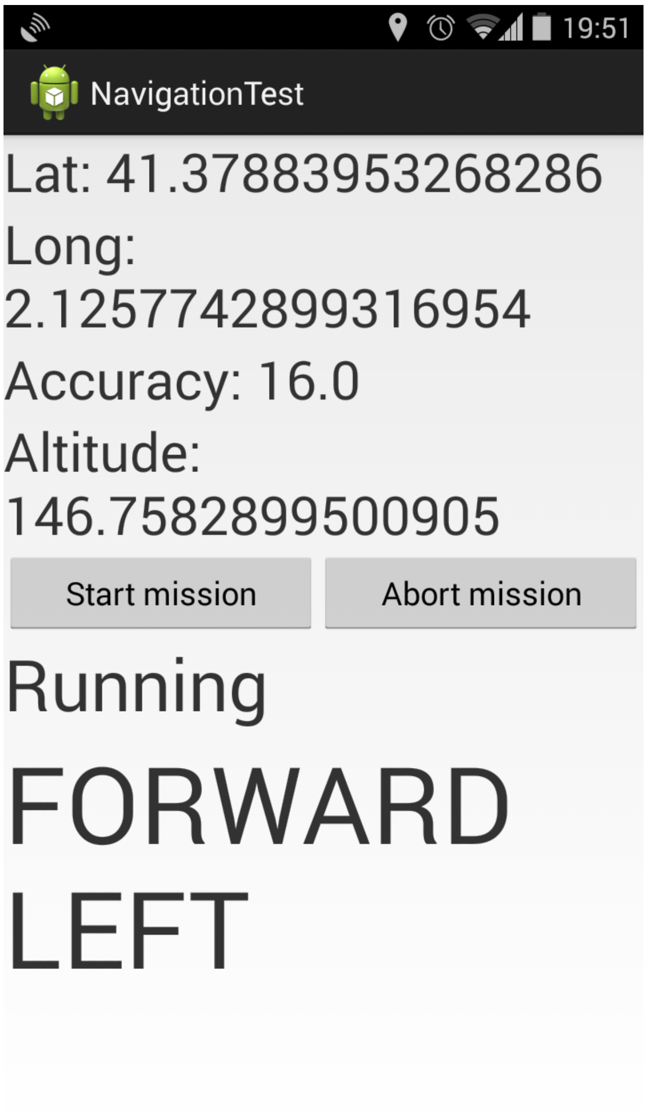
2.8.1: Flight mission

The flight mission is what the participants have to develop, it contains the navigation algorithm and instructions on how and when to make sensor measurements and photos.

Developing this algorithm is not especially complicated; having to test it with flights however, is at least time consuming and potentially dangerous when the algorithm malfunctions; we still have the manual override to tackle this situations.

To ease development, both for us and the participants, we developed another program that works almost identically but is designed to be used walking instead of flying. It shows the movement it wants to perform on the screen and the user walks in that direction.

Using this program, testing and troubleshooting was easy and safe.



Lat: 41.37883953268286
Long: 2.1257742899316954
Accuracy: 16.0
Altitude: 146.7582899500905

Running

FORWARD
LEFT

Figure 16: The NavigationTest program

With the help of this tool, we tweaked the GPS error tolerances to increase location precision and navigation performance.

Once the algorithm was good enough, we proceeded to play with the movement speed of the unit. Going faster means completing missions in less time (good) but going too fast will miss waypoints (bad) and give less time to react in case of emergency (worse).

We ended up with a good balance between speed, accuracy and safety.

2.9: System integration

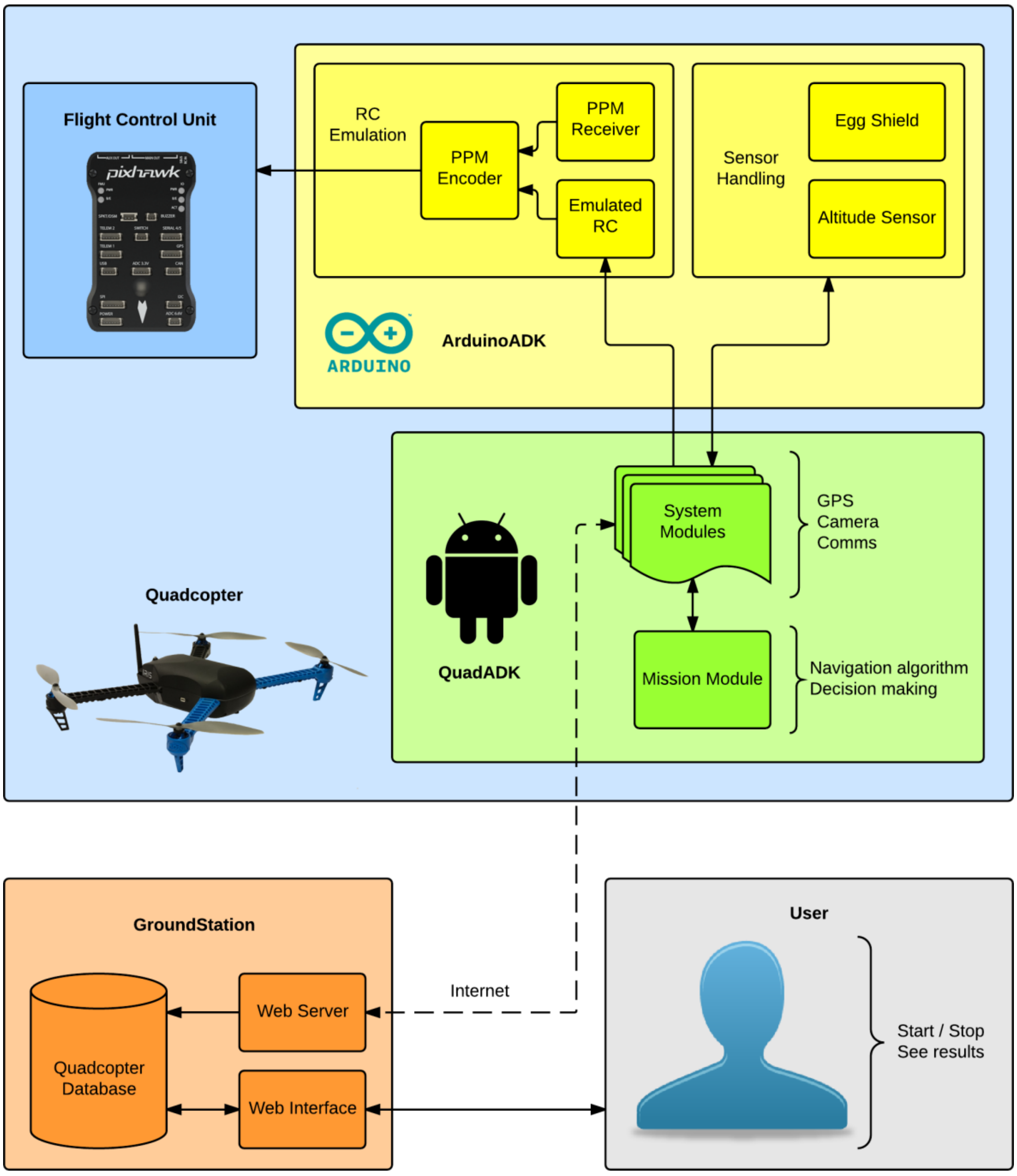


Figure 17: System overview

The core component is the mission module of the *QuadADK* program; everything is controlled from there, through the system modules. It is also the only component that needs to be modified by the participants. For this reason, all the complexity is hidden behind helper functions that do all the work; for instance, to take a picture you only have to call a helper function (one line of code) that does all this work behind the scenes:

- Deal with the camera module to take a picture
- Get the resulting picture file
- Save it in internal memory for redundancy
- Send it to the *GroundStation* through the Android – GroundStation communications module

This setup ensures that people with very basic programming skills (and even without experience in android programming) will be able to work with the platform. To add functionality however, you do need a better understanding of Android development.

Going back to the system overview, the smartphone is the brain and gives simple orders to the Arduino board. These orders are either set channel N of the virtual RC to value X or perform a measurement of the NO2 sensor. Arduino takes care of generating the appropriate PPM stream to control the quadcopter and handles the sensors.

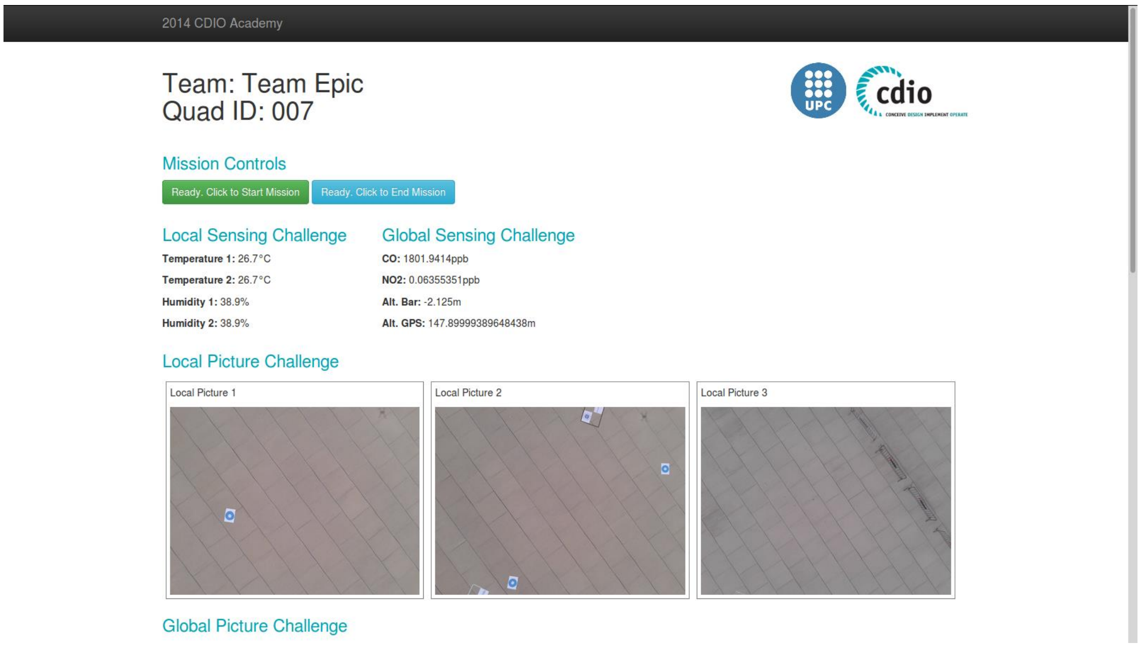
The Flight Control Unit of the quadcopter, which has not been modified in any way, just sees a regular RC. Takeoff and landing use its automatic modes instead of trying to do it with the smartphone, as it is much easier and safer.

The final part of the system, the *GroundStation*, has these components:

- Web server that receives all the data from the quadcopter(s)
- Web interface to show results, photos and sensor data, and start / stop the mission

The *GroundStation* can work with multiple quadcopters at the same time (although it's advised to only have one in the air at a time) as each one has a unique identifier. It stores data from all the quads in a MySQL database.

Also, while everyone can see the results of any quad, giving start and stop commands is password protected.



Team: Team Epic
Quad ID: 007

Mission Controls

Local Sensing Challenge

Temperature 1: 26.7°C
Temperature 2: 26.7°C
Humidity 1: 38.9%
Humidity 2: 38.9%

Global Sensing Challenge

CO: 1801.9414ppb
NO2: 0.06355351ppb
Alt. Bar: -2.125m
Alt. GPS: 147.89999389648438m

Local Picture Challenge

Global Picture Challenge

Figure 18: The GroundStation web interface

2.10: Final tests and validation

To verify the last prototype after all the design process, we ran a series of test to thoroughly check all the systems.

Safety is the most important aspect and thus most of the tests where focused on the Arduino program; we tested every scenario we could imagine (faulty or missing sensors, faulty or malicious Android code that tries to saturate the Arduino and a fully overloaded Arduino). The manual override was always available and ready no matter what.

Next, we ran different missions with different waypoints to test our proposed navigation algorithm. Again, no problems.

With these runs we also tested the *GroundStation*, storing and displaying data; different quad IDs and different passwords.

With only minor tweaks here and there, mostly the web interface layout, the validation was successful. The design process was finished.

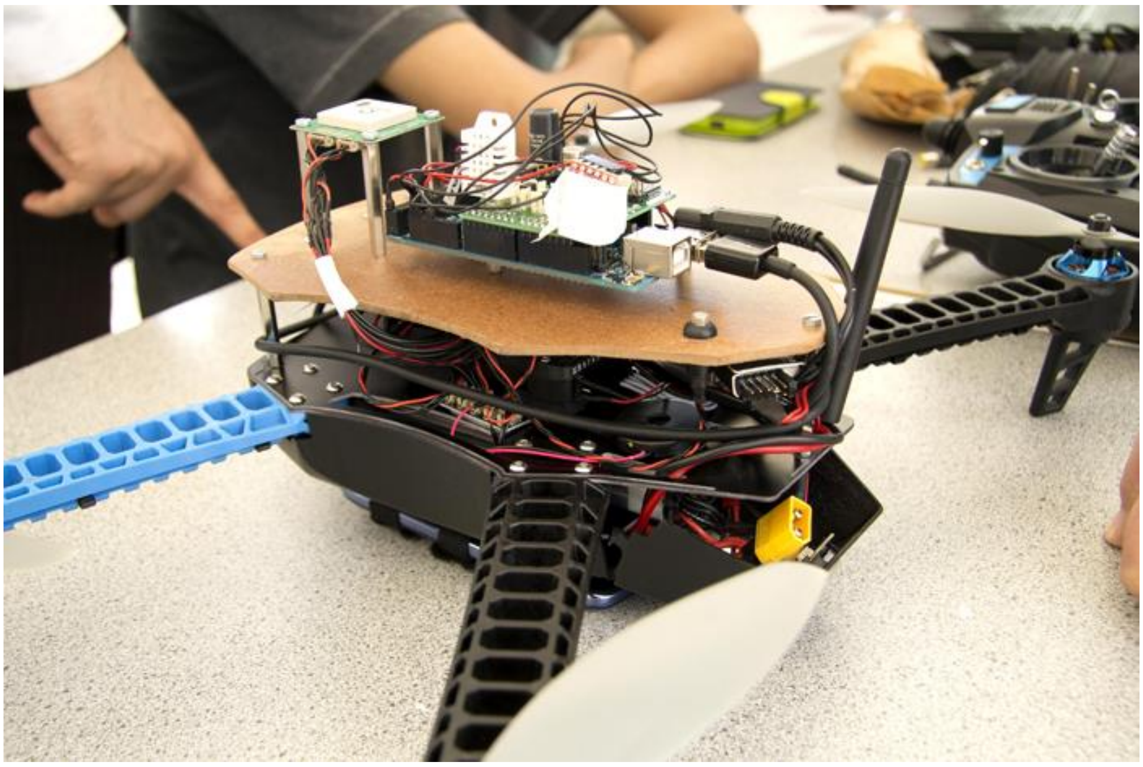


Figure 19: Finished unit

3: Project platform

This section is a reference guide to work with the platform.

3.1: Hardware

3.1.1: Quadcopter

This platform uses the 3DR Iris quadcopter, a commercial and ready to fly unit. It offers stabilized flight modes and automatic modes for taking off and landing.

Flying this quad is as easy as forward, backward, left and right horizontal movement plus vertical movement. Any non-experienced pilot should be able to fly it confidently with just a day of test flight and training.

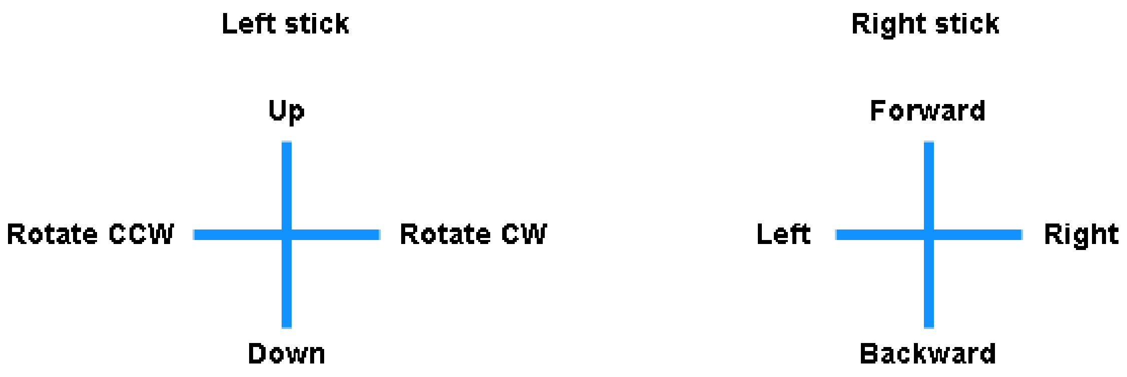


Figure 20: RC controls

Refer to the Operation Manual [12] for more information.

3.1.2: Added components

3.1.2.1: Android smartphone

The smartphone is the brain of the unit. It runs the program *QuadADK* and follows the preconfigured mission.

It has to be an Android smartphone for the wired connection between Arduino and smartphone; it is based on the Google's Accessory Development Kit [8].

Aside from OS, the only requirements are:

- GPS module
- Internet connectivity (cellular data or Wi-Fi)
- Camera

Most, if not all, smartphones these days fulfill these requirements.

The smartphone is installed under the quadcopter and held in place with two Velcro ties. It sits very close to the ground so, if using other smartphones, be careful not to scratch the camera (or add a protector around it so it doesn't come in contact with the ground).

If the smartphone doesn't fit under the unit, you can use the extended legs (included) to add 16 cm of clearance. This however, elevates the center of mass and decreases stability, especially when landing; only use them if strictly necessary.

Figure 21: Extended legs

3.1.2.2: Arduino MEGA ADK

The Arduino board hosts all the sensors and controls the quadcopter though an RC PPM signal. It runs the program *ArduinoADK*.

We use the MEGA ADK model because it comes with an onboard USB host interface, needed for the ADK communication, and provides plenty of processing power.

The board is powered from the quad's LiPo battery with a power lead soldered to the power distribution board (inside the black chassis). It also powers the smartphone through the USB connection so the whole system runs off the quad's battery. The impact on flight time is minimal as the motors use much more power than the rest of the components.

3.1.2.2.1: Body modifications

The Arduino board is installed on top of the quadcopter. The top cover is removed and an extra support panel is added; this panel holds the Arduino board with all its sensors, the quadcopter safety button (to arm motors), the RC receiver and the GPS module.

The GPS module of the quadcopter, originally located next to the Pixhawk Flight Control Unit, has to be moved on top of the added panel and elevated to have a good reception.

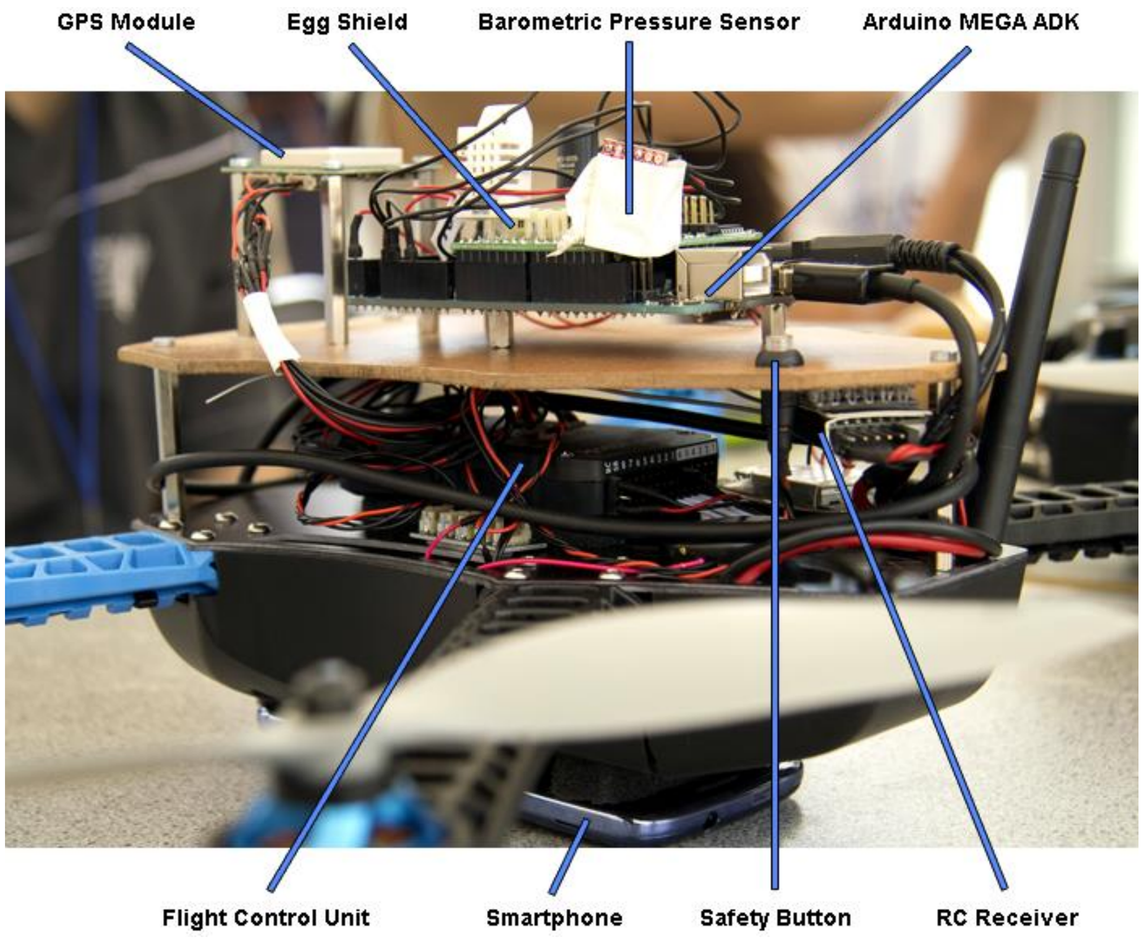


Figure 22: Components and their location

3.1.2.3: Sensors

The included sensors are the following:

- Egg Shield [10] extension board; carbon monoxide (CO), nitrogen dioxide (NO2), temperature, humidity
- MPL3115A2 [11] barometric pressure (altitude)

Egg Shield has all the required sensors for air quality sensing, mainly CO and NO2, while the barometric pressure sensor is useful for vertical navigation; keep in mind that the altitude reported has a changing drift over time so it is only useful for temporally close relative measurements, like going up roughly 10 meters.

More sensors can be added through the standard I2C bus. Software modifications, for both *ArduinoADK* and *QuadADK*, will be needed to use them, however.

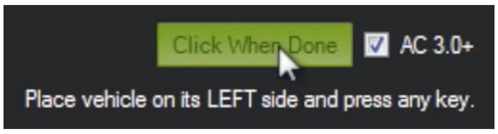
3.1.3: Calibrations

These are the calibrations required to operate the unit.

3.1.3.1: Quadcopter

All these calibrations are performed with the Mission Planner [13] [14] software, which has wizards for each. Refer to the manual [15] to see how to connect to the quadcopter and the YouTube guides [16] [17] [18] for calibrations.

Whenever there is a physical change in the quadcopter (adding, removing or rearranging components) the accelerometer sensor has to be recalibrated. The wizard asks you to put the unit on different orientations and only takes a minute.



Place vehicle on its LEFT side and press any key.

Figure 23: Accelerometer calibration wizard

If the GPS module is moved in any way, the compass also has to be calibrated (it is located on the GPS module).

This wizards takes more time, about 5 minutes, and it makes you constantly rotate the unit on its three axis. Note that the compass calibration wizard shown in the YouTube guide [17] is outdated, this is the current version:

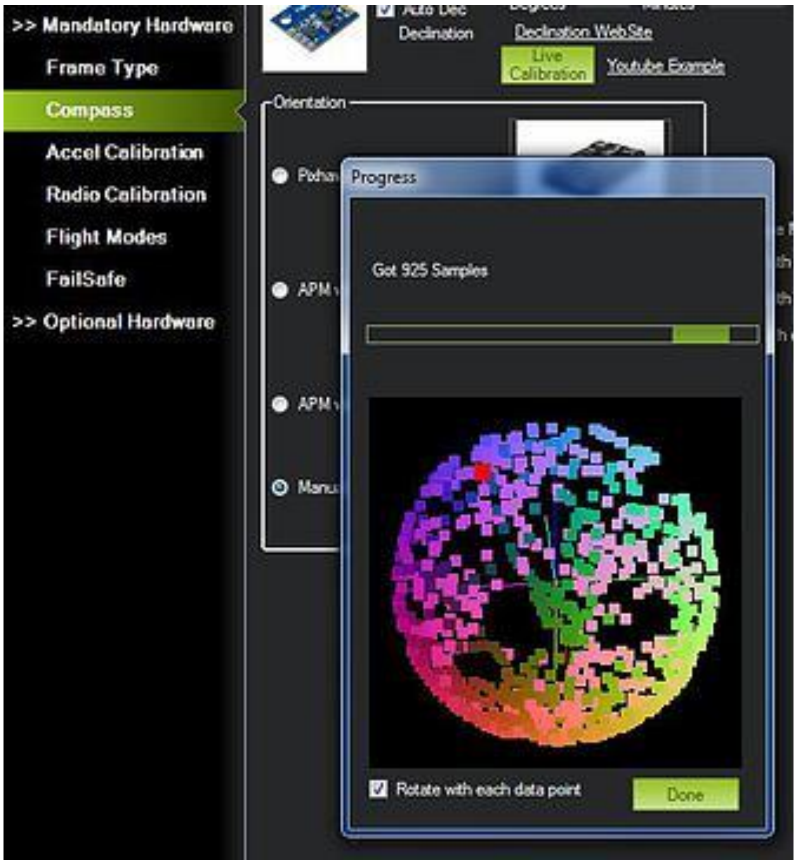


Figure 24: Compass calibration wizard

The RC also needs to be calibrated if you do any change to the values used in the program or change the RC for another one; the *ArduinoADK* also needs to know the new value ranges (minimum, maximum and neutral).

The wizard asks you to move all the analog sticks within full range and toggle all the switches.

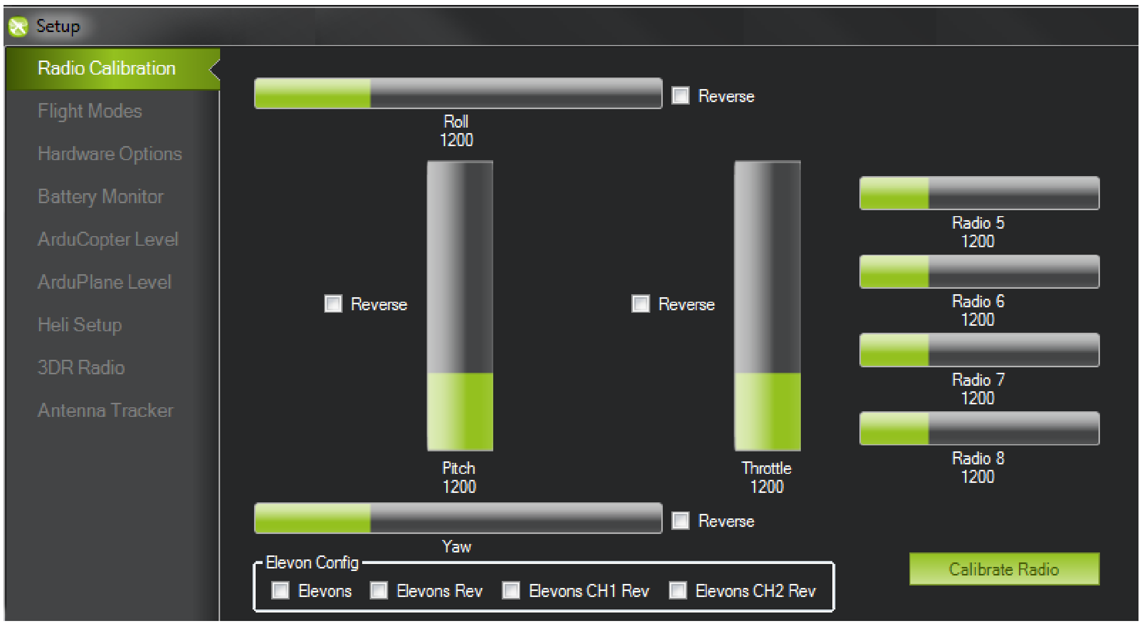


Figure 25: RC calibration wizard

3.1.3.2: Others

The Egg Shield air quality sensors also need to be calibrated using the tools included in the *EggBus* library. See [19] for more information. This calibration only needs to be done once.

3.2: Software

This section contains instructions on how to setup the building environment and explanations on how the software works.

3.2.1: Environment setup

3.2.1.1: Source code

The source for this project is hosted in a GitHub repository [5].
You can either clone the repository or just download the source code [20].
Using a version control solution such as Git to develop is highly recommended.

3.2.1.2: Android SDK

Android programs are, mainly, written in Java and it is recommended to use Eclipse IDE with the ADT plugin.

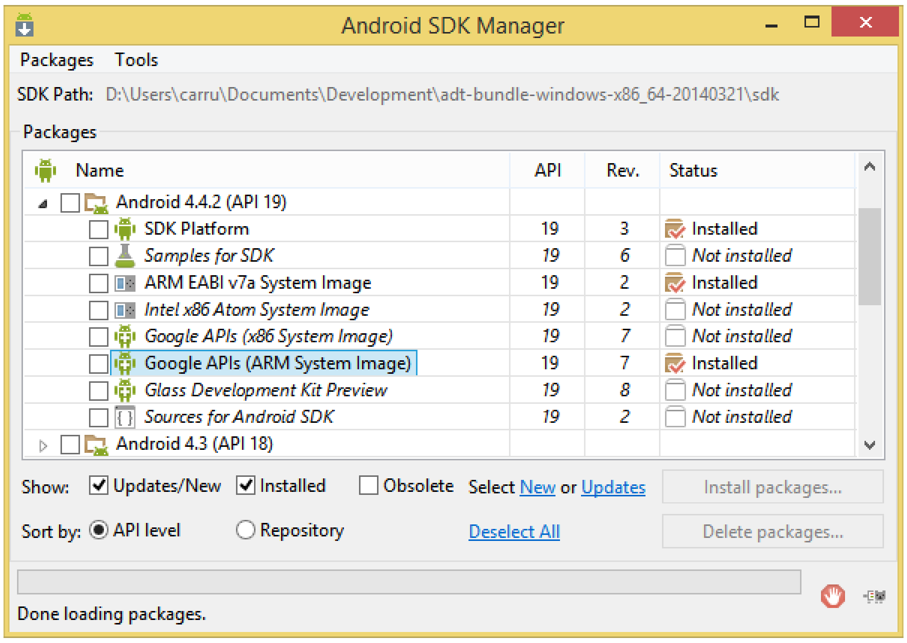
Google provides a ready to use stand-alone package with all the required tools (Eclipse, ADT and the Android SDK) called ADT Bundle [21].

Note about Android Studio:
Google is working on this new IDE called Android Studio with new features and improvements over Eclipse. However, at the moment of this writing it is still in beta and thus not recommend.

3.2.1.2.1: Google APIs

The source uses additional APIs not found in the Android SDK; the project has to be compiled against the Google APIs System Image.

First download the Google APIs System Image through the SDK Manager and then change the build target in the project properties.

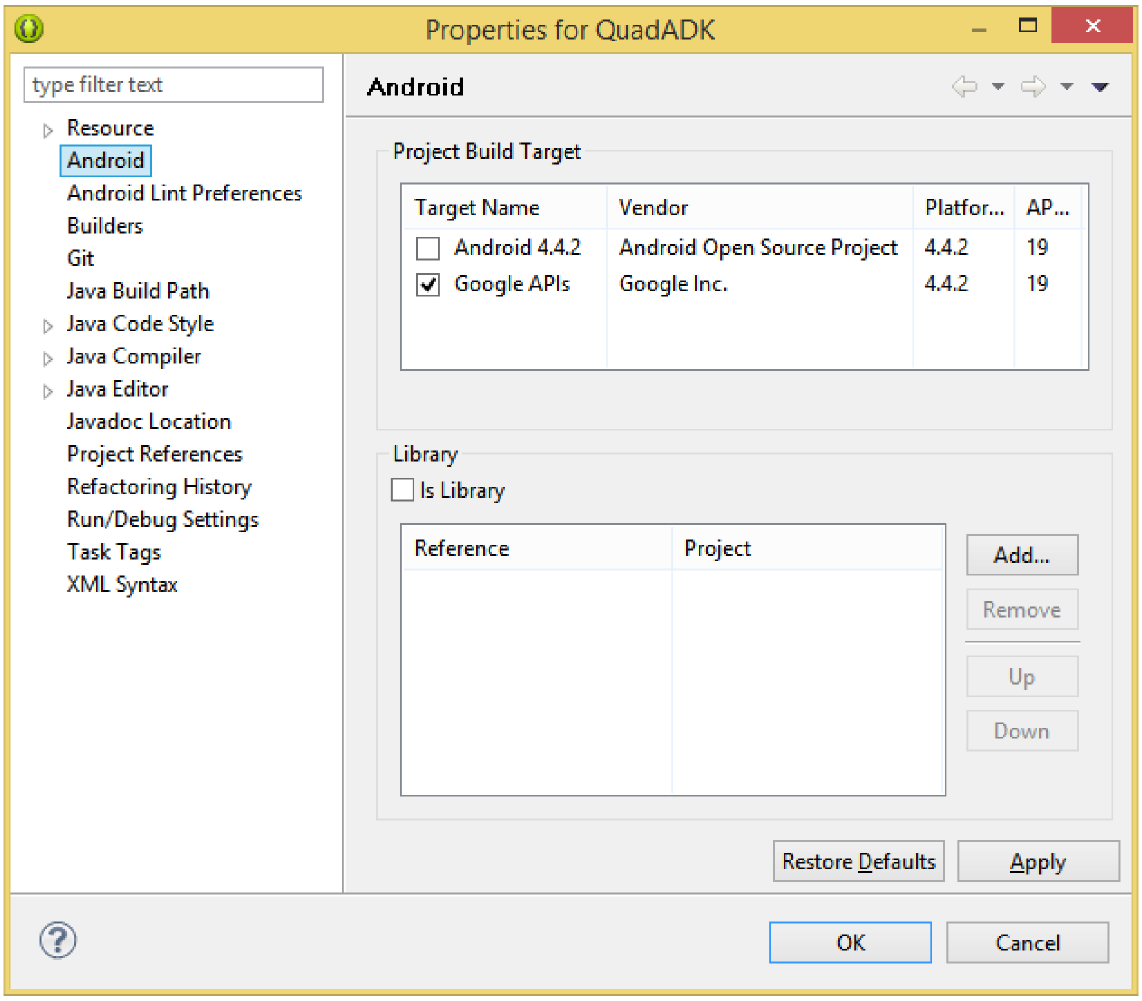


SDK Path: D:\Users\carru\Documents\Development\adt-bundle-windows-x86_64-20140321\sdk

Name	API	Rev.	Status
Android 4.4.2 (API 19)			
SDK Platform	19	3	Installed
Samples for SDK	*19*	*6*	*Not installed*
ARM EABI v7a System Image	19	2	Installed
Intel x86 Atom System Image	*19*	*2*	*Not installed*
Google APIs (x86 System Image)	*19*	*7*	*Not installed*
Google APIs (ARM System Image)	19	7	Installed
Glass Development Kit Preview	*19*	*8*	*Not installed*
Sources for Android SDK	*19*	*2*	*Not installed*
Android 4.3 (API 18)			

Done loading packages.

Figure 26: Android SDK Manager



Android

Project Build Target

	Target Name	Vendor	Platfor...	AP...
☐	Android 4.4.2	Android Open Source Project	4.4.2	19
☑	Google APIs	Google Inc.	4.4.2	19

Library

☐ Is Library

Reference	Project

Figure 27: Project properties, build target

3.2.1.2.2: Required libraries

These are the additional libraries required:

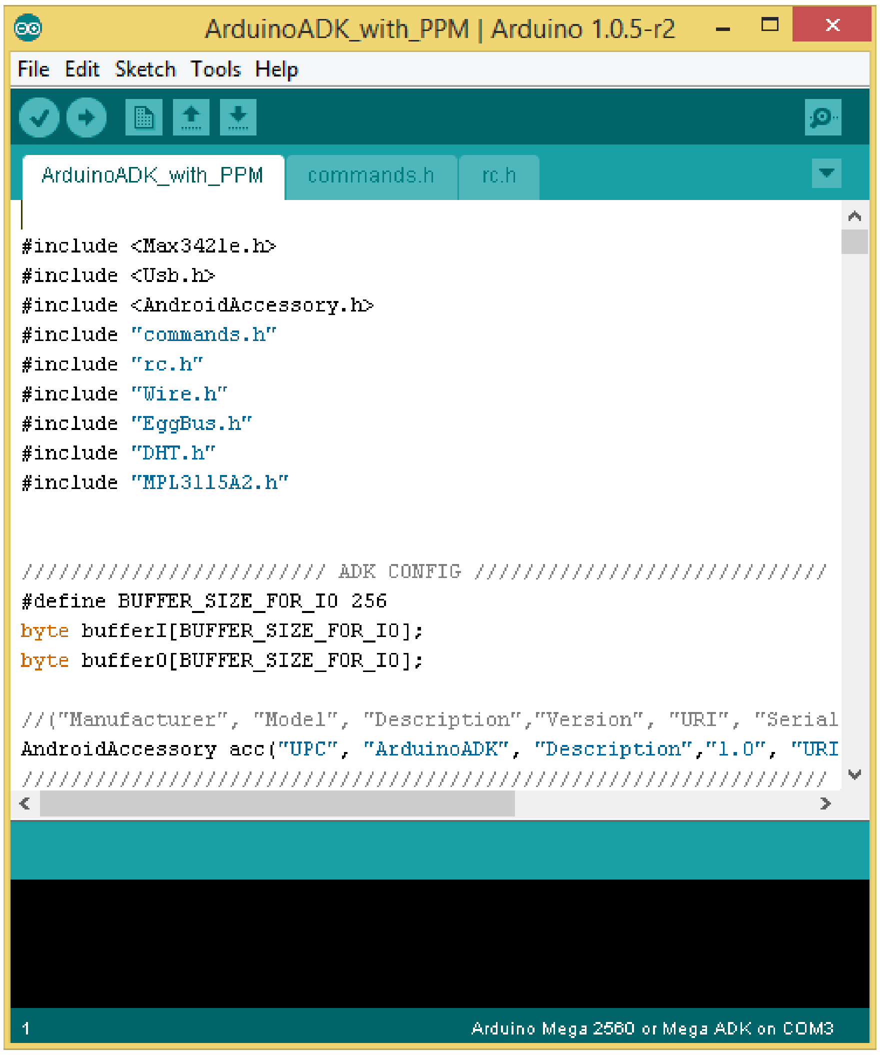
- ADK, for Android-Arduino communication
- GSON and HTTP, for Android - GroundStation communication

They are already included in the source (Quad / QuadADK / libs). If you require additional libraries for your application, place them in this folder.

3.2.1.3: Arduino IDE

Arduino is programmed in C/C++ and using the Arduino IDE [22].

Since it is a standard AVR development board, you can use other compilers and tools but the official IDE is recommended as it provides easy building and flashing.


ArduinoADK_with_PPM | Arduino 1.0.5-r2
File Edit Sketch Tools Help
ArduinoADK_with_PPM
commands.h
rc.h
1
Arduino Mega 2560 or Mega ADK on COM3

```
#include <Max3421e.h>
#include <Usb.h>
#include <AndroidAccessory.h>
#include "commands.h"
#include "rc.h"
#include "Wire.h"
#include "EggBus.h"
#include "DHT.h"
#include "MPL3115A2.h"


////////////////////////// ADK CONFIG //////////////////////////////
#define BUFFER_SIZE_FOR_IO 256
byte bufferI[BUFFER_SIZE_FOR_IO];
byte bufferO[BUFFER_SIZE_FOR_IO];

//("Manufacturer", "Model", "Description","Version", "URI", "Serial
AndroidAccessory acc("UPC", "ArduinoADK", "Description","1.0", "URI
////////////////////////////////////////////////////////////////////
```

Figure 28: Arduino IDE

Don't forget to set your type of Arduino board (Tools > Board > Arduino Mega 2560 or Mega ADK).

3.2.1.3.1: Required libraries

These are the additional libraries required:

- Android Accessory and USB Host Shield for the ADK Android – Arduino link
- DHT and EggBus for the Egg Shield board
- MPL3115A2 for the barometric pressure sensor

They are already included in the source (Quad / Arduino / libs). If you require additional libraries, place them in this folder.

3.2.2: Software walkthrough

3.2.2.1: ArduinoADK

The Arduino board has two main functions: emulate an RC device for the quadcopter and manage the sensors, the former having priority to ensure a proper emulation.
All the source is written in C.

The header file *rc.h* contains constants for the values of different flight mode (channel 5 of the RC) as well as constants for the different channels of the RC.

The header file *commands.h* contains all the constants of commands for Arduino – Android communication. These values have to be the same as those on the *ArduinoCommands* interface of the *QuadADK* source (*es.upc.lewis.quadadk.comms.ArduinoCommands*).

The ADK information (manufacturer, model and version) used in the declaration of the *AndroidAccessory* object also has to match with the information used in the *QuadADK* source *(/QuadADK/res/xml/usb_accessory_filter.xml*).

The source of the program is in the *ArduinoADK_with_PPM.ino* file, a special filetype that includes project (also known as sketch) properties and source.

At the top there are constants to configure different parts of the program: RC channels and their range, PPM stream specifications and input/output pins.

On startup, the setup() function is called and does the following tasks:

- Initialize serial communication for debugging.
- Initialize and configure sensors, Egg Shield and altimeter.
- Open the ADK link for incoming connections.
- Configure the input and output pins and internal timers for PPM input and output.

Next, it enters the main loop, *loop()*, where it waits for commands from the Android device. These commands (see *3.2.3.1.1: Arduino - Android*) include movement orders (changes in the emulated RC) and sensor readings. When a command is read, it extracts the data, if any, and performs the order with a big switch/case structure (*attendCommand(byte command, int value)* function).

There are two interrupt handlers: *intHandler()* and *ISR(TIMER1_COMPA_vect)* which handle PPM reading and generating respectively.

These are the other functions and their purpose:

sendSensorData(byte sensor, float value)	Send a sensor reading to *QuadADK*
setPPMChannel(int channel, int value)	Change the value of a given channel
setFlightMode(int mode)	Change to the specified flight mode
setSlidersNeutral()	Set all sticks to their neutral position
setSlidersNeutralNoThrottle()	Set all sticks to their neutral position and throttle at minimum

3.2.2.1.1: RC emulation

RC emulation works by sending to the quadcopter a PPM stream (see appendix *1: Pulse Position Modulation*) containing the different channels of the RC, eight in this case.

For safety, the program can override autonomous control with a real RC. To accomplish this, it maintains two snapshots (RC states, values of all the channels) in arrays of 16 bit integers, one per channel:

ppm[]	Virtual RC, *QuadADK* sends updates to this snapshot
ppm_in[]	Manual, updated automatically with an interrupt

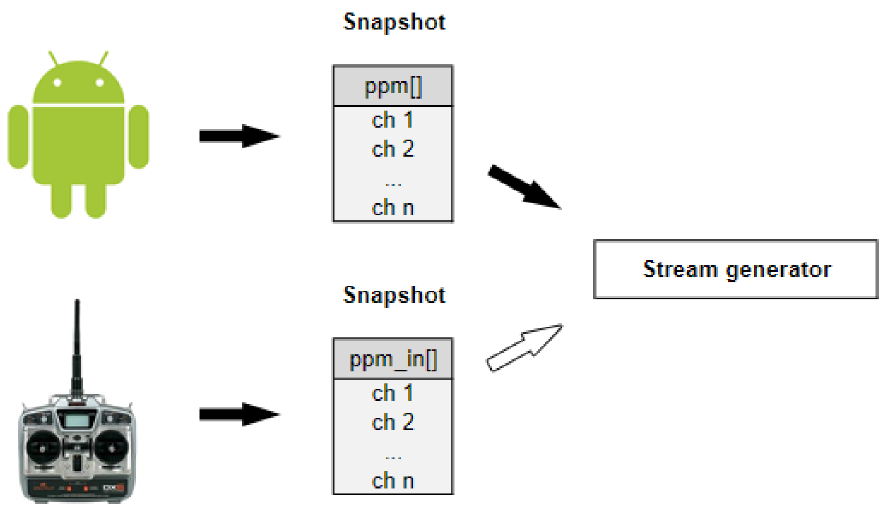


Figure 29: RC emulation in the source code

The generator then picks one or the other depending on the state of the manual override switch (channel 7), see *2.4.1: Manual override* for more information. The signal is generated using a timer.

Since manual RC pass-through is done with interrupts (timers trigger as interrupts) in both acquisition and generation, it has priority over the rest of the program; this gives the added security of always available manual mode even if the program misbehaves or the Arduino is overloaded.

PPM reading is done with an interrupt that triggers on every rising edge of the signal; the interrupt handler calculates time between interruptions in microseconds, this time is the value of the current channel, and then saves this values in the *ppm_in[]* array sequentially. It syncs with the receiver using the blank time of the stream, which is longer than the maximum value of a channel.

PPM generation uses a timer and its interrupt to generate the signal channel by channel. When generating it reads from *ppm[]* or *ppm_in[]* (virtual and real RC respectively) depending on the mode, autonomous or manual.

3.2.2.1.2: Sensors

Sensors include those on the Egg Shield (nitrogen dioxide, carbon monoxide, humidity and temperature) and the barometric pressure sensor. To work with them, use their respective libraries.

For the Egg Shield, temperature and humidity are accessed with the *DHT* object while CO and NO2 use *EggBus*. The altitude sensor uses the *MPL3115A2* object.

Note that nitrogen dioxide and carbon monoxide sensors have to be accessed in a slightly different way because of how they are implemented in the Egg Shield.

To add additional sensors it is recommended to use the I2C bus of the Arduino.

3.2.2.2: QuadADK

The Android application, *QuadADK*, is designed to automatically start when the Arduino board is plugged in and its main, and only, activity (user interface) shows ADK connection and GPS status.

The interface is not pretty as most of the time the smartphone will be in the air. It also has a couple buttons used to debug and troubleshoot issues (trigger an action manually, like sending sensor data to the web server).

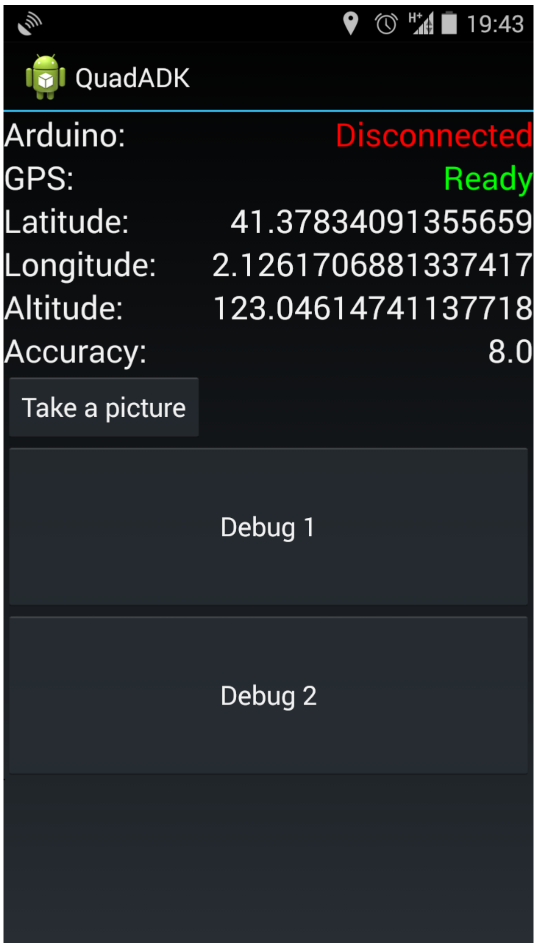

19:43
QuadADK
Arduino: Disconnected
GPS: Ready
Latitude: 41.37834091355659
Longitude: 2.1261706881337417
Altitude: 123.04614741137718
Accuracy: 8.0

Figure 30: QuadADK interface

This application is designed in a modular way so that different components are encapsulated and is easier to work with them or add new ones for more functionality. Further information of these modules is available in *3.2.3: Modules*.

All the modules can talk to each other with application wide broadcasts; each object declares which broadcasts wants to listen to. For instance, the module that handles the *GroundStation* link sends broadcasts notifying commands like start and stop mission; the module that handles the ADK link sends broadcasts with the result of a sensor read command issued previously. This makes it easy to add new functionality as it is very easy to talk to other modules.

The Activity is launched when the smartphone is plugged to the Arduino. This is achieved by registering the Arduino as an accessory of this application. The first time though, the Android system may ask for confirmation to launch the application.

Also note that, due to limitations of the ADK platform, the application has to be restarted if you disconnect the Arduino board.

On startup (*onCreate()* method) all the different modules are initialized and then it waits for the *GroundStation* to signal the start of the mission.

When stopping the application, the Activity stops and closes all the modules (required to free system resources, like the camera).

3.2.3: Modules

3.2.3.1: Communications

3.2.3.1.1: Arduino – Android

Because the processing power of the Arduino board is limited, communications between *QuadADK* and *ArduinoADK* use a very simple and lightweight custom protocol; see appendix *2: Arduino – Android communication protocol* for more information.

These messages are used to perform sensor readings, retrieve the results and change the emulated RC snapshot.

AndroidADK reads pending messages at each iteration of the main loop and decides the action to perform in a big switch structure.

QuadADK has a thread called *CommunicationsThread* that handles this link; it reads incoming messages and broadcasts them to the other modules, it also has public methods to let other modules send messages easily.

The *ArduinoCommands* interface of *QuadADK* and *commands.h* of *ArduinoADK* contain all the Command IDs.

To add new commands you have to add their IDs to the previous two files and add the code to manage them for both *ArduinoADK* and *QuadADK*.

3.2.3.1.2: Android – *GroundStation*

This communication link is handled with standard HTTP protocol. The *GroundStation* is a PHP web server and *QuadADK* reads and sends data using *HttpGet*.

The web server has a MySQL database with sensor data fields and mission start/stop flags for each quadcopter; photos are uploaded to the server.

QuadADK has these threads that take care of different parts of this communication link:

- *MissionStatusPolling* continually polls the server for changes in the mission start/stop flags and notifies them to other modules.
- *SendDataThread* sends sensor data to the specified database field.
- *SendPictureThread* uploads pictures to the server with the specified filename.

The class *HTTPCalls* contains low level implementations of the functions used in the previous threads. This approach makes the source code easier to read and understand.

Performing networking tasks (or other I/O tasks) cannot be done in the same thread of the Activity. That is why these tasks are encapsulated in separate threads.

3.2.3.2: Sensors

3.2.3.2.1: Camera

The *SimpleCamera* class encapsulates all the code required to manage and operate the camera. It offers a public method *takePicture(String)* that takes a picture, saves it with the name given as a parameter and sends it to the *GroundStation*, using the classes discussed in *3.2.3.1.2: Android - GroundStation*.

Note that, due to Android limitations, there has to be a camera preview view in the Activity to be able to take pictures. The *CameraPreview* class handles this view and, as a workaround, it has a size of 1 by 1 DPs (minimum size).

The most significant method of this class is the *PictureCallback*; it is triggered after taking a photo and receives the binary data of the image. It logs information messages for troubleshooting, saves the picture to local storage and sends it to the *GroundStation*.

3.2.3.2.2: GPS

The *MyLocation* class handles the GPS of the device. It discards readings with a precision worse than a given threshold and broadcasts readings to other modules.

It can also send a warning if there are too many bad samples (poor precision) in a row in case it loses GPS connection; this however is commented out (see *gpsFailsafe()* method) because it is not implemented in the mission module.

Another unused feature is GPS height reporting. It was finally discarded because the precision is very low. See appendix *3: GPS height precision* for more information.

3.2.3.3: Mission

The *MissionThread* thread is the main part of the whole project. It defines the mission, or program, that the quadcopter will follow: how and when to move, make sensor measurements and take pictures.

The *MissionUtils* class contains helper methods to send commands to the Arduino and encapsulates complex tasks like arming motors, taking off and landing.

This comment block from the beginning of the file explains how to work with this module:

```
/**
 *
 * Implement your mission in this class (inside the run() method).
 *
 *
 * The MainActivity will start this thread when the server signals to start.
 * If the server aborts the mission you'll receive an ABORT_MISSION broadcast (see broadcastReceiver).
 * This BroadcastReceiver also receives data from the Arduino sensors; all it does now
 * is send it to the server.
 *
 *
 * Useful methods:
 *
 * locationProvider.getLastLocation()       Returns your GPS position (updated roughly once per second)
 *
 *
 * The class MissionUtils has this methods you should use:
 *
 * takeoff             Start the quadcopter and go to the starting position
 *
 * returnToLaunch      Return to the starting position and land
 *
 * abortMission        Interrupt and stop your mission and issue a returnToLaunch
 *                     Do not use it to end your mission when all the work is done
 *                     (see endMission for this purpose)
 *
 * endMission          Prepare to end this thread
 *                     You should finish your mission with returnToLaunch and endMission
 *
 * wait                Sleep for a given number of milliseconds
 *
 * readSensor          Example: utils.readSensor(MissionUtils.TEMPERATURE);
 *                     Tell the Arduino to measure temperature and receive the result in the broadcastReceiver
 *
 * takePicture         Take a picture with the back camera and send it to the server
 *                     String parameter is the name
 *
 *
 * Movement methods:
 *
 * hover               Stay in place, don't move in any direction
 *
 * To move your quadcopter horizontally you have to simulate movement of the right stick
 * The neutral (middle) position of the stick is 1500 and the range is [1000, 2000]
 * Channel 1: right (high) and left (low)
 * Channel 2: forward (low) and backward (high)
 *
 * Examples
 * Forward             utils.send(ArduinoCommands.SET_CH2, MissionUtils.CH_NEUTRAL - 200);
 *                     Moves forward
 *
 * Right               utils.send(ArduinoCommands.SET_CH1, MissionUtils.CH_NEUTRAL + 300);
 *                     Moves right faster
 *
 * Note how the channel value (second parameter) is an offset (neutral position) with
 * the movement added. With +/- 300 we get 1200 and 1800
 * We recommend values from +/- 200 to +/- 300
 *
 * You can combine them to move diagonally:
 * utils.send(ArduinoCommands.SET_CH2, MissionUtils.CH_NEUTRAL + 200);
 * utils.send(ArduinoCommands.SET_CH1, MissionUtils.CH_NEUTRAL + 200);
 * This will make the quadcopter move backward and right at the same time
 *
```

```
*
* Channel 3 is the throttle; leave it at neutral (MissionUtils.THROTTLE_NEUTRAL) to maintain
* altitude and use 'utils.goUp()' to increase it
* WARNING!: Never try to descend. Also, throttle channel has a different range than the others
* but you don't need to directly modify it.
*
* Use only these two methods to change throttle:
* utils.send(ArduinoCommands.SET_CH3, MissionUtils.THROTTLE_NEUTRAL)
* utils.goUp()
*
*
* Channel 4 controls yaw (rotation)
* The quadcopter will maintain the orientation it started with and we always start with it pointing north
* Because of this, you can assume the quadcopter is always pointing north and thus don't need to
* manually change yaw
*
* When pointing north, moving to the right increases longitude and moving forward increases latitude
*
*
* Channel 5 is the flight mode
* You don't have to worry about it. Your flight mode during the mission will be Loitter
* Don't change it
*
*
* Channel 7 controls who has control over the quadcopter, Android or the RC
* Don't change it
*
*
* Channels 6 and 8 are unused
* Don't change them
*
*/
```

QuadADK is designed to manage everything from the *MissionThread*, for instance: to make a sensor reading you call the helper method of *MissionUtils*, this will send the command through the communications module, perform the operation on the *ArduinoADK*, send the result back through the communications module and broadcast the data to receive it in *MissionThread*.

When the *MissionStatusPolling* thread signals the start mission flag, the mission starts with the *run()* method and it all runs inside a try-catch structure that catches the *AbortException* exception. This exception is thrown when the mission has been aborted, as signaled by the *MissionStatusPolling* thread. This lets you abort the mission (via the *GroundStation* web interface) mid-air and safely land; useful when your instructions are misbehaving.

It is also very important to set the quad ID in the *QUAD_ID* String and it must be unique. For the challenge, we used 001, 002, etc.

The source has a sample mission with a simple navigation algorithm: it follows a list of predefined waypoints (latitude, longitude), making measurements with all the sensors and taking a picture at each waypoint and sending it all to the *GroundStation*; on the last waypoint it goes up for a short time and takes a final picture before landing.

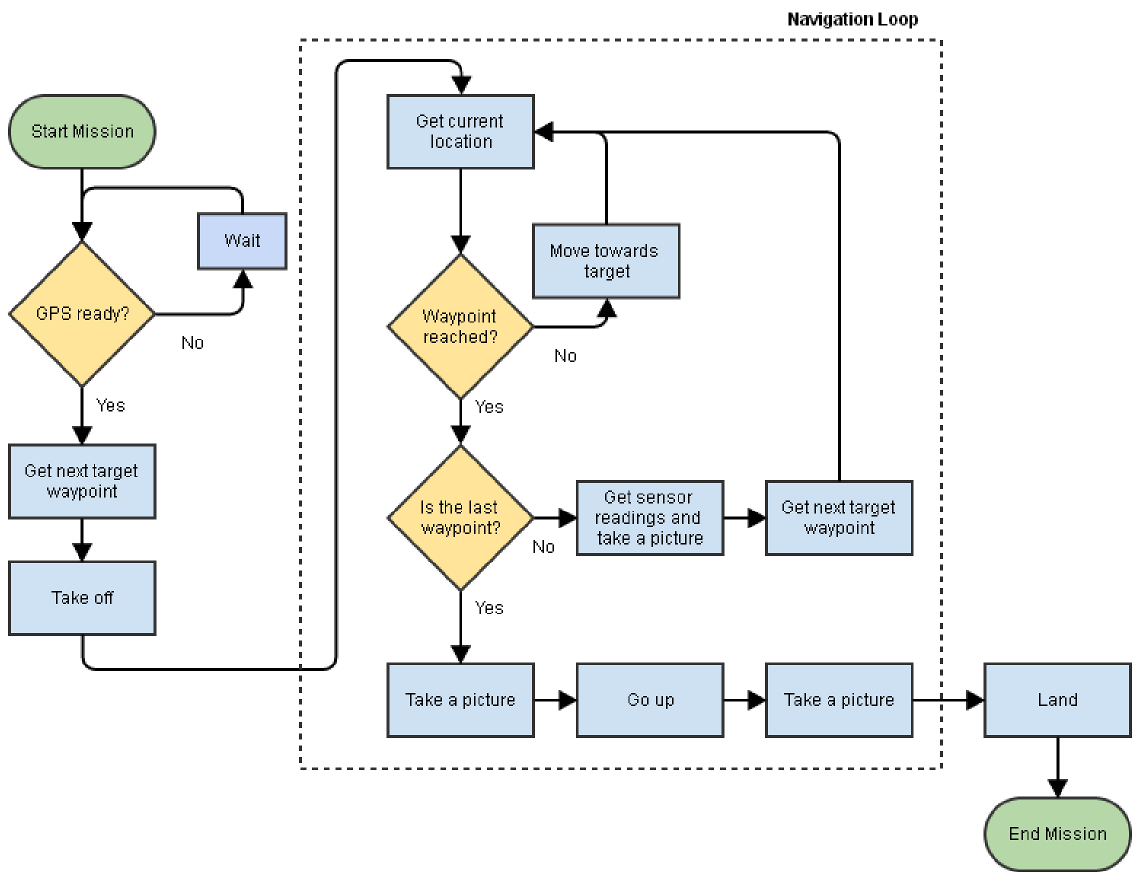


Figure 31: Sample mission flow diagram

This simple navigation algorithm is a loop with a given period (*NAVIGATION_LOOP_PERIOD*, 250 milliseconds by default); at each iteration it decides whether the target waypoint is reached or not based on a latitude and longitude difference, if the difference with the target is less than a given threshold (*WAYPOINT_LATITUDE_ERROR* and *WAYPOINT_LONGITUDE_ERROR*) it has been reached, otherwise it moves towards it.

3.2.4: Further development

There are many ways of expanding this project and add more functionality. These are some of the ideas we had and other things that can be done:

- Add more sensors. Increase functionality by adding more sensors to the Arduino board.
- Implement a camera assisted hover. Use the camera of the smartphone and image processing to improve hovering (standing still, also known as loiter).
- Improve GPS precision. Increase GPS accuracy with proper data processing, even if it means longer acquisition times and thus slower movements.
- Implement a geofence. Restrict flight zone so that the unit won't fly outside given boundaries.

4: Application

This capstone project was used as a challenge in the CDIO Academy 2014 within the CDIO framework.

4.1: CDIO

From their website [23]: “The CDIO™ INITIATIVE is an innovative educational framework for producing the next generation of engineers. The framework provides students with an education stressing engineering fundamentals set in the context of Conceiving — Designing — Implementing — Operating (CDIO) real-world systems and products. Throughout the world, CDIO Initiative collaborators have adopted CDIO as the framework of their curricular planning and outcome-based assessment. CDIO collaborators recognize that an engineering education is acquired over a long period and in a variety of institutions, and that educators in all parts of this spectrum can learn from practice elsewhere. The CDIO network therefore welcomes members in a diverse range of institutions ranging from research-led internationally acclaimed universities to local colleges dedicated to providing students with their initial grounding in engineering.”

cdio

CONCEIVE DESIGN IMPLEMENT OPERATE

Figure 32: CDIO logo

The 2014 annual conference [24] was held in Barcelona, at Universitat Politècnica de Catalunya, and discussed new education methodologies.

4.2: CDIO Academy

The CDIO Academy event runs parallel to the CDIO conference and hosts students from worldwide universities. It is divided in two parts [25]:

- Part 1: Teams of engineering students are invited to participate in the conference and to submit innovative design projects to the competition. Design projects should have been completed at their home institution. Projects will be presented in a poster format in a juried design project exhibit. The Contest for the CDIO Cup will be based on the presentation of these projects.
- Part 2: To take advantage of the multidisciplinary nature (within engineering) of the institutions involved in the CDIO initiative, students will be redistributed in multidisciplinary and international teams to work in a common project that will involve several technologies during the Conference days.

The second part is where this thesis comes into play. The capstone project described here was used as the challenge for the multidisciplinary international teams.

4.3: Challenge description

The challenge required participants to program the quadcopter to complete as many as possible of the following tasks, in any order:

- Local pictures. Take a picture of each of the three university logos scattered on the ground.
- Global picture. Take a picture that contains the four blue corner markers on the ground.
- Get the biggest gradient of temperature possible within any point of the mission.
- Get the biggest gradient of humidity possible within any point of the mission.
- Get a carbon monoxide (CO) and nitrogen dioxide (NO2) reading of the air. This measurement is compared with the reading of a properly calibrated and settled unit.

Other factors such as elapsed time, picture quality and energy consumption were also taken into account.

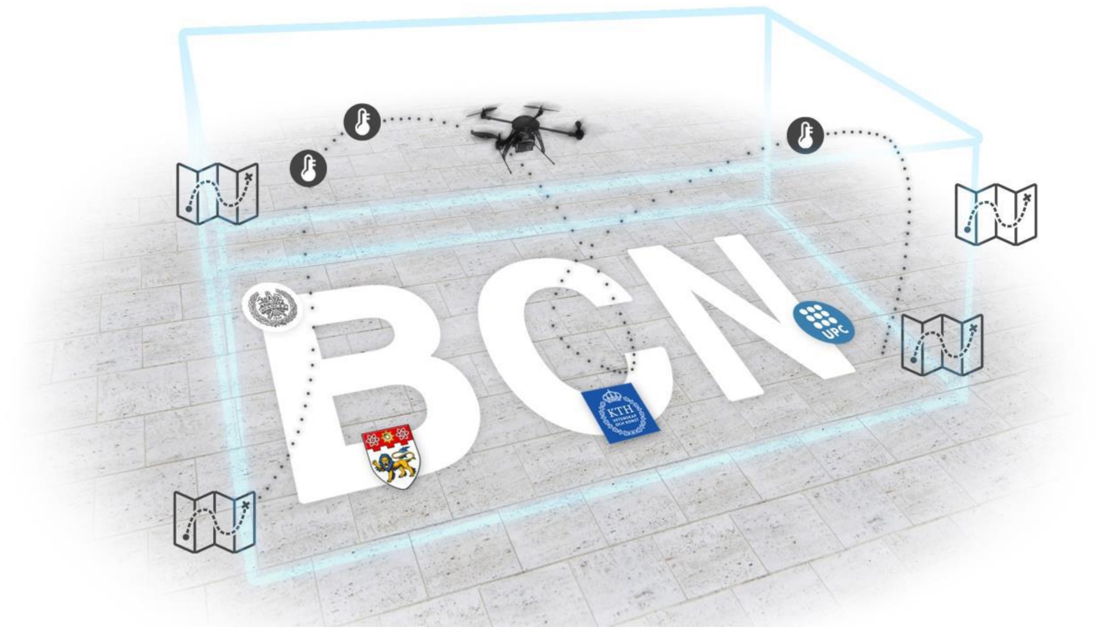


Figure 33: Challenge diagram

To properly achieve this tasks, each team had to solve these problems:

- For the global picture, the quadcopter needs to gain altitude to be able to get the four targets on a single picture. How higher? Find out the field of view of the smartphone camera and calculate the appropriate height.
- Picture quality matters; it's best to let the quadcopter stabilize before shooting.
- CO and NO2 sensors have to be calibrated and their settling time has to be determined.

Each team also had degrees of freedom to find the best approach:

- Reduce movement speed for greater precision at the expense of mission time and battery consumption (we did not allow increasing the speed for safety reasons).
- When to make sensor readings and how many, they had to upload six measurements in total (2x temperature, 2x humidity, CO, NO2) but could make as many as they saw fit.
- Define the path to follow or the order in which they would complete each subtask.
- Implement their navigation algorithm.

We introduced this degrees of freedom by going backwards in the design process and leaving choices open. Naturally this could only be done with certain aspects, as allowing modifications of critical components, like the Arduino program, would require many more validation steps and time; time we did not have.

Restricting their decisions guarantees that all the safety mechanisms will still work.

4.4: Comments and results

Given the limited time the participants had to work on the challenge (two mornings and one evening), we decided to give away the source code of the sample mission, a fully working example, so they did not have to develop it from scratch.

They also had the *NavigationTest* utility available to fine tune their navigation algorithms, and also save the limited flying time for more important tests.

Results were positive with two of the four teams completing a full mission. The major problem, as most of the participants acknowledged, was the very limited amount of time to work on the challenge.

We were also surprised by some creative solutions that differed from what we had in mind. For instance: since we had very intense sunlight during the challenge, the ground was very hot and one of the teams took a temperature reading before taking off, thus giving them a bigger temperature gradient than only inflight readings, as we have intended. All of the teams also realized they could pre-heat the sensors by powering the Arduino board before it was their turn to fly (they each had their own board), this greatly reduced the settling time of the sensors.

These ingenious approaches, while not following our idea of the challenge, were perfectly ‘legal’. Shows how thinking outside the box can yield better results.

Talking with participants after the event, feedback was very positive and, in their words, they enjoyed the challenge and learnt new things.

As a personal note, working and participating in this project has been a very enriching experience. I have learnt new technical things as well as organizing a big project and working with people of different disciplines from around the world.

Some photos of the event:

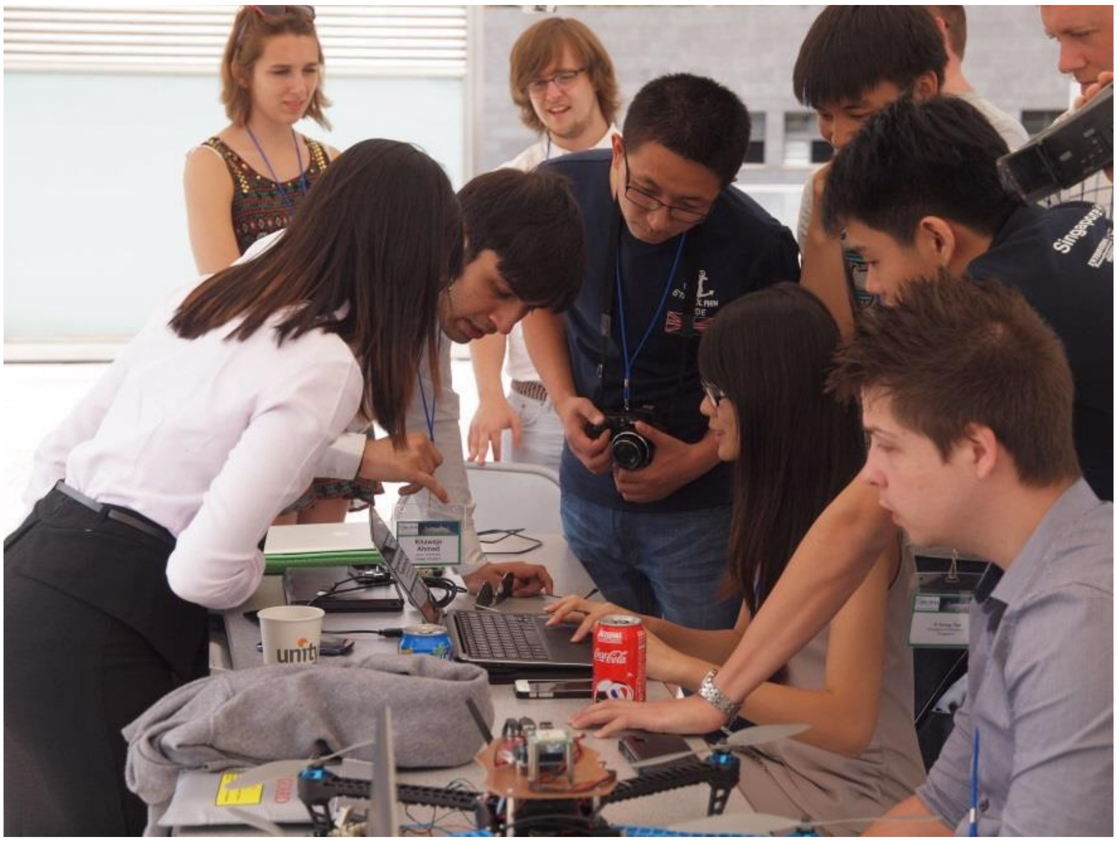


Figure 34: Participants working (1)

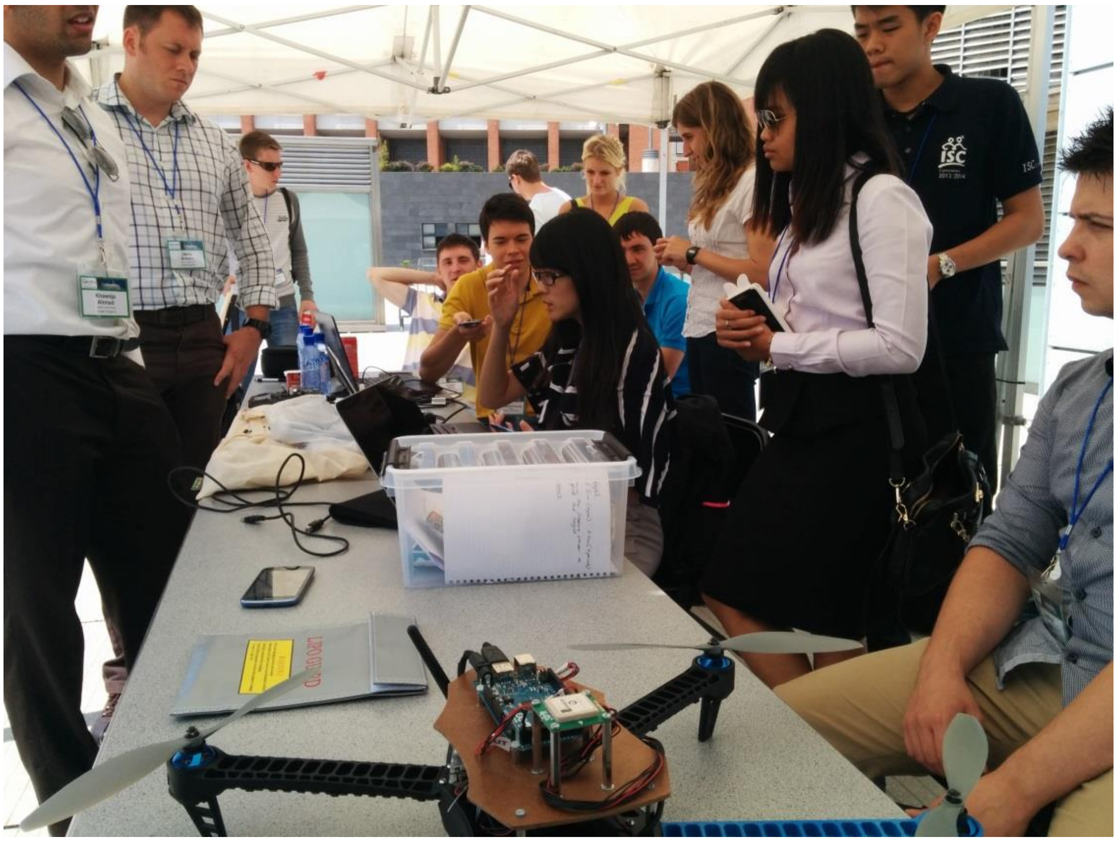


Figure 35: Participants working (2)

Figure 36: Participants working (3)

Figure 37: One of the quadcopters used for the challenge

La Vanguardia also made a great video of the challenge (in Spanish) [26].

5: Conclusions

The main objective was to create an air sensing platform that follows waypoints, takes photos and samples air quality. The unit we designed achieves this objective but in a much different way than our initial ideas.

The plan, at first, was to design and build a quadcopter from scratch but we quickly realized this task would take several months of development. Given our ignorance in the subject, we didn't anticipate all the work required to stabilize and fly the unit. At this point, we fully understood the multidisciplinary nature of multirotor helicopter design discussed in *1.1: Rationale*. With this limitation in mind, we went for a consumer quadcopter and modified it for our platform.

The second major change from our initial idea is the communication smartphone – quadcopter. As discussed in *2.2: How to control a quadcopter*, we went for an RC emulation; the proper way to do it would have been modifying the firmware of the flight control unit to talk to it directly through a standard serial connection or similar.
RC emulation works but is not "elegant"; implementing a communication link with the flight control unit is the first thing we would do as further development of this project. With this link and proper modifications, we would be able to access sensors of the quadcopter, like GPS and altitude sensor (both with higher precision than those on the smartphone).

In essence, we underestimated the expertise required to design a quadcopter and had to settle for a less ideal, albeit functional, solution.

With more time, our proposal of flight control unit, smartphone and Arduino would converge to a single board with direct access to the hardware, all the features of a flight control unit and smartphone-like connectivity built in. This solution would offer increased navigation performance (both speed and precision) and export precise information of the quad (such as heading, tilt, speed, raw sensor data, etc.) for applications that require it.

This capstone project has been a first step in the UAV world.

About the challenge; because of the limited time, it had to be simplified, more than we initially intended. Still, it was very interesting to see the ideas that came from engineers of different fields.

This platform however, even as it is right now, can be used for other challenges with different applications or goals. Having more time would let us introduce more things not strictly related to programming, like building the quadcopter with a DIY kit.

The most important aspect to keep in mind is that things go wrong. In our challenge, participant's decisions and modifications were limited and thus, we could add features like the manual override (2.4.1) to guarantee safety. If design is more open and there is more freedom, this kind of features can't always be in place; safety has to be implemented otherwise. A giant net that covers the flight zone perhaps? Why not. The more freedom you allow, the harder it is to keep things under control.

6: Appendices

6.1: Pulse Position Modulation

From Wikipedia, "Pulse-position modulation (PPM) is a form of signal modulation in which M message bits are encoded by transmitting a single pulse in one of 2^M possible time-shifts. This is repeated every T seconds, such that the transmitted bit rate is M/T bits per second."

Commonly used in RC, it multiplexes different channels into a single stream or signal.

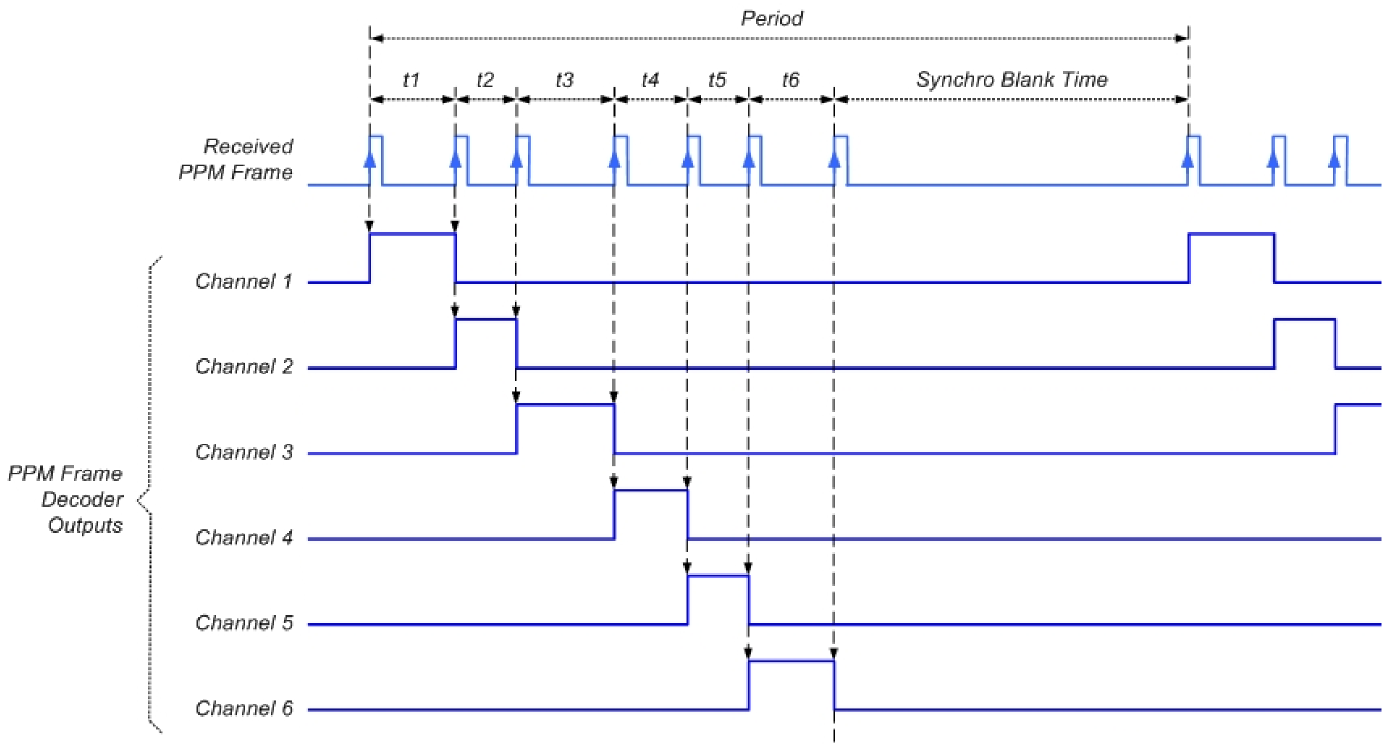


Figure 38: Example of a PPM stream

In our case we have 8 channels with values (pulse widths) of [1000, 2000] µs. The pulse width of the PPM stream is 400 µs and the total frame length (including blank time) is 27 milliseconds.

Also note that the polarity is inverted, i.e. the opposite of the example above.

6.2: Arduino – Android communication protocol

This protocol is designed to reduce processing time and thus workload on the Arduino.

Each message is encapsulated in a package that looks like this:

Command ID	Data	Data
1 Byte	2 Bytes	2 Bytes

Figure 39: Arduino - Android message blueprint

One byte to identify the command (256 total commands) and a total of four bytes to send data relative to said command.

Following this blueprint, there are three kind of messages:

- Notifications or commands: just the Command ID, only 1 byte.
- RC commands: Command ID to select a channel and 2 bytes of data with the value.
- Sensor reading results: Command ID identifying the sensor and 4 bytes for the reading.

These are the different commands used with their associated byte or ID; taken from *commands.h* of the *ArduinoADK* source:

```
// SENSOR RELATED COMMANDS
#define READ_SENSOR_TEMPERATURE 0x01
#define DATA_SENSOR_TEMPERATURE 0x11

#define READ_SENSOR_HUMIDITY 0x02
#define DATA_SENSOR_HUMIDITY 0x12

#define READ_SENSOR_NO2 0x03
#define DATA_SENSOR_NO2 0x13

#define READ_SENSOR_CO 0x04
#define DATA_SENSOR_CO 0x14

#define READ_SENSOR_ALTITUDE 0x05
#define DATA_SENSOR_ALTITUDE 0x15
```

```
// RC RELATED COMMANDS
#define SET_CH1 0xF1
#define SET_CH2 0xF2
#define SET_CH3 0xF3
#define SET_CH4 0xF4
#define SET_CH5 0xF5
#define SET_CH6 0xF6
#define SET_CH7 0xF7
#define SET_CH8 0xF8

#define SET_MODE_ALTHOLD 0xE0
#define SET_MODE_LOITTER 0xE1
#define SET_MODE_AUTO    0xE2
#define SET_MODE_RTL     0xE3
#define SET_MODE_STB     0xE4
```

Figure 40: Arduino - Android command IDs

6.3: GPS height precision

Before using the barometric pressure sensor, the only height reporting source was the smartphone's GPS but the performance was not very good.

The *GPSLogger* class of *QuadADK* writes all the GPS data that the *MyLocation* class outputs to log files in the Smartphone internal memory. Plotting these logs the result is the following:

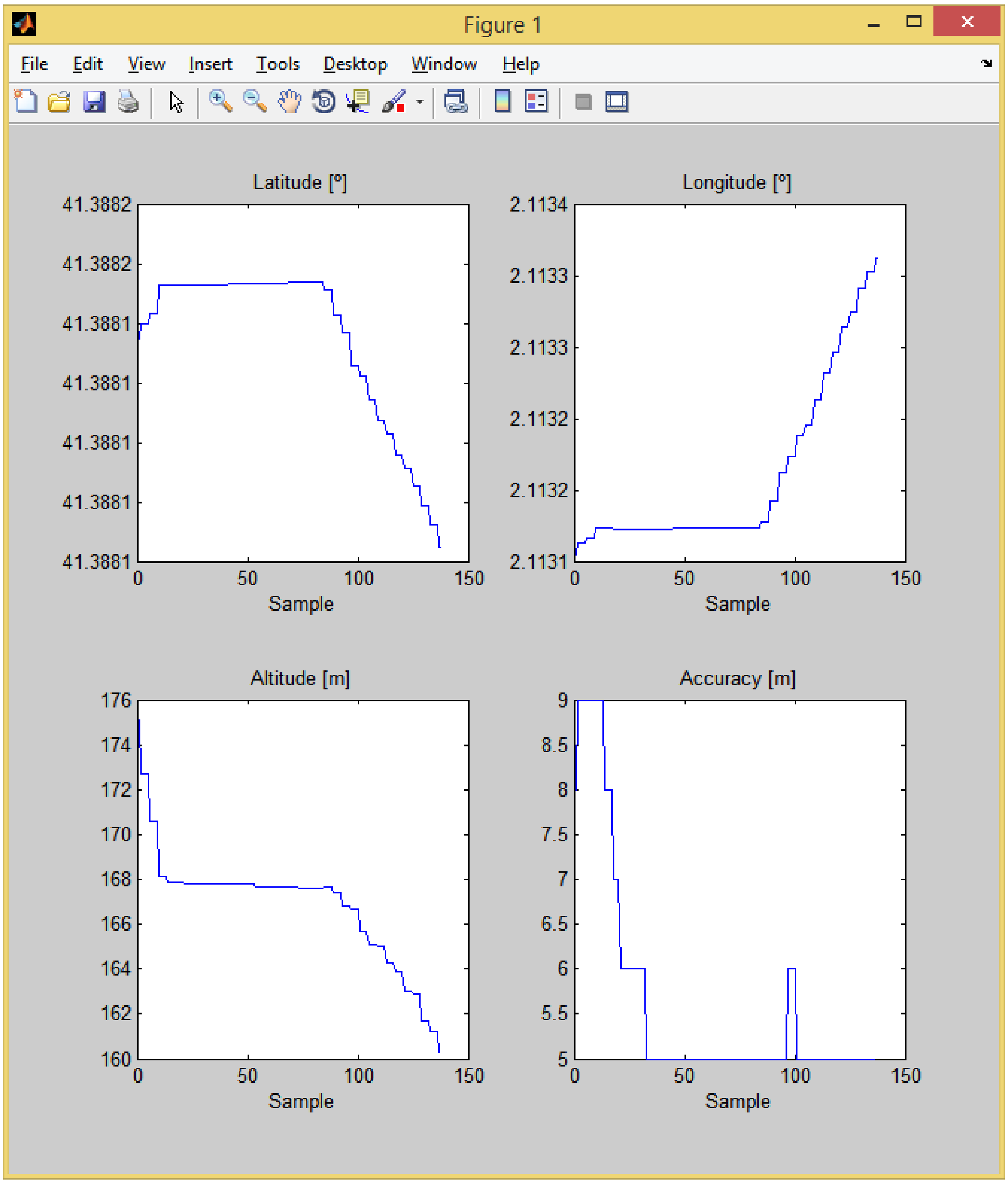


Figure 41: Smartphone GPS log

This log was taken walking on a flat surface with direct sky line of sight. Notice how the height, here shown as altitude, remains more or less constant when standing still but drops when moving.

This error renders the height reported by the GPS useless.

References

[1] Amazon, "Amazon Prime Air," Amazon, [Online]. Available: http://www.amazon.com/b?node=8037720011. [Accessed 2 September 2014].

[2] Google, "Introducing Project Wing," YouTube, 28 August 2014. [Online]. Available: https://www.youtube.com/watch?v=cRTNvWcx9Oo. [Accessed 2 September 2014].

[3] Forbes, "Drones Are Coming To Hollywood: FAA Set To Announce Approval For Use In Filming," Forbes, 23 September 2014. [Online]. Available: http://www.forbes.com/sites/gregorymcneal/2014/09/23/drones-are-coming-to-hollywood-faa-will-announce-approval-this-thursday/. [Accessed 29 September 2014].

[4] P. May, "Look up: The commercial drone market is about to take off," Mercury News, 1 March 2014. [Online]. Available: http://www.mercurynews.com/business/ci_25256472/look-up-commercial-drone-market-is-about-take. [Accessed 2 September 2014].

[5] C. Carruesco and G. Martínez, "Quad Repository," GitHub, [Online]. Available: https://github.com/carru/Quad. [Accessed 6 October 2014].

[6] 3D Robotics, "3DR Iris+," 3D Robotics, [Online]. Available: http://3drobotics.com/iris/. [Accessed 29 September 2014].

[7] D. Melanson, "Google announces Android Open Accessory standard, Arduino-based ADK," Engadget, 10 May 2011. [Online]. Available: http://www.engadget.com/2011/05/10/google-announces-android-open-accessory-standard-arduino-based/. [Accessed 1 October 2014].

[8] Google, "Accessory Development Kit 2011 Guide," Google, [Online]. Available: http://developer.android.com/tools/adk/adk.html. [Accessed 1 October 2014].

[9] M. Böhmer, Beginning Android ADK with Arduino, Apress, 2012.

[10] Wicked Device, "Egg Shield," Wicked Device, [Online]. Available: http://shop.wickeddevice.com/resources/air-quality-egg/egg-shield/. [Accessed 3 April 2014].

[11] SparkFun Electronics, "Altitude/Pressure Sensor Breakout - MPL3115A2," SparkFun Electronics, [Online]. Available: https://www.sparkfun.com/products/11084. [Accessed 3 April 2014].

[12] 3D Robotics, "3DR Iris Operation Manual," 3D Robotics, [Online]. Available: http://3drobotics.com/wp-content/uploads/2014/04/IRIS-Operation-Manual-v6.pdf. [Accessed 5 October 2014].

[13] APM Multiplatform Autopilot, "Mission Planner Manual Table of Contents," APM Multiplatform Autopilot, [Online]. Available: http://planner.ardupilot.com/wiki/table-of-contents/. [Accessed 1 October 2014].

[14] APM Multiplatform Autopilot, "Mission Planner Installer," APM Multiplatform Autopilot, [Online]. Available: http://ardupilot.com/downloads/?did=82. [Accessed 1 October 2014].

[15] 3D Robotics, "Using a ground station Mission Planner," [Online]. Available: http://3drobotics.com/wp-content/uploads/2013/12/using-a-ground-station-mission-planner.pdf. [Accessed 1 October 2014].

[16] 3D Robotics, "3DR IRIS - Calibrating the Accelerometer," YouTube, 14 April 2014. [Online]. Available: https://www.youtube.com/watch?v=8TWjZLlATIE. [Accessed 1 October 2014].

[17] 3D Robotics, "3DR IRIS - Re-calibrating the Compass," YouTube, 7 May 2014. [Online]. Available: https://www.youtube.com/watch?v=oD9Z9IR2TNU. [Accessed 1 October 2014].

[18] 3D Robotics, "3DRobotics IRIS How-To Videos," YouTube, [Online]. Available: https://www.youtube.com/playlist?list=PLszd_sCs2VsLLqvMM-_ixdkDH5d-k7dqb. [Accessed 1 October 2014].

[19] Wicked Device, "Air Quality Egg: Setting R0 Values," Wicked Device, 17 November 2012. [Online]. Available: http://shop.wickeddevice.com/2012/11/17/air-quality-egg-setting-r0-values/. [Accessed 1 October 2014].

[20] C. Carruesco and G. Martínez, "Quad Repository source code download," GitHub, [Online]. Available: https://github.com/carru/Quad/archive/master.zip. [Accessed 6 October 2014].

[21] Google, "Get the Android SDK," Google, [Online]. Available: http://developer.android.com/sdk/index.html. [Accessed 6 October 2014].

[22] Arduino, "Download the Arduino Software," Arduino, [Online]. Available: http://arduino.cc/en/main/software. [Accessed 6 October 2014].

[23] CDIO, "CDIO," CDIO, [Online]. Available: http://www.cdio.org/. [Accessed 6 October 2014].

[24] CDIO, "10th International CDIO conference," CDIO, [Online]. Available: http://lewis.upc.es/web/. [Accessed 6 October 2014].

[25] CDIO, "CDIO Academy 2014," UPC, [Online]. Available: http://lewis.upc.es/web/#academy. [Accessed 7 October 2014].

[26] D. Palacios, "Futuros ingenieros compiten con mini helicópteros no tripulados," La Vanguardia, 18 June 2014. [Online]. Available: http://videos.lavanguardia.com/tecnologia/20140618/54410073244/futuros-ingenieros-compiten-con-mini-helicopteros-no-tripulados.html. [Accessed 7 October 2014].

Glossary

ADK	Accessory Development Kit
ADT	Android Developer Tools
API	Application Programming Interface
ArduinoADK	Arduino program developed in this project
CDIO	Conceive Design Implement Operate
DIY	Do It Yourself
DP	Density independent Pixels
GPS	Global Positioning System
GSON	Google GSON
HTTP	Hypertext Transfer Protocol
I2C	Inter-Integrated Circuit
IDE	Integrated Development Environment
OS	Operating System
PHP	Server-side scripting language
PPM	Pulse Position Modulation
PWM	Pulse Width Modulation
QuadADK	Android application developed in this project
RC	Radio Controller
SDK	Software Development Kit
UAV	Unmanned Aerial Vehicle
USB	Universal Serial Bus