FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Bc. Michal Raška

Framework for Customizable Autopilot Solutions

Department of Software Engineering

Supervisor of the master thesis:	RNDr. Filip Zavoral, Ph.D.
Study programme:	Informatics
Study branch:	Software Systems

Prague 2017

Title: Framework for Customizable Autopilot Solutions

Author: Bc. Michal Raška

Department: Department of Software Engineering

Supervisor: RNDr. Filip Zavoral, Ph.D., Department of Software Engineering

Abstract: The thesis analyses, designs and implements the framework for creation of customizable autopilot solutions for radio controlled airplanes. As a proof of concept of this framework a set of applications, which use this framework, is created.

The result of this thesis is the extensible modular system, capable of airplane's attitude control, telemetry transmission and wireless communication with the ground station. There is great diversity in the components used by the system, which must be hidden by the framework. The differences must be encased in order to deliver required user experience for the programmers using the framework and the end users of the resulting applications as well. The tests, which validate the goals of the thesis indicate, that the resulting system is capable of all required tasks and ready to implement additional features which the end users might require in the future.

Keywords: autopilot framework radio controlled airplanes

I want to thank my supervisor, doctor Filip Zavoral, for his invaluable advices, his time and willingness.

I also want to thank my father, Ivo Raška, for helping with the building of the test aircraft and tests in real environment.

My special thanks go to my girlfriend Barbara, my parents and my sister for supporting me throughout the whole time.

Finally, I want to thank Captain James T. Kirk and Captain Jean-Luc Picard and their crew members for providing the very vital distraction.

Contents

1. Introduction

The increasing affordability of the radio controlled (RC) airplane models and the outbreak of the *credit-card* sized computers creates a vast range of new ideas and possibilities in fields such as the autopilot systems, telemetry, controlling model from the first person view (FPV) etc. The *purpose* of this work is to construct a *modular framework* which creates the environment for creating autopilot systems which address these new concerns of users.

1.1 Problem

There are existing solutions which address the particular problems such as autopilot system, FPV systems, flight stabilization[1] etc. Some of these systems can be integrated together, creating a more complex solution. However none of the existing solutions are easily modifiable. For example we wanted to feed my autopilot system with the GPS data from the Android phone, because the GPS module for the autopilot was quite expensive. It turns out that none of the current autopilot solutions can be modified easily to reflect our requirement, including the open source ones. We want to create a modular system, which can adapt the user needs and can be easily adjusted to meet the changing requirements. In the context of the example, the system should accept the GPS data from the mobile phone. Then, if we replace the GPS source some day, it should be possible to easily modify the system to integrate this new GPS source.

1.2 Aims and Objectives

The work has two main objectives:

1. **design a framework** for creating customizable autopilot solutions
2. **create an pilot application** based on the framework as a proof of concept

The framework is named **Up** and the implementation is named **Raspilot**. This work is highly motivated by the *Do It Yourself* principle (DIY). Therefore the result should be easily modified, understandable and easily usable by other users, which are familiar with the field of RC airplane models.

1.3 Thesis Outline

The thesis is divided into 5 chapters. In Chapter 2 the requirements, existing solutions and possible approaches are discussed more in detail. Chapter 3 explains design of the framework and the application more in detail. In Chapter 4 are stated the implementation details of the Up framework and the Raspilot. The last chapter evaluates the results of the work.

[1]Flight stabilization is a feature which balances the attitude of the airplane, for example in windy conditions.

2. Analysis

In this Chapter we state the *pilot's requirements* (requirements of the RC airplane pilot, who is the end-user of the Raspilot), the existing solutions, the functional and nonfunctional system requirements (these arises mainly from the pilots requirements) and the possible resources which can be used to meet these requirements.

2.1 Pilots Requirements

We can divide the pilot's requirements into two groups:

1. **hardware** - requirements which are related to the hardware which must be added to the airplane such as new sensors, microcontrollers, etc.
2. **features** - requirements which are related to the capabilities of the system such as stabilization, waypoint navigation, etc.

2.1.1 Hardware Requirements

The two most important hardware requirements are:

- hardware added to the airplane must be as *lightweight* as possible
- hardware added to the airplane must be as *small* as possible

The autopilot solution which requires heavy components on-board would be useless. The fuselage of the airplane is often *overstuffed* (with servos controlling the control surfaces, batteries, etc.) even without the autopilot system, so it is necessary to occupy *minimal additional* area.

Besides these there are some other requirements, which are not as crucial as the previously mentioned:

- **power consumption**[1] - should be as small as possible
- **impact protection** - can be solved by some *additional casing* of the on-board hardware (which adds weight on the other hand)

2.1.2 Features Requirements

The framework must be ready to implement all of the features usually found in existing solutions (for more information about existing solutions see Section 2.2). The most anticipated are the following:

- **flight stabilization** - balances the airplane's attitude, handy in windy conditions

[1]The power consumption inflicts also the weight. With great power consumption the system will need bigger and heavier power source.

- **return home** - the airplane returns to the takeoff location, safety feature used in case of RC signal loss
- **position, altitude, heading hold** - the system holds the specified position (usually the airplane pivot around the position) or the specified altitude etc.
- **telemetry** - various telemetry data can be transfered from the airplane such as airspeed, altitude, power source voltage etc.

It may appear that the features list is quite short, but all of the items stated are quite *complex tasks*.

2.2 Existing Solutions

There are currently many autopilot systems available. Some of them are expensive and full-featured, some of them are low-priced and lack some functionalities. The list of the most used follows:

1. RVOSD
2. Naza-M
3. Naze32
4. Multiwii
5. Ardupilot

Apart from these there are many others, but these are ones most frequently used and we have the experience with most of them.

RVOSD

The RVOSD [1] is the full-featured autopilot system for airplanes with integrated on-screen display (OSD). The OSD is perfectly implemented, the autopilot lacks some of the features (such as waypoint navigation) but overall it is surely a good choice when looking for hobby-level flight controller. Functions such as Return Home, Fly By Wire (another term for flight stabilization) are a matter of course. But the fact that the system is a closed-source makes any further custom-made extensions impossible.

Naza-M

The Naza-M [2] is a superb autopilot system solution for multirotors manufactured by the DJI. We do not have the personal experience with this system, but the Internet is full of very positive feedback. Naza-M also packs the OSD like RVOSD but also plenty of other features such as gimbal support (pivoted support used to stabilize the camera recording video), integration with iDevices (ground station app for iPads can be downloaded from the App Store). It is a total solution which can be used by professionals. The system is modular, so there is more freedom when configuring the system but only DJI certified products can be used. There is a possibility that components originating from other manufacturers will work with DJI modules, but there is no support for this from the side of DJI.

MultiWii and Naze32

Naze32 [3] is a modification (more precisely a GitHub fork) of the MultiWii [4] system. Both of these are missing many of the RVOSD and Naza-M features, but their application is quite different. While RVOSD and Naza are high end systems, these two are rather mid-range (with corresponding price tag). Naze32 is used in the racing multirotor, where the crash is very probable and the autopilot features are not required. Naze32 can be used without GPS module and in this configuration can only stabilize the multirotor which is quite sufficient for the racing purposes. Also OSD can be added, but is not present on the board by default. These two are open source projects, but their architecture and design are not formed to accommodate changes.

Ardupilot

As the name suggests Ardupilot [5] is the autopilot system solution based on the Arduino [6, 7]. Ardupilot has a big community, is open-source and actively developed. Also it can be used in multirotors, planes or rovers (possibly others, but these are the most common options). The Ardupilot is also very feature-rich. Autopilot functions, gimbal integration, telemetry feed which can be read by ground station computer, flight planning and many others. Also unlike Multiwii and Naze32 the firmware is quite well documented. The Ardupilot firmware is massive, therefore the potential changes must be integrated very carefully. When considering this project as the base for this thesis, we ran into several major problems such as quite specific hardware requirements etc. Therefore, we have decided to design and implement the framework from scratch. Also the history suggests that with the newly created framework the overall quality should be higher than patching the existing solutions and trying to force them into something they are not meant to be. One of the possibilities for the future development of the framework is to integrate the Ardupilot firmware as a flight controller module for example.

Summary

All of these autopilot solution have something in common. While excelling as an autopilot systems all of them *lack* the ability to be extended. RVOSD na Naza-M systems are *closed-source* products. We can, therefore, change only what the authors allow us using their configuration software. Naze32, Multiwii and Ardupilot on the other hand are *open-source*, but their design and architecture is not very friendly to changes. Also, the last three are licensed under the GNU GPL license, which can sometimes be limiting. Therefore, one of the goals of this thesis is to implement a change-friendly framework which can be easily modified or extended.

2.3 System Requirements

In this Section we will present the requirements for the framework and the implementation. Unlike the Section 2.1 which states the end-user requirements, this

Section states the **technical requirements** which must be taken into account when designing and implementing the system. The framework is named **Up** and the implementation is named **Raspilot**[1].

The Up and Raspilot must be able to execute these tasks:

- create **bi-directional** communication line between airplane and the user - downstream is mainly used for telemetry, upstream for sending messages to the airplane (such as enabling return home feature)
- **control** the airplane
- read **various sensor**

These requirements are discussed more in detail in following sections.

2.3.1 Communication Between Airplane and the User

The communication line between airplane and the user must be bi-directional. The *downstream* is used mainly for the *telemetry* and the upstream is used to send *messages* to the airplane. The communication line must be wireless of course.

When talking about wireless communication usually the WiFi or Bluetooth comes to mind. But these technologies are **not suitable** because of their **range limitations**. In case of Bluetooth the range is up to *100m* (but usually it somewhere around 20m). In case of WiFi the range is somewhere around *100-150m.* The range of Bluetooth nor WiFi is sufficient as the airplane might be *700m far* or even more.

Other possibility is to integrate our communication to the existing RC radio communication. This should be possible, however not all RC radios are capable of *bi-directional* communication. Also it would require some complex modifications of the communication protocol. Therefore, the Raspilot can be run only with RC systems, which have their protocol modified, which will narrow the range of usable RC radios. The other disadvantage of this approach is that in the case when we loose the RC signal, we loose also the communication with Raspilot.

Therefore we have decided to use the **cellular network**. Today the users usually have a data plan which can be used. The speed of the 3G or even the 2G network is *sufficient* for the expected data flow. The cellular data network uses the TCP/IP architecture which is well-known in IT world. Also when we use this approach, the system obtains another communication channel based on other technology [2]. Therefore the possibility of loosing both communication channels *at the same time* is lower.

2.3.2 Airplane Control

One of the core functionalities of the autopilot system is the ability to control the plane. To achieve this, the autopilot must know at least the *attitude* of the airplane. The attitude can be read from the *gyroscopes* and *accelerometers.* **Gyroscope** is used to determine the *orientation* of the airplane (based on the

[1]The name means: A **Ras**pberry Pi based auto**pilot** - the Raspilot.

[2]One channel is RC radio, the second channel is the cellular network

gravitation). **Accelerometer** measures *non-gravitational acceleration* and can be used to read the rate of attitude change. Usually gyroscopes and accelerometers are used together.

Besides gyroscopes and accelerometers the autopilot can also take the *location* into account. This enables features such as *return home* or *waypoint navigation.*

The gyroscopes and accelerometers allows the control of the planes attitude, the GPS allows calculation of the required heading (to get to the required location). In addition to these also other information can be used to deliver even more *reliable aircraft control* such as *airspeed*, *altitude* (can be read from GPS or from barometric sensor), current sensors (the plane might return in case when batteries are being depleted), etc.

However, the sensors gives us the *raw data.* For more information about origins of the data and how they are read refer to Section 2.3.3. These data needs to be processed. Almost all existing solutions uses the *PID Controller* [8], and therefore we have decided to use it also. The PID Controller is used to continuously calculate the output based on the input and previous decisions. The PID Controller is presented more in detail in Section 2.3.3.

2.3.3 Sensor Readings

The sensor data are required not only for airplane control, but also for *telemetry.* There are various types of sensor which have various communication interfaces. The Up must be ready to integrate these sensors and support as much communication interfaces as possible. The most common communication interfaces are:

- **UART** - Universal Asynchronous Receiver/Transmitter [9]
- **I^{2}C** - Inter-Integrated Circuit [10]
- **PWM** - Pulse Width Modulation [11]

For more information about the communication interfaces refer to following Sections.

The required sensors for the autopilot are the following:

- **IMU** - an inertial measurement unit (IMU) is an electronic device that measures and reports a body's specific force, angular rate, and sometimes the magnetic field surrounding the body, using a combination of accelerometers and gyroscopes, sometimes also magnetometers.
- **input from RC radio receiver**[1]

and the optional sensors which add additional features are:

- GPS
- barometer
- airspeed sensor

[1]In order to implement the stablization mode, the autopilot needs to take the user input into account in order to achieve the desired attitude.

- current sensor and voltmeter
- and many others

Some of these sensors has *more than one* communication interface, but that depends on the particular sensor. For example the RC radio receiver have usually only the PWM output, but the newer receivers support also Serial communication.

UART

The acronym stands for the **Universal Asynchronous Receiver/Transmitter** [9]. It is a device that translates the data between characters in computer (usually bytes) and an asynchronous serial communication format. The asynchronous serial communication format encapsulates the data between start and stop bits. UART is for example used for communication between computer and low-level peripherals or microcontrollers. The UART device is usually present on the microcontrollers and can be added as a USB device to those computers, which lack this device.

I^{2}C

I^{2}C or the **Inter-Integrated Circuit** [10], is used to attach lower-speed peripherals to processors and microcontrollers. I^{2}C is used for short-distance intra-board communication. For example the IMU sensor usually has the I^{2}C interface. I^{2}C devices are connected to a special port of the microcontroller or computer.

PWM

The PWM stands for **Pulse Width Modulation** [11]. The PWM is a technique of encoding message into pulsing signal. The principle is changing the widths of otherwise regular rectangular signal. In RC models the PWM is used to *control the positions* of servos and the *speed* of the motors. The servos and the motors are connected to the radio receiver. The radio receiver generates the PWM pulse based on the received signal. For more information about PWM in context of RC radios refer to Section 2.4.1.

PID Controller

The three letters in PID stands for:

- **P** - proportional
- **I** - integral
- **D** - derivative

The PID Controller consists of these three terms.

The PID Controller [8] continuously calculates an *error* between the *desired value* and the *measured variable*. Desired value is often called the **setpoint** and the measured variable is called **process variable**. The controller tries to *minimize* the error by adjusting the **control variable**, which in our case is the position of a servo. The output is determined by the following formula [8, p. 293]:

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau) d\tau + K_d \frac{de(t)}{dt}$$

where K_p, K_i and K_d are **non-negative** coefficients for the proportional, integral and derivative terms.

One of the biggest *issues* with the PID Controller is the need to **tune** the K_p, K_i and K_d parameters. If the parameters are not set correctly, then the autopilot tends to be very *unstable* and sometimes *absolutely fails* to control the aircraft. The values of the parameters depend on many factors such as used sensors, overall stability of the aircraft, servos speed etc. Therefore, they must be tuned in *every aircraft.*

The *effects* of the three coefficients are the following:

- K_p - the proportional term gives us the **present error**, which can be modified by this coefficient. When this coefficient is a large number then the error will also be large and the control variable will be adjusted heavily. When this coefficient is a small number then the error will also be small and the control variable will be adjusted lightly.
- K_i - the integral term accounts for the **past values of the error**. If the current output is not strong enough, error will accumulate over time and the controller will respond with stronger action.
- K_d - the derivative term accounts for **possible future values** based on the current rate of change.

2.4 Proposed Components of the Framework

Based on the previous Sections, this Section discusses the hardware and software requirements which must be fulfilled in order to meet the previously stated demands. At first we will state the needs which arises from the requirements and then the possible solutions will be presented. Let us start from the things *we cannot change* and that is the RC radio receiver.

2.4.1 RC Radio Receiver

The RC Radio Receiver is a hardware peripheral which usually *controls servos and motors*. RC radio receiver provides the stabilized 5V power source and controls the positions of the servos. Receivers vary in many parameters such as frequency, number of channels, output interface (PWM, Serial, etc.), input voltage tolerance etc. These specific attributes must be encapsulated and the illusion of the single type of the receiver must be provided to the system.

PWM Pulse

The information, in this case the *desired position of the servo*, is encoded into the *width* of the pulse which is otherwise regular and rectangular. The standard is that the minimal angle is represented as a 1000 µs wide pulse, the center position as a 1500 µs wide pulse and the maximum is represented as a 2000 µs wide pulse. However, the marginal values differ between different manufacturers. The figure 2.1 shows the illustration of the pulse width and the servo position relationship.

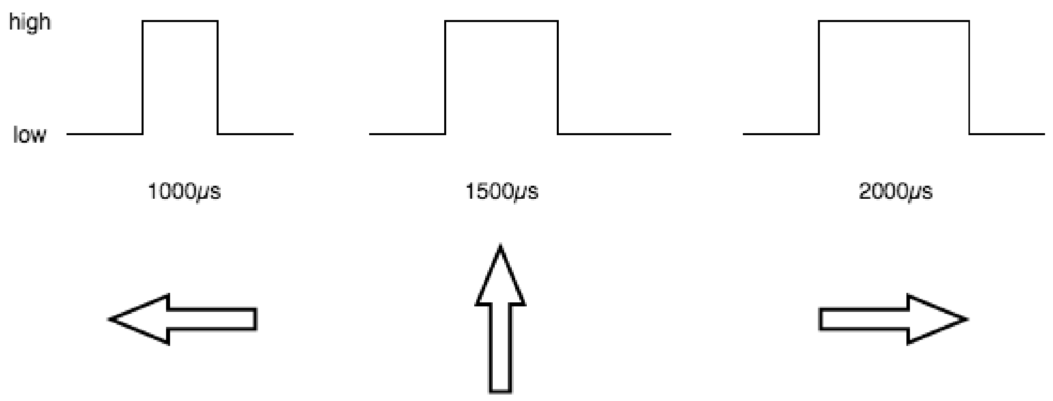


Figure 2.1: PWM Pulse and Corresponding Servo Position

2.4.2 Servos and Electronic Speed Controlers

Both **servos** and **electronic speed controllers** (abbreviated as ESC) are hardware peripherals. The servos allow *precise control* of angular position and are used to move the control surfaces[1] of the aircraft. Usually servos control ailerons (adjusts roll), elevator (adjusts pitch) and rudder (adjusts yaw). Other surfaces such as flaps (if present on the aircraft) might be also controlled with servos as well as other functionalities such as gear. Besides servos we need to control the thrust. Speed of the motor (or possibly multiple motors) is controlled with ESC. ESC is an electronic circuit which varies the speed of the motor. There are multiple types of ESC but the principles of ESCs are not subject of this thesis. Servos and ESCs have something in common and that is the input. Both read the PWM signal and react to the received pulse. In case of servos the angle is altered, in case of ESC the speed is altered (1000 µs wide pulse means motors off and the 2000 µs wide pulse means full throttle).

2.4.3 Microcontroller

The Raspilot needs a platform which will *host* the autopilot and *integrate* all the other components such as sensors, radio receiver and can control the servos. As stated in Section 2.4.1 we need to reliably *read and generate PWM* pulse. Section 2.3.3 states various communication interfaces which we must be able to integrate. Based on these requirements we decided to use a microcontroller to carry out these low-level tasks. There are many microcontrollers available, but we have selected the *Arduino* because of its community and support.

Arduino

The Arduino [7] is an *open source prototyping* platform based on the *ATMega* [12] microcontroller. The microcontroller allows execution of time precise operations

[1]Control surface is movable part of the aircraft which allows to control the airplane's attitude

such as PWM reading. Therefore, the Arduino is suitable for PWM readings and PWM generation (required for servo and motor control and the airplane control). Besides this Arduino can handle I^2C, UART and many other communication interfaces.

2.4.4 The Core

The Arduino handles the low-level tasks, but the higher-level tasks such as communication between the airplane is hard to implement on the Arduino. Therefore, we need a component which will carry out these tasks:

- provide a communication channel between user and the airplane
- transmits data through the channel, such as telemetry, messages, etc.
- executes user required actions

We wanted to provide as much *comfort for the programmers* which will use the system as possible. With the explosion of card-sized computers this is quite easy to achieve as many of them runs Linux which is a well-known environment. There are also many card-sized computers available but we have selected the *Raspberry Pi* because of the same reasons why we have chosen Arduino and that is the community and support. Also the characteristics of the card-sized computers, such as dimensions, computational power are quite similar.

Raspberry Pi

The core of the Raspilot will be run on the Raspberry Pi [13]. Its dimensions, weight, power requirements and computational capability [13, 14] are *perfect match* for the anticipated situations. With its size it should be fitted into any reasonably big aircraft without complications. Also the relatively low weight (45g) won't change the flight characteristics. The Raspberry is powered by 5 V and draws 0.5 A [14]. Both of these power requirements are easily met by the todays batteries which are commonly used in the RC aircraft models. The Raspberry itself comes in many models which vary in the computational power:

- **model B+** - 512MB RAM, 700MHz Broadcom BCM2835 CPU
- **model 2** - 1GB RAM, 900MHz *quad-core* ARM Cortex-A7 CPU
- **model 3** - 1GB RAM, 1.2GHz 64-bit *quad-core* ARMv8 CPU

. It must be possible to run the system on the most used models (in time of writing this thesis it was the model B+, model 2 and model 3). Therefore, the core of the system should be as **platform independent** as possible.

2.4.5 Sensors

In the previous Sections we have discussed the RC radio, servos, and computation hardware which will execute the autopilot. But we still do not have analyze the *sensor data* which the autopilot needs.

There is a vast range of sensors types and a vast range of manufacturers which creates these sensors. However, we can use something which many of us has in their pockets and that is the *mobile phone*. Almost all of todays mobile phones are equipped with a GPS sensor and some of them also have gyroscope, accelerometer, humidity sensor, barometer, etc. All of these sensors can be read and used by autopilot if they are integrated into the Raspilot. When it comes to the smartphones we generally have three possible choices:

1. **iPhone** - not very suitable, because of the price tag
2. **Windows Phone** - we don't have much experience with this platform
3. **Android** - reasonable price tag, well documented, quite popular

We have chosen the Android because of the price tag and our personal experience with these devices.

Android Device

Nowadays, the Android devices have many sensors (such as IMU, GPS, barometric sensor...). The Raspilot relies on the data from these sensors. Android device is connected to Raspberry Pi via TCP/IP[1]. We need to create a connection between the on-board Android and the Raspberry Pi. The connection can be wired or wireless. The wireless connections are not very suitable because they might interfere with the RC radio. Therefore we have decided to use the wired connection. The Android device can create a TCP/IP network and we have decided to use this feature of the Android platform. Besides sensor data Android device will also provide the necessary Internet connection for the Raspilot which will be used for the communication between airplane and the user.

2.4.6 Communication via TCP/IP

In the Section 2.3.1 we have stated the reasons why we have decided to use the cellular network for the communication channel between user and the airplane. The choice of the cellular network implies that the communication will be based on the TCP/IP network. The TCP/IP networks offers two main protocols, the TCP and the UDP. Let us discuss why we have chosen to make possible to use both these protocols in the Raspilot.

The TCP protocol is *connection-oriented* and *reliable* but is much more *resource demanding*. The UDP on the other hand is *connection-less* and *unreliable*, but much more *simple*.

Besides the communication between user and the airplane, there is also a communication line between on-board Raspberry Pi and the on-board Android device, which serves as a sensors source. This communication is also based on a TCP/IP network. We want these communication line to be as *fast* as possible because we want to *minimize* the delay between sensors readings and processing of the sensor data. The reading is executed on the Android device, then the data are transferred to Raspberry Pi where they are being processed. The communication

[1]The Android devices are capable of creating TCP/IP network and because this network architecture suits the needs of the Raspilot, we have decided to use this feature of the Android.

line between on-board devices is quite *reliable* so we find the reliability of the TCP protocol quite redundant here. But the Blackbox Mode (see Section 2.4.7 for more information) makes this situation a bit more difficult. The Blackbox Mode uses only the on-board Android device for telemetry readings. The control capabilities are *unavailable*. We need however, to connect somehow the on-board Android device with the user. This can be done only via the TCP protocol for the same reasons as stated in the next paragraph, which are mainly the *limits* of the telecommunication operators. Therefore, we have decided to make possible to use *both* UDP and TCP protocol for communication between on-board devices. The choice will be based on the particular situation.

The communication channel between the user and the airplane is a different case. The data transfer channel over the cellular network is quite *unreliable*. The possibility of signal loss is quite high due to the signal coverage, interference, etc. This indicates that between the user and the airplane the TCP should be used. This decision is supported also by another fact. The smartphones usually do not have public IP addresses because of the telecommunication operators. It is possible to have a public static IP but usually it is charged with some monthly fee and therefore, we cannot expect the mobile phone will have a public IP address. Please note that the user will be connected to the Internet probably via the mobile phone as well (although some other connection types might be used, which might have a public IP). Nevertheless the on-board device is surely connected via cellular network (special cases when the user uses WiFi or other short-range connection are not taken into account). Because of the limitations stated at the beginning of this paragraph, we *cannot* send the UDP packets between the user and the airplane because the on-board device might not have a public IP and might be not on the same network. Therefore, the TCP *must* be used for communication between the user and the airplane. In fact some kind of **proxy** server is also required, because there is no possibility to establish incoming connection to the mobile phone. This proxy server will have two channels and upon **receiving** the message in **one channel**, it **sends** the message to the **other channel**. The two channels corresponds to the airplane and the user. This enables the bi-directional (full-duplex) communication between the user and the airplane.

2.4.7 Operation Modes

In many cases the autopilot feature is not absolutely necessary and the user wants only to read the telemetry. Therefore the system can be run in two modes:

1. **Flight Controller Mode**
2. **Blackbox Mode**

The differences and use cases for these modes are presented in the following Sections.

Flight Controller Mode

The Flight Controller Mode is the *full-featured* mode, but it requires *all* of the components to be available. Therefore it takes more space, weights more, consumes more power, etc. The required components are the following:

- **Arduino** - for reading and generating PWM, reading sensors, and other low-level, time critical tasks
- **Raspberry Pi** - runs the core of the Raspilot
- **Android**, or other sensor source - to provide the required data for the Raspilot
- **power source** - for the Raspberry Pi and Arduino, Android is usually powered from its own battery

Please note that we do not list components such as radio receiver, servos, batteries, etc. because these must be present in the airplane even if we do not have Raspilot installed.

This mode is used when the user wants the autopilot features and the added weight is not an issue.

Blackbox Mode

The Blackbox Mode *lacks* the control functionalities, but the benefit of this mode is *lower weight* and fewer on-board devices. In fact only the Android is present on-board in this mode, which decreases the weight and also the space requirements quite significantly.

This mode is used when the control capabilities are not required and the telemetry is sufficient.

2.5 Non-functional Requirements

To this point the requirements to the functionalities of the autopilot system are analyzed. But other than the autopilot implementation the thesis should also design a framework for creating such systems. Apart from the functional requirements the non-functional requirements are also very important especially when designing a framework. Following non-functional requirements were identified as critical:

- Modularity
- Interoperability
- Extensibility
- Documentation
- Robustness
- Fault tolerance
- Portability
- Testability

All of these must be taken into account during design and development.

Modularity, interoperability and extensibility are connected together. If the system should be extensible we think it must be as modular as possible. Also when we want to add the ability to cooperate with other systems in the future, it should be achieved by adding new module to the system not by changing the whole system.

The first three must be supported by very good **documentation**. It will be meaningless to have perfectly modular and extensible system when no one will understand it.

The **robustness and fault tolerance** are required because of the nature of the system. It will be unpleasant if the aircraft will crash only because the corrupted data were received. It's crucial that the system will be fault tolerant and will behave as normal as possible even if some of the components will fail. In case of fatal failures the system must execute such actions to minimize the loss.

The provided implementation uses the Raspberry Pi, but some other platforms might be used to run the system in the future so the system must be as **portable** as possible. This will impact the whole system design. Well designed, loosely coupled system is surely ported to other platforms more easily than a tight coupled one.

The **testability** requirement is necessary, because no one would fly an not tested flight controller. Because of the modularity and documentation requirements, the system should be tested easily. The modules must be loosely coupled with well documented interfaces.

3. Design

In this Chapter we present the design decisions as well as reasons why we have decided to solve the problems in such fashion.

3.1 Overall System Architecture

Let us begin with a diagram (see Figure 3.1) which shows the high level view of the system architecture. The solid lines represent the local connections and the dashed lines the remote connections. Duplex connections have arrows on both sides, while simplex connections only on one side, on the side of the receiver.

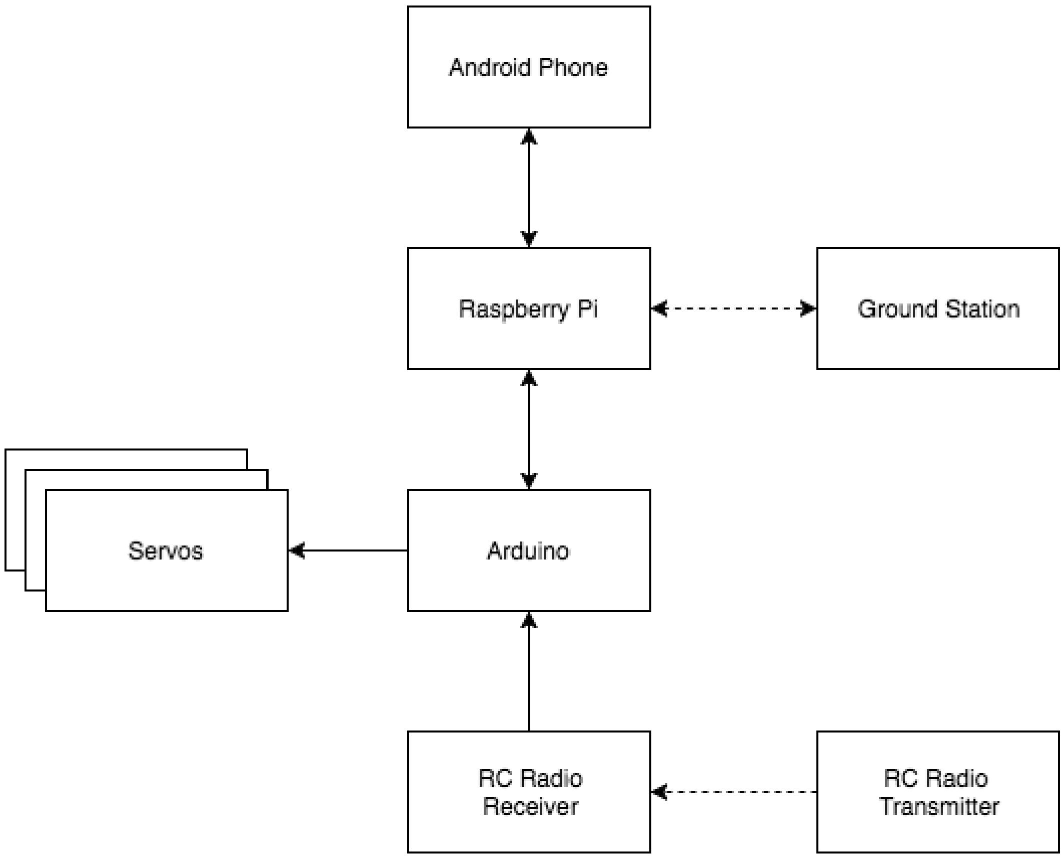


Figure 3.1: High Level View of the System Architecture

The system architecture is implementation dependent. Other implementations might not need a particular component or might add some others. For example the Raspilot architecture in the Black Box mode follows the diagram shown on Figure 3.2 (grayed-out components are missing).

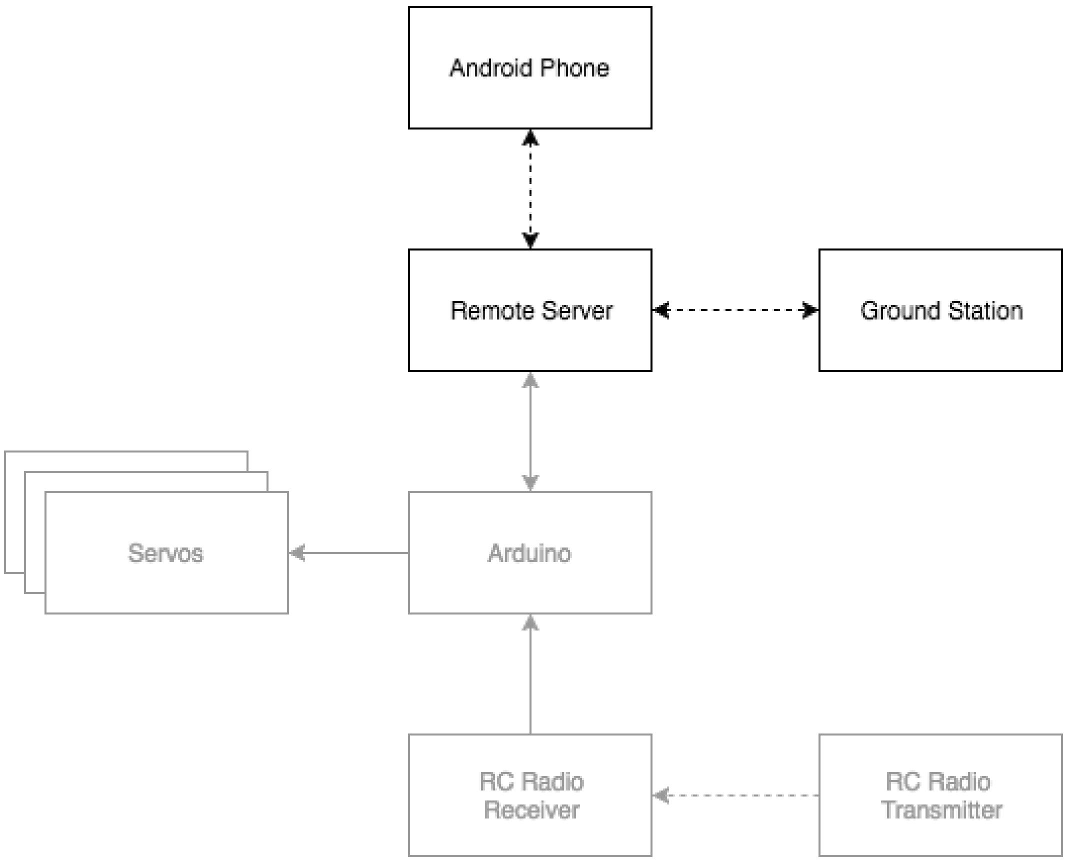


Figure 3.2: High Level View of the System Architecture when running in Black Box mode

3.1.1 Raspberry Pi

The Raspberry Pi hosts and runs the core of the whole system and it is an on-board device. Reasons why we have decided to use Raspberry Pi are explained in Section 2.4.4. For comprehension the list of functionalities which Raspberry deals with follows:

- runs the autopilot system (which uses the Up framework)
- reads sensor data of the Android device
- reads data from the Arduino and sends data from which then the Arduino generates the PWM pulse for the servos and ESCs
- communicates with the ground station

The exchanged data and connection architecture are described in following Sections.

3.1.2 Arduino

The drawback of the typical Linux environment is the inability to execute real time operations. Measuring the time intervals in orders of µs is undoubtedly time critical. We decided to use Arduino because of the existing community, documentation and overall support for this platform.

The Arduino exchanges data with Raspberry via the Serial connection. This connection type is easiest to implement. The Serial communication works out of the box on the Arduino platform, while the other types of the communication requires modifications and possibly some additional hardware. Therefore, we have decided to use this communication channel between Arduino and Raspberry Pi.

The Arduino is responsible for:

- reading the PWM output of the RX
- communication with the Raspberry
- generation of the PWM pulses (to control the servos and ESCs)

Communication Protocol Between Arduino and Raspberry

The Serial connection is byte oriented and the communication protocol is message based. Messages passed in the Up and Raspilot are called **Commands**. The message outline is following:

```
command_type|payload
```

Figure 3.3: Outline of the Message Exchanged Between Arduino and Raspberry

Please note, that the | character is not present in the message.

`command_type` is a single character[1]. This character specifies the Command type. Based on the Command type the payload length varies. Some of the Command types might not have payload at all.

`payload` is the data packet associated with the Command. For example the Command which holds the servo angles which should be set, has 4 numbers in range <1000, 2000>representing the four PWM values of the servos and ESC.

The serial communication protocol is not reliable but we consider the communication line between Raspberry and Arduino reliable enough that no confirmation mechanisms are used. The only fault tolerance method used is the timeout of the Command. If whole payload of the Command is not received in the specified time limit, the whole message is discarded and the system returns to the state when it awaits a new Command.

Some of the Commands (such as PWM readings) must be sent as often as possible, but we might overload the Raspberry processing capability. This is handled by the Panic Mode which is described more in Section 3.2.5.

3.1.3 Android device

Today's Android phones are equipped with plenty of sensors. For example the Nexus 5 which is used as a testing device has the following:

- accelerometer and gyroscope

[1]The consequence of command_type being only a single character is that there are only 256 possible Command types (if we include all possible values of a byte type). However, this should be sufficient. If not, it should be easy to extend the command_type to two or more bytes.

- GPS
- barometer sensor
- compass
- many other which are not used actively now such as temperature sensor, proximity sensor, etc.

Some of them (accelerometer, gyroscope, GPS) are necessary and some of them are great if more precise data (barometric altitude is usually more precise than GPS altitude) are required.

All of the required sensor data are being forwarded to the Raspberry. Android device is a **source** of sensor data which are later processed by the Raspberry.

The sensor data varies in means of update frequency, precision, importance in the whole process of aircraft stabilization and piloting. For example the GPS data are updated much less often than the data from accelerometer and gyroscope. Also it is more acceptable if GPS data are delayed than if the orientation data would be delayed. Therefore two protocols were designed:

- Orientation Forward Protocol
- General Message Transfer Protocol

We think that it is more robust and easier to use if the two protocols are separated. The orientation data must be delivered as *fast* as possible but is *strictly specified* how these data are formatted and what they contain. On the other hand some other data such as the GPS location does not need to be delivered as fast (but some time limits still apply). Also the Commands might have varying types of payload, might require confirmation etc. Therefore, we have split the communication between Android and Raspberry between two independent protocols. Both of these protocols can use the TCP or UDP connection.

Orientation Forward Protocol

The Orientation Forward Protocol handles only one type of Commands - the data from accelerometer and gyroscope. The Android framework has built-in support for reading the current orientation of the device as well as reading the rotation rate around the three axes. The protocol sends six numbers in this order:

1. **roll** - rotation around the front-to-back axis
2. **pitch** - rotation around the side-to-side axis
3. **yaw** - rotation around the vertical axis
4. **roll rate** - rotation speed around the front-to-back axis
5. **pitch rate** - rotation speed around the side-to-side axis
6. **yaw rate** - rotation speed around the vertical axis

The Figure 3.4 depicts the roll, pitch and yaw axes of the airplane

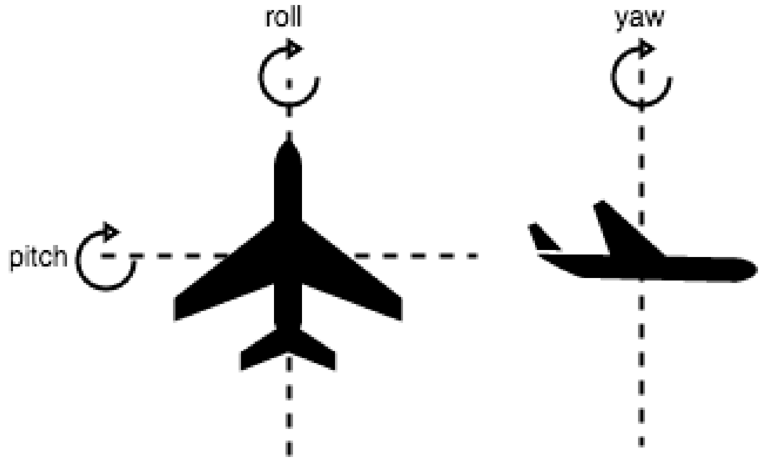


Figure 3.4: Illustration of the roll, pitch and yaw

Raspberry receives the orientation data which are then used in the stabilization and flight controlling process. The orientation data should be sent as fast as possible, but we must ensure that the data flow from the Android device will be not excessive and will not overwhelm the Raspberry. This is handled by the Panic Mode which is described more in detail in Section 3.2.5.

General Message Transfer Protocol

Besides the orientation data other types of Commands must be also exchanged such as GPS location updates, the previously mentioned Panic Command, etc. All of these are sent via the General Message Transfer Protocol.

The speed is not so crucial as in the Orientation Forward Protocol. Therefore we want the data to be as human readable as possible. In such cases the XML or the JSON is usually used. XML is more easily read by the humans, but also less efficient than JSON. Therefore, we have decided to use the JSON format, because it is more efficient yet still readable. All messages must follow this outline:

```
{
  "id": "unique identifier of this Command instance",
  "name": "unique name determinig the type of Command",
  "data": {
    \\ can contain any valid JSON object
  }
}
```

Figure 3.5: General Message Outline

The field `id` is used for purposes of the confirmation or replying to Commands. Might be not set when this is not required.

The `name` field determines the type of the Command. Based on the value of this field the `data` field is parsed and processed. This field must be always set.

The `data` field holds any data which are relevant to this Command type. For example the GPS location update Command holds the latitude, longitude and altitude in its data field. Content of this field is very variable and might be not set at all.

If a malformed message is received it is discarded.

3.1.4 Ground Station

The Ground Station serves as the *mission control center*. It receives telemetry and sends various Commands to the aircraft. These Commands might alter the flight mode (stabilize, return home, etc.), the required altitude, heading etc. In the Raspilot an *Android device* is used as a Ground Station, but it might be some other platform[1].

One of the main purposes of the Ground Station is message exchanging. Usually the telemetry data are received and various Commands are transmitted. However, the potential *delays* in message deliveries are not as unacceptable as in case of orientation data. The communication is transmitted over the cellular network, therefore it is likely that the data will be delayed for a very long period of time (in order of hundreds of milliseconds). The connection is obviously unavailable if the mobile phone signal (Ground Station as well as the aircraft uses the mobile data plan) is lost. Therefore, the system *must be able to overcome* these delays and connectivity issues.

The *portability* requirement must be also taken into account. Therefore a platform independent communication technique must be used.

Let us summarize the requirements for the communication channel between Ground Station and the aircraft:

- various Command types
- various payload types depending on the Command type
- delays are acceptable

All of these are implemented in the *General Message Transfer Protocol* (see Section 3.1.3). That is why we have decided to use this protocol also for the communication channel between Ground Station and Raspberry Pi.

3.1.5 RC Receiver and Servos

The RC radio receiver (RX) and the servos are not implemented nor designed by us. In the Raspilot the RC receiver outputs PWM pulse for each controlled channel[2].

The servos which are usually connected to the receiver, receives the PWM generated by RX and sets the required angle. However, because of the stabilization requirements, the servos are connected to the Arduino.

When the stabilization functions are disabled the Arduino only forwards the read PWM to the servos. If the stabilization functions are enabled, the PWM is

[1]Another Raspberry Pi is being considered as possible Ground Station platform

[2]The channel corresponds to one ability of the aircraft. For example the ailerons are one channel, the elevator is second, gear is third, etc.

read and forwarded to Raspberry. The Raspberry calculates the new PWM values based on the orientation data, received PWM (and possibly other variables) and sends them to the Arduino. Arduino generates the PWM based on the received data.

Figure 3.6 shows the flow when the stabilization functions are enabled. In case when stabilization functions are disabled after step 1 the system continues with step 6.

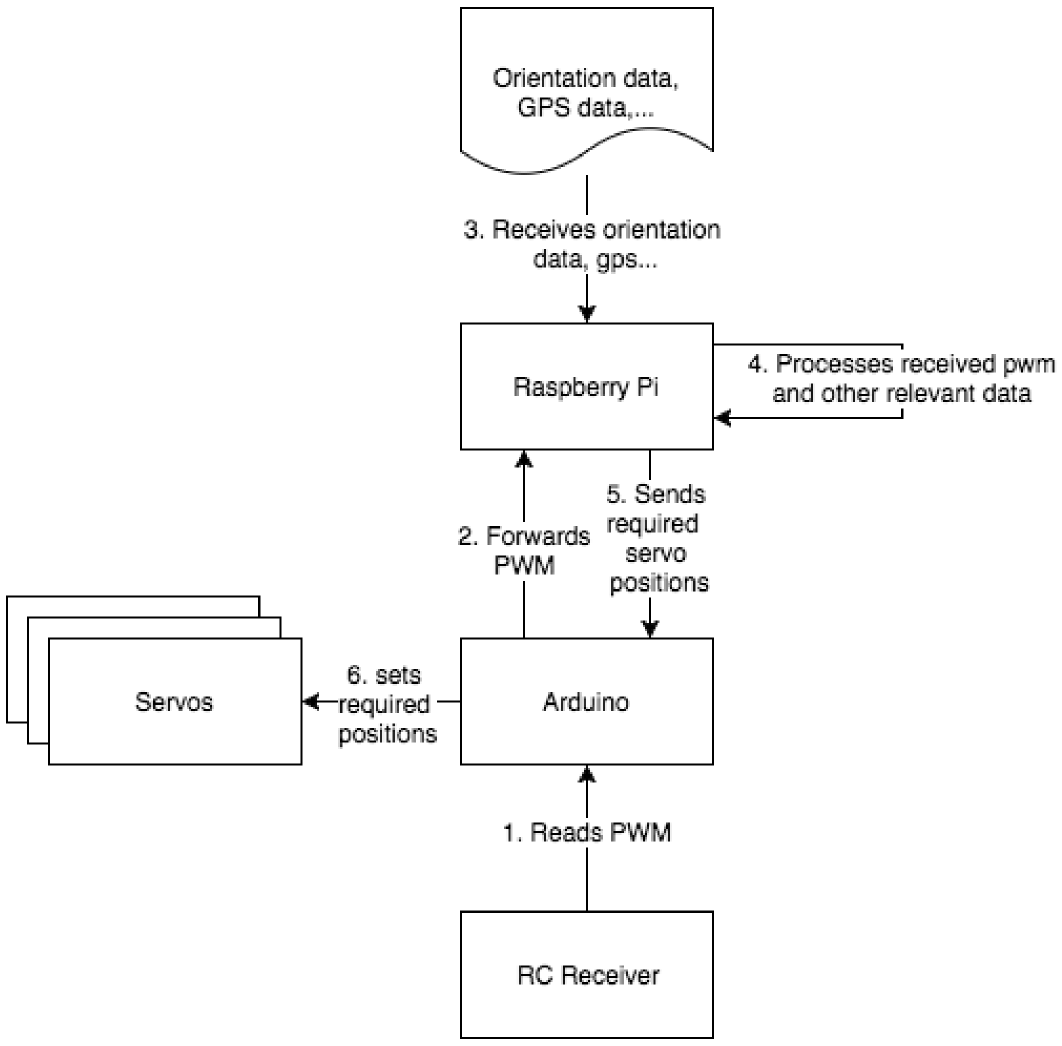

Orientation data, GPS data,...
3. Receives orientation data, gps...
Raspberry Pi
4. Processes received pwm and other relevant data
2. Forwards PWM
5. Sends required servo positions
Servos
6. sets required positions
Arduino
1. Reads PWM
RC Receiver

Figure 3.6: Flow of the PWM Signal When Stabilization Is Enabled

3.2 Framework Architecture

One of the main goals of this thesis is to design a framework for creating autopilot systems. So far we have discussed the overall architecture. But this architecture, as has been stated, is very implementation dependent. For example some implementations might use other sensor sources than Android device. The Up framework must encapsulate these implementation details. In this Section we will thoroughly discuss the Up framework design and why we have decided to use the selected approach.

3.2.1 High-level View

The Figure 3.7 shows the diagram of the framework's architecture. View of this diagram is high-level, more detailed diagrams are shown in the following Sections.

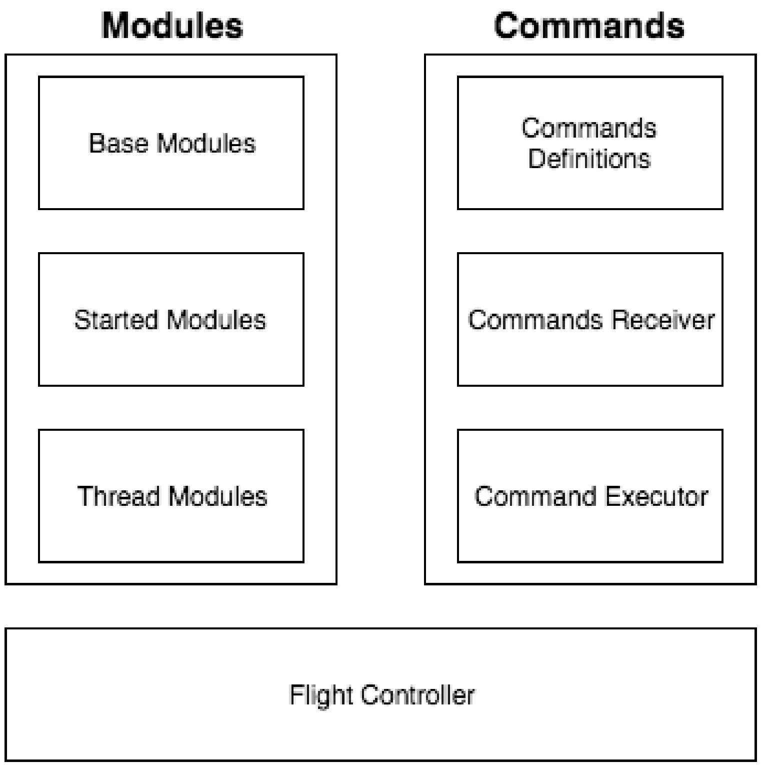


Figure 3.7: High Level View of the Framework Architecture

Modules

The core of the whole Up framework are Modules. Each Module has its own single function.

For example the RX Provider is a Module. The responsibility of this Module is to read the RX PWM data and make them available to the rest of the framework.

During the system start-up the Modules are registered within the framework. The communication between Modules and other components is possible.

There are some special types of Modules which are discussed more in detail in Section 3.2.2.

Commands

The Commands component consists of three subcomponents which are related very closely.

The **Commands Definition** component holds the actions of all available Commands, which the system can respond to. Upon receiving the Command, the action defined in this module is executed. Example of such Command might be the Arm Command. When this Command is received, the engines should be armed [1].

[1]When the engines are disarmed, the RX output of the throttle channel is being ignored and the PWM with frequency of 1000 µs is generated regardless of the RX output value. This

The receiving of the Commands is handled by the Commands Receiver component. The purpose of this component is to receive the Command, ensure its validity and forward it to the Command Executor component. Invalid Commands are discarded.

The Commands Executor receives Commands from the Commands Receiver. Then it looks for the definition of the Command (in the Commands Definition component). If the Command is defined, it executes the appropriate action. The undefined Commands are ignored.

Flight Controller

The Flight Controller is the most complex component responsible for the the flight control process (such as stabilization, navigation, etc.). In order to do so it uses many other Modules. The Flight Controller is explained more in Section 3.2.3.

3.2.2 Modules

In this Section we present the Modules component more in detail, especially the different types of Modules and their purpose in the whole system.

Basic Module Types

The **Base Module** as the name suggests, is the base of all other Modules. It does not have a special purpose besides some implementation related matters. However, it is very important because all other types are derivations of the Base Module.

The **Started Module** is an extension of the Base Module. The added functionality is that this Module can be started and stopped.

The initial versions uses the threading model where there was **one thread per Module** which turns out to be very inefficient as the number of Modules rose. Therefore, the concept of Thread Modules were introduced. The **Thread Module** is an extension of the Started Module which spawns a thread during its start. The Thread Modules are used for recurring and blocking tasks in general.

With the presence of Thread Modules the whole system can be single threaded and only those Modules which needs to perform blocking tasks (such as networking) spawns additional threads. The number of Thread Modules should be limited.

Provider

The Provider Module is responsible for supplying some functionality to other Modules. The following providers are available:

- **RX Provider** - provides the RX related data to the rest of the system
- **Orientation Provider** - provides the orientation data to the rest of the system

is a safety feature.

- **Mission Control Provider** - provides the connection with the Ground Station to the rest of the system
- **Location Provider** - provides the GPS location data to the rest of the system
- **Altitude Provider** - provides the altitude of the aircraft to the rest of the system
- **Heading Provider** - provider the heading of the aircraft to the rest of the system

All Providers are extensions of the Started Module.

Controller

The Controller is a special type of Module, which controls some of the other Modules. They are usually used to execute recurring tasks such as telemetry transmission. The Controllers manage the Recorder Module generally (presented in next Section), but control of other Module types is also possible.

The Controllers are extensions of the Thread Module. The Up framework spawns one Controller per Recorder Module. The number of Recorders should be limited.

Recorder

Recorders are Modules designed exclusively to capture state of the relevant system component. For example the Load Guard Recorder will capture the CPU utilization. Recorders are used by the Controllers, but also other Modules might incorporate Recorder in their functionality.

Custom Modules

The preceding Sections present the built-in Modules. However, it is absolutely necessary to make possible the integration of custom build Modules. Custom Modules must extend any of the previously mentioned Module type and are registered during the system start up. Based on the type of the custom Module, appropriate actions are executed (such as starting the Started Modules etc.). The custom Modules used in the Raspilot are discussed in Section 3.3.

Summary

Figure 3.8 depicts the relations between the Module types.

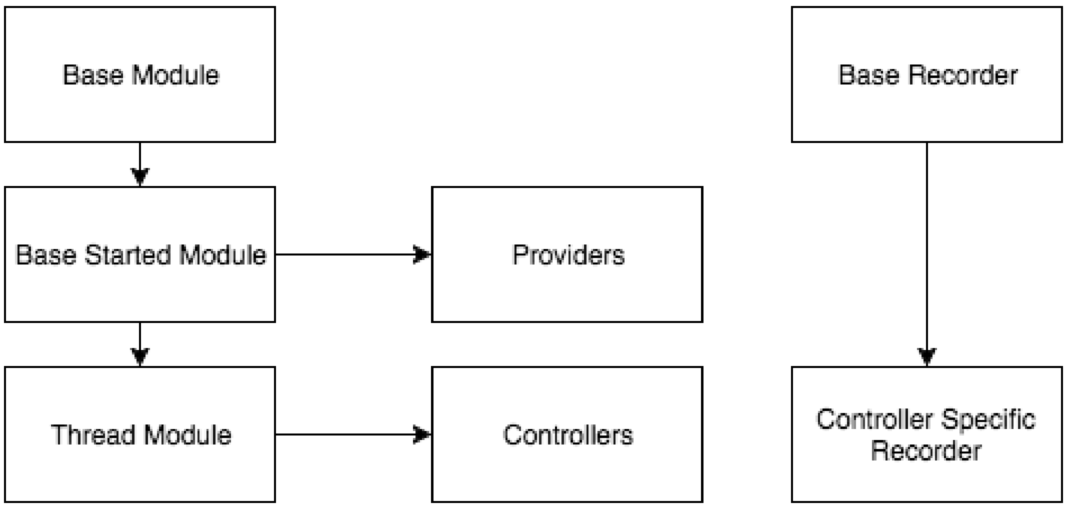


Figure 3.8: Relations Between Module Types

3.2.3 Flight Controller

As the name suggests the Flight Controller is responsible for the control of the flight. This includes the data readings, combining, processing and executing required actions. To do so the Flight Controller uses other components which has been previously discussed.

The data are read from various Providers. If the stabilization mode should be available at least Orientation Provider and RX Provider must be available. In case of the other flight modes such as return home, also Location Provider with GPS data must be available. Only then the navigation is possible.

Except for these, some other Providers might be required in custom implementations. The Figure 3.9 depicts the flow of the flight control process (the dashed Modules are required for some flight modes only).

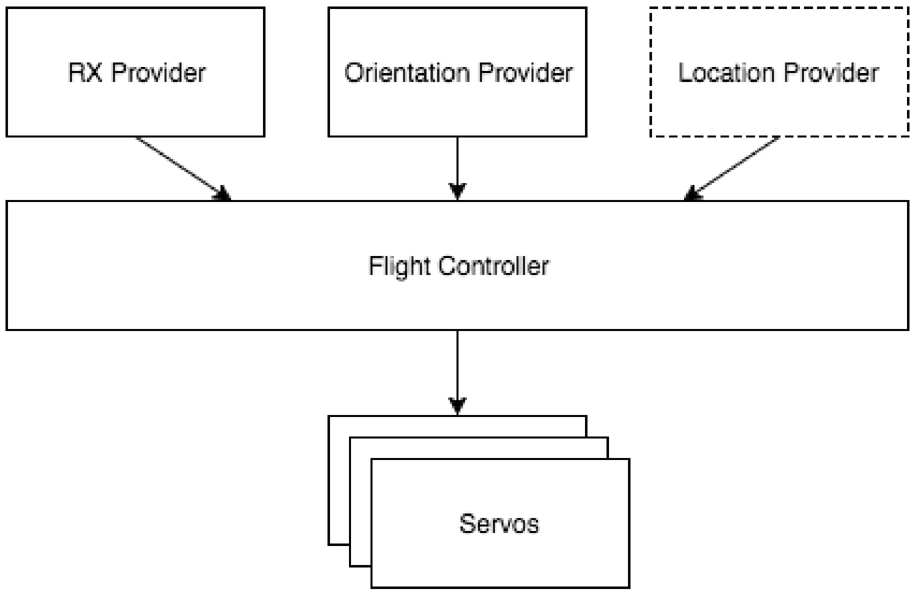


Figure 3.9: Flow of the Flight Control Process

As can been seen in the Figure, at first the data are integrated in the Flight

Controller which processes them and calculates the output which is sent to the servos.

3.2.4 Up Framework Architecture Summary

This Section states the reason, why this architecture reflects all of the requirements which were analyzed in Chapter 2.

The requirements for modularity and extensibility are satisfied with the built-in Modules and the possibility to implement custom Modules. Also the interoperability is satisfied with the ability to create custom Modules. The other non-functional requirements are more implementation related but the design supports them.

Each of the functional requirements are covered within the proposed design. Sensor and RX readings are implemented via the Providers, communication requirements are handled in the Mission Control Provider and Commands Module and the aircraft control ability is present in the Flight Controller. The Up framework design is clear enough to be expanded easily so other functionalities might be added in the future.

3.2.5 Panic Mode

Situations when the Raspberry is overloaded might have disastrous outcome. It is absolutely necessary to minimize possibility of occurrence of such situations.

Tests have shown that the main source of load is the communication. Data processing and calculations are far less resource demanding. This is the reason why we introduce the Panic Mode.

When the CPU utilization rises over the specified limit the Panic Mode is entered. The Command is automatically sent from Raspberry to Android and Arduino. After receiving this Command both devices **must** increase the delay between messages transmissions. The Command also states how long the delay should be. This ensures the decrease in the network traffic and the lowering of the CPU utilization. After the utilization drops below a specified limit, the Panic Mode is disabled, also by sending the Command. After disabling the Panic Mode the system is in a normal state with normal delays between messages transfers. Note that the peripheral devices (Android and Arduino) *must accept and follow* the specified delay. Only then the Raspberry can ensure the messages are received as fast as possible and they do not flood the system.

3.3 Raspberry Pi Application Architecture

The Raspilot on the Raspberry Pi adds some custom functionalities to the Up framework. The Figure 3.10 shows the components which are available in the Raspberry application.

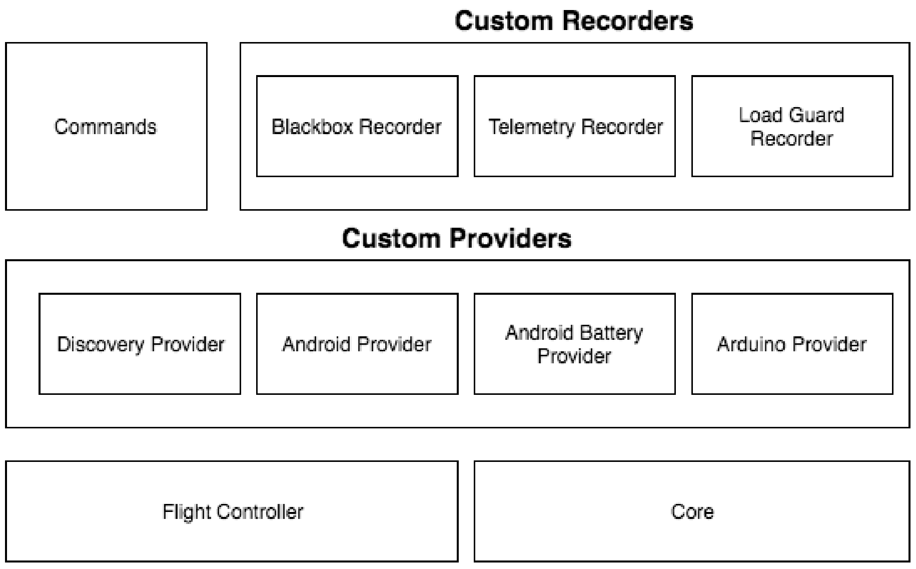


Figure 3.10: Raspberry Pi Application Architecture Overview

The architecture of the Raspberry Pi application is related to the High Level Architecture (see Figure 3.7) in the following way:

- *Custom Recorder, Custom Providers* - extends the Modules module
- *Commands* - extends the Commands module
- *Flight Controller* - the implementation of the abstract Flight Controller from the module Flight Controller
- *Core* - implements the abstract components of the Up framework

3.3.1 Core

Most of the components are implemented as an abstract class with abstract methods which must be overridden in the specific implementation. With this in mind we can say the core module contains all the descendants of the required abstract base classes.

3.3.2 Flight Controller

The Flight Controller is an abstract base class, as has been stated in the previous Section. The implementation of the flight control uses the PID Controller. We have decided to use this control algorithm because of its simplicity and wide usage amongst other autopilot systems. For more details about PID Controller see the Section 2.3.3. The Figure 3.11 shows the basic process of aircraft attitude control.

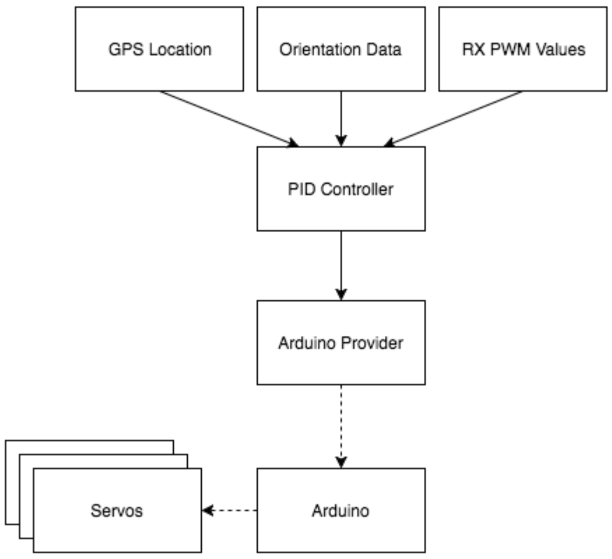


Figure 3.11: Aircrafts Attitude Control Process

The orientation data and RX PWM values serves as input values for the PID Controller. PID Controller calculates the output values which are then transmitted to Arduino which generates the corresponding PWM pulse for the servos.

The design of the Flight Controller was heavily influenced by the work of Dr. Gareth Owenson from the University of Portsmouth. His superb explanation on his web pages provides an excellent starting point [16].

His Flight Controller is based on the Flight Controller from the Ardupilot (see Section 2.2). His tutorial is for the multirotors and we will adapt it for the airplanes.

For each controlled channel the Flight Controller uses *two* PID Controllers. The first PID Controller is called the **Rate PID** and the second is called the **Stabilize PID**. We will start with explanation of the PID Controller in general, then we will explain the Rate PID and the Stabilize PID and as the last there is explanation of the third Navigation PID. The Navigation PID is added as a third controller to the basic Flight Controller from the tutorial.

PID Controller

Results of the PID Controller analysis have been stated in Section 2.3.3. For comprehension we state the basic principle of the PID Controller.

The PID (Proportional-Integral-Derivative) controller calculates the **output** value based on the **current** value (called process variable), **desired** value (called setpoint) and **previous errors**. It can be tuned with three parameters: K_p for the proportional part, K_d for the derivative part and K_i for the integral part.

The PID Controller is implemented as a library for all major languages (such as Java, Pyton, C/C++) and platforms (including Android, Arduino and Raspberry) therefore, we have decided to use those libraries. In case the Raspilot will be implemented in a language which does not have a PID Controller library, it must be adapted from other languages or designed and implemented from scratch.

Rate PID

Purpose of the Rate PID is to **rotate** the aircraft at *particular* rate based on the user input.

For example if we move the ailerons transmitter stick, the airplane starts to rotate. When we move the ailerons stick back to neutral position, the airplane *stops* the rotation but *will remain in position where we have left it*, it will *not* level with the horizon. This mode of control is also called **Acrobatic Mode**. In this mode the airplane is controlled **only** by the pilot.

To achieve such behavior the Flight Controller uses the gyroscopes and the input from the pilot. The Figure 3.12 depicts Rate PID inputs and output.

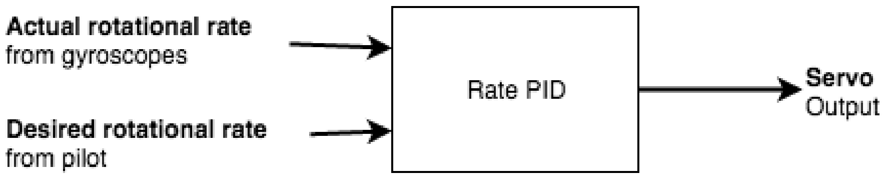


Figure 3.12: Rate PID

For example, if we want the aircraft to rotate at 15 deg/s and the aircraft is not rotating, we can model the situation as following:

$$\begin{aligned} error &= desired - actual \\ error &= 15 - 0 \\ error &= 15 \end{aligned}$$

The *error* is what the Rate PID calculates as a difference between setpoint (*desired*) and process variable (*actual*). Based on this and the previous errors the Rate PID then gives us output which is used to **alter the position** of the servos. If we want the Stabilize mode (sometimes called Fly By Wire) we need to integrate the Stabilize PID.

Stabilize PID

In the Acrobatic mode the position of the transmitter sticks controls the rotation rate of the aircraft. In the **Stabilize mode** the position of the transmitter sticks sets the **desired angle** of the airplane.

For example if we move the ailerons stick, the airplane adjusts its roll, but do not start rotating (it stops the rotation after the desired roll angle is achieved). When we move the ailerons stick back to neutral position, the aircraft *will level.* In this mode the airplane is **controlled by the pilot and the Flight Controller** as well.

To achieve the behavior of the Stabilize Mode, the Flight Controller uses the accelerometers, input from the pilot and the Rate PID. The Figure 3.13 depicts inputs and output of the Stabilize PID and the Rate PID.

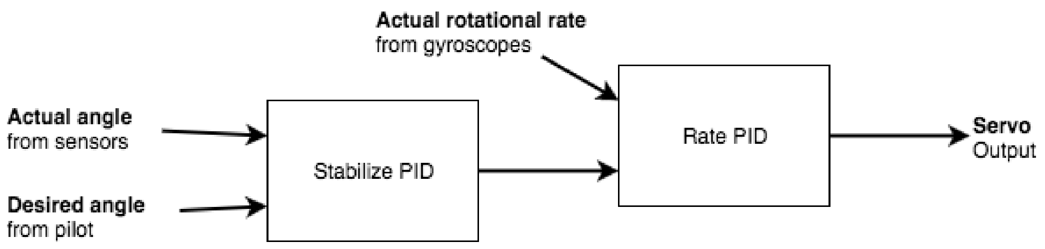


Figure 3.13: Stabilize PID

The output of the Stabilize PID gives us the **desired** rotation rate. This enters the Rate PID as a *setpoint*. With the actual rotation rate the Rate PID gives us output which is used to alter the position of the servos.

With this setup the Raspilot is able to level the aircraft and implement the Stabilize Mode.

Navigation PID

With previous two PID Controllers the Raspilot is able to level the aircraft. If we want the Raspilot to be able to navigate the aircraft, we have to add another PID Controller. This PID Controller takes as setpoint the **desired heading**. Current heading is the process variable. The output of the PID Controller is used to determine the required change of the aircraft's heading in order to reach the desired location.

The desired heading is calculated based on the **current GPS** location and the **destination GPS** location. Current heading is read from the magnetometer of the Android device. With these inputs the Navigation PID outputs the required change of the yaw angle. This enters the Stabilize PID. In this mode, which is called the Navigation Mode, the aircraft is controlled **only by the Flight Controller**.

The Figure 3.14 depicts the inputs and the output of the Navigation PID, Stabilize PID and the Rate PID.

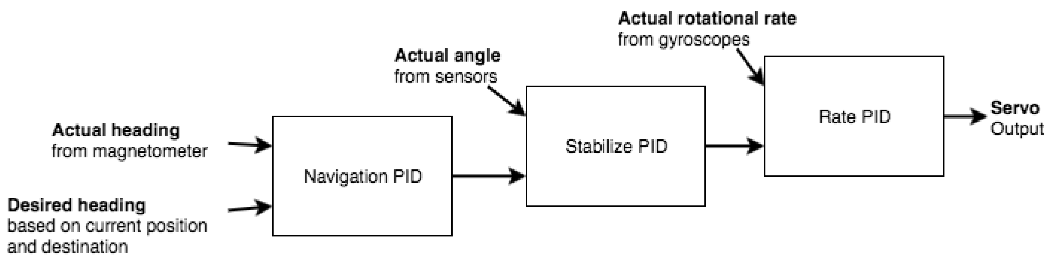


Figure 3.14: Navigation PID

3.3.3 Custom Providers

Besides the standard providers defined in the Up framework (see the Section 3.2.2 for more information) the implementation requires some custom Providers as well. The following Sections present them more in depth.

Discovery Provider

Raspberry Pi and Android are connected together via a TCP/IP network. The IP address of the Raspberry Pi is **dynamic**. Therefore a discovery mechanism must be available.

Android device *broadcasts* the *UDP* packet. The application on Raspberry will receive this packet and responds with message containing its address. This message is send via UDP *unicast* to the Android. The Android device now knows the IP address of the Raspberry and can connect via TCP.

The receive and reply part of this protocol are handled by the **Discovery Provider** on the side of the Raspberry. Note that the Discovery Provider is meaningless when the system is run in the Black Box mode because the address of the server is always known. Furthermore the UDP packets will not be functional because of the cellular network restrictions.

Android Provider

The Android Provider manages the communication channel with the Android device. This provider implements *General Transfer Message Protocol* and *Orientation Forward Protocol* as well. This is achieved by managing two sockets. The sockets might be TCP or UDP (Black Box mode requires TCP, Flight Controller mode *might* prefer UDP).

Let the socket implementing the Orientation Forward Protocol be named *Orientation Socket*. The other socket will be named *General Socket*.

Data received through Orientation Socket are forwarded to the Orientation Provider which processes them. In the Raspilot the communication between Orientation Provider and Android Provider is quite busy. Also the Orientation Provider depends on the Android Provider which serves as a data source in this case.

Messages received through the General Socket are forwarded to the Commands Receiver which processes them. For example the Location Provider registers a custom Command. The action of this Command invokes method of Location Provider which sets the received latitude, longitude and other data and makes them available to the rest of the system.

Android Battery Level Provider

Simple provider which serves the purpose to obtain the battery level of the Android device. This battery level is then present in the telemetry updates.

Arduino Provider

The Arduino Provider maintains the Serial communication channel with the Arduino. The communication protocol is discussed more in detail in Section 3.1.2.

RX Provider

As in case of the Location Provider the RX Provider is designed in a very similar fashion. Arduino Provider receives RX data and forwards them to the RX provider.

3.3.4 Custom Recorders

The Recorders in Up are also implemented as an abstract classes which must be sub-classed. The Raspilot incorporates three types of Recorders.

The **Black Box Recorder** and **Telemetry Recorder** are used to periodically record the state of the whole system. Data from the Black Box Recorder are being saved locally on the Raspberry while the data from Telemetry Recorder are transmitted via Mission Control Provider. These two are separated despite their very similar functionality. The telemetry may lack some information which is not so important (to save the network traffic) but on the other hand, the local storing of this information might be helpful for additional reviews.

Load Guard Recorder periodically checks the CPU utilization. If the CPU utilization is higher than the specified limit, the Panic Mode is enabled (for more information see Section 3.2.5).

3.3.5 Commands

Besides Commands defined by the Up framework the Raspilot adds some custom Commands.

Android Battery Command is sent from the on-board Android and contains information about current battery level of the device. This is necessary because if the Android battery will discharge, orientation data, GPS, network connection will be lost which is a fatal issue for the system. Action of this Command sets the value of the Android Battery Provider.

Location Update Command contains the data from GPS such as latitude, longitude and altitude.

System State Command holds the data about the system state such as the utilization, available providers, whether is Arduino connected etc.

The **Telemetry Update Command** which holds all of the telemetry data sent to the Ground Station. Update contains the following:

- orientation data
- location data
- current system status (utilization, battery level, whether Panic Mode is enabled etc.)

This Command is transmitted every 100ms, but the delay might be altered after testing. 100ms delay will be probably too long and will be lowered.

The **PID Tunings Command** which holds all of the PID tunings parameters is sent from the Ground Station. Command contains the following:

- K_p, K_i and K_d for Rate, Stabilize and Navigation PID for all controlled channels

3.3.6 Automated Load of the Modules

One of the features of the Up framework is the automated loading of Modules, Recorders and Flight Controller. During the startup of the Raspilot all Modules, Recorders and Flight Controller are automatically loaded integrated and registered within the system. To do so the directory structure must be defined. The following Figure illustrates the directory structure which must be followed:

```
<ROOT_FOLDER>
  flight_controller/   # contains the Flight Controller
  modules/             # contains custom Modules
  recorders/           # contains custom Recorders
  commands/            # contains custom Commands and
                       # Command Handlers

  ...other custom files and folders...
```

Figure 3.15: Folder Structure

Names of the folders wrapped in <> are not defined and can be set by user. However, the folders `modules`, `recorders` and `flight_controller` must have exactly these names as they are further processed by the Automated Load feature.

The Module is handled depending on the superclass which it extends (for example if the Module extends the Mission Control Provider it will be registered as Mission Control Provider).

The Automated Load feature simplifies the startup, Modules instantiation and registration process. This feature also makes the following possible: if a new Module is implemented there is no need to alter the rest of the system, the author just puts the Module in the `modules` folder and the Module is automatically integrated into the system.

This however implies some limitations. For example, all Modules must have unified constructors. But these are rather implementation problems. It is certainly possible to implement such feature because there are systems which use a kind of autoload[1]. On the other hand it brings some sort of organization to the system structure which increases the overall simplicity.

3.4 Server Application Architecture

One of the requirements was for the ability to run in the Black Box mode. The Black Box mode was presented more in detail in Section 2.4.7. The main idea of the Black Box mode is to reduce the number of on board components in exchange for the cost of reduced functionality. This is achieved by removing the Raspberry Pi and Arduino from the aircraft. Only the Android device is present on board in this mode.

The main purpose of the Up framework is to create autopilot solutions not telemetry systems. So this might look like a different problem which should not be considered when designing this framework. However, we consider the required

[1]For example the framework for creating the web applications - Ruby on Rails. This framework has very strictly defined folder structure and the models, controllers, view etc. are being searched for in the specified folder. If the user misplaced the source of some component, the system will not find it.

changes small and easy to implement. In fact the framework is not changed at all. The general idea is following: *run the same application which is run on the Raspberry but in some special mode.*

The Figure 3.16 depicts a high level view of the server application architecture.

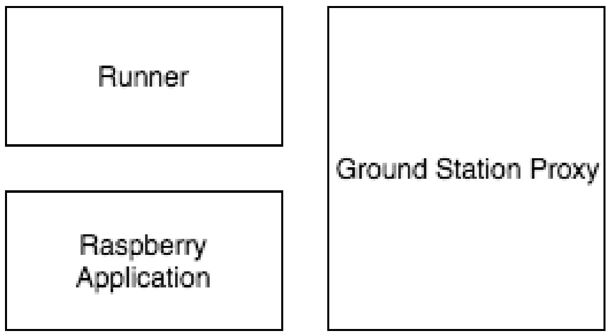


Figure 3.16: Overview of the Server Application Architecture

The Raspberry Application module contains the Raspberry Application - the Raspilot.

The Up framework itself was not designed to run for a very long periods of time [1]. For example the application produces large amount of logs (which are necessary because of testing, debugging etc.) which might be large enough to cause some sort of problems on the server. The design of the Up framework should not be changed, therefore the **Runner** component was created.

The function of the Runner is to spawn the application instances when requested. The process of application spawn is following:

1. the spawn request is received
2. if the application is already spawned, do nothing, just respond
3. if the application is not spawned, spawn the application and respond with the result of the spawn process
4. monitor the status of the application

Note that the Runner must be always listening for the spawn requests.

When the application should be started in Black Box mode, the airborne Android will at first send a spawn request and upon receiving the successful response, connects to the application. Besides the spawn process, everything works as in the Flight Controller mode. The only exception is that some of the Modules such as Arduino Provider (see 3.3.3) are not being instantiated.

3.5 Android Application Architecture

The Android Application is much more *constrained* than the Raspberry Pi Application, because it must comply with the architecture of the **Android framework**. On the other hand it should be as modifiable as possible. The following Sections will discuss the basic building blocks of the Android framework. Then the architecture of the application run on Android devices is presented.

[1] Orders of hours are not considered long periods of time, but weeks surely are.

3.5.1 Basic Components of the Android Framework

In this section we will discuss the basic components of the Android framework. The extensive analysis of the Android framework is not interesting in context of this thesis and will not be discussed here.

Activity

The Activity [17] is a single, focused thing that user can do. For example in our context, the screen with telemetry will be an Activity.

The Activity does not only render the user interface (UI) and interacts with the user, but can also start background services (see following Sections), manage connections to the database, etc. Everything what is done in the Activity is executed on the *UI thread*. This should be considered in case of blocking operations such as network communication.

Fragment

The Fragment [18] represents a certain functionality of the Activity. The Activity can contain more fragments. We can think of the Fragment as a Subactivity. In our context the Telemetry Activity might contain two Fragments: one which will display the position of the aircraft on a map, and second which will display a list with telemetry data.

The Fragment can also start background services, access database, etc. The advantage of the Fragments is the ability to *reuse* them.

For example a *small* mobile phone screen cannot display large map and the telemetry data at the same time. However, on the *larger* screen of a tablet this might be possible. Without fragments, we have to implement multiple activities, some for the smaller screens and some for the larger screens. If we use the Fragments, the change is done much more easily. The Activity on the smaller screen displays only one fragment at the time, while on the larger screen it displays them both.

As with the Activities, the Fragments also executes their methods on the *UI thread*.

Service

The Android framework offers multiple tools to implement the background tasks. Depending on what we need, we can use the **Service** [19] or **Intent Service** [20].

The Service does not have a Graphical User Interface and usually does not interact with the user. The Service is useful for example to obtain the GPS coordinates of the device, to read the sensors, communicate with remote server, etc. The Service also executes all its methods on the UI thread, however we can start a new thread[1] which can handle the blocking tasks.

The Service can be **bound** or **started**. The *bound* Service lives till at least one component (usually a Fragment or Activity, but also other Service) is bound. When the number of bound components reaches 0, the Service is destroyed. We

[1]The Activity can also start thread, it is more of a design choice to start them in dedicated Service than to start them in Activity.

can use this for example to receive the telemetry on the ground device. The Fragment which displays the telemetry bounds the telemetry Service when it is started, and unbounds when it is destroyed. It makes no sense to receive telemetry if there is nothing which will present it.

The started Service must be *explicitly* started and stopped. If we start the Service, it will live till it is explicitly stopped. This can be used for example for Service, which reads the sensors. We want to send the sensor data even if the application is in background, or the device is locked. The component which starts the Service (usually a Fragment or Activity) can differ from the component which stops the Service (the Service can also stops itself). The started Service can be also bound. In case of the started service even if the number of bound components reaches 0, the Service is not destroyed.

The **Intent Service** is a special case of a Service. We can think of it as a work queue processor. It receives the requests from other components (usually a Fragment or Activity) and processes them. All requests are handled in a **single worker thread**, which differs from the UI thread. The difference between Service and Intent Service is that the Intent Service executes its task, and dies. The Service lives till it is stopped or no one is bound. Therefore, the Intent Service is suitable for tasks such as downloading files, or for example spawning the Up Application on the remote server.

3.5.2 Up Library for Android

The Up Library for Android adapts several patters from the Up framework. The Commands and Commands Executor are present also in the Up Library. The pattern of Commands is used for communication between airborne devices and between the ground device and Raspberry as well.

Beside the Commands, we need to read the sensors, pass the sensor readings to Raspberry Application and execute Commands incoming from Raspberry Application.

To make the exchange of Commands between airborne devices possible we need the following:

- an Android **Service** which connects to the Raspberry Application
- the Android Provider (see Section 3.3.3) running on the Raspberry

With this setup we are able to exchange the Commands between the airborne Android device and Raspberry Pi. This is contained in the **Communication Module**, which administer this task. This module has also one another purpose. It handles the communication between the ground device and the Raspberry as well.

Sensor Readings

Let us consider a situation when we want to calculate the altitude from the barometric pressure sensor of the Android device. We need to read the data, possibly process them, and transmit them to the Raspberry Pi. This can be encapsulated into a single module, let us call it the Altitude Provider.

The flow is identical for all sensor types. What differs is the reading process, type of read data and how the raw data are being processed. Let us call the module which reads some sensor and transmits processed data the **Provider**.

We need to create a flexible way of registering and managing the Providers. This is contained in the **Providers Controller Module**.

The following figure depicts the overview of the architecture of the Up Library for Android:

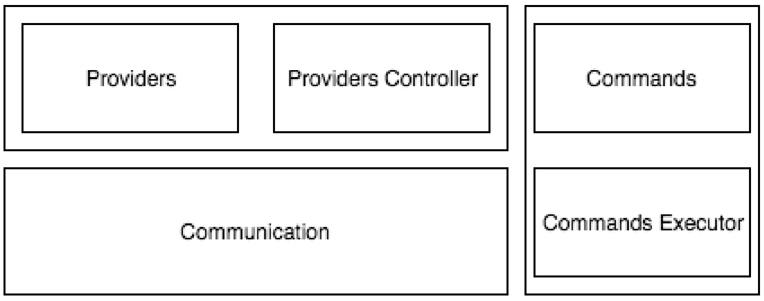


Figure 3.17: Overview of the Up Library for Android

3.6 Arduino Application Architecture

The Arduino Application should be also designed in such manner it can be modified easily. Many of the concepts are inspired by the design of the Up framework. The Figure 3.18 shows the overview of the Arduino application architecture.

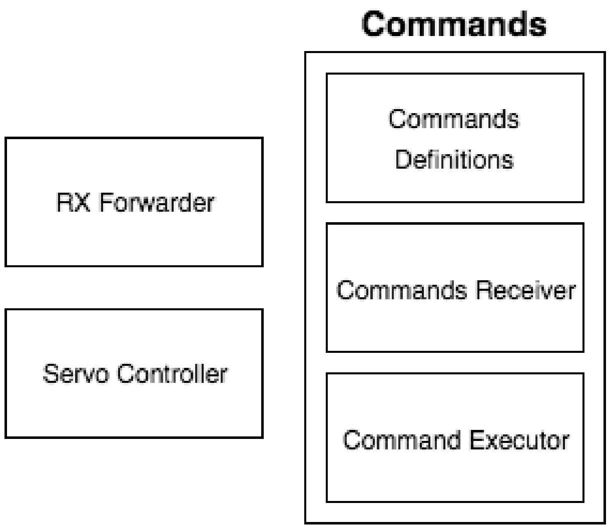


Figure 3.18: Overview of the Arduino Application Architecture

Commands

All of the three subcomponents have the same function as the same components in the Up framework. See Section 3.2 for more information.

Commands Definition component contains the definitions of recognized Commands, their payload size and action which should be executed after receiving of such Commands.

Commands Receiver is responsible for receiving, validation and correct identification of the transmitted Commands.

Available Commands

For clear idea we list some of the available Commands in the table 3.1.

Name	Payload Data	Effect	Target
Start	–	enables the application[1]	Arduino
Arm	–	enables the motors	Arduino
Disarm	–	disables the motors	Arduino
Panic	new delay	sets the delay between message transmissions	Arduino
Output	positions for 4 channels	sets the required positions	Arduino
Forward	PWM of all channels	transfers the data to RPi	Raspberry

Table 3.1: Commands Exchanged Between Arduino and Raspberry

Command Executer receives valid Commands from the Commands Receiver and executes them.

RX Forwarder

This component is responsible for reading the PWM (possibly otherwise encoded) information from the RX. This is the main reason why Arduino must be present in the system as the Raspberry cannot handle this responsibly. For more information about PWM pulses refer to Section 2.4.1. After reading all channels the data are transfered to the Raspberry via a Command.

Servo Controller

The responsibility of this module is to control the servos' positions and motors' speed as required. The input originates from the Flight Controller Module on Raspberry. After receiving the specific Command the corresponding PWM pulses are generated.

4. Implementation

This Chapter discusses the implementation of the all required software and is divided into *six* Sections which covers the following topics:

1. the Up framework implementation
2. the Raspilot application implementation
3. the Arduino application implementation
4. the Android application implementation
5. the Server application implementation
6. the hardware components

The Installation Guides for all components are available in the Attachment 1.

4.1 Up Framework

The Section *Up framework* discusses the implementation of the framework. It covers the following topics:

- the necessary technical details
- the differences between design and implementation
- the details of the process of system startup
- the details of communication between components are discussed
- the abstract Flight Controller

4.1.1 Technical Details

The Up framework is the core of the whole autopilot system. In the Section 2.4.4 we have discussed why the Up framework will be hosted on the Raspberry Pi. This has affected the choice of *the programming language* and other related *technologies* as well.

One of the purposes of this system is to be *extensible and modifiable.* Therefore the programming language should be well known. On the other hand the Raspberry Pi is powered by the processor with the **ARM architecture** [15]. Therefore, the language must have support for the ARM processors. The following choices are the most relevant:

- Java - very resource hungry, managed
- C/C++ - consumes less resources than Java, not managed
- Python - managed, adequately resource demanding

The Raspberry Pi itself is *short* on memory and we cannot afford to waste any. Therefore we have decided not to use Java. The following two choices are more of a personal choice. Because the time critical operations will be run on the Arduino, we have decided to select the Python.

4.1.2 Design Modifications

There are few design modifications which needs to be made.

The **Recorders and Controllers** have been dropped. The tests have shown they are in fact a sort of Thread Module. The design specifies, that the Controller will periodically invokes the Recorder. However, this is in fact exactly how Thread Modules operates. They spawns a thread in which a blocking or recurring tasks are being executed. Therefore the Recorders and Controllers have been *removed* in favor of the Thread Modules.

As a consequence of Recorders removal, the **telemetry** system has been also changed. The Up now contains a built-in Thread Module, which periodically asks all other Modules for their telemetry data. Then the telemetry data of all Modules is merged in transmitted to the Ground Station.

Each Module should return a dictionary with following format:

```
{
  "arduino": {
    "connected": True,
    ... other data...
  }
}
```

Figure 4.1: Module Telemetry Data Example

In this example the root object contains one key - `arduino`. Let us call this key the *module key*. This key will be merged into the resulting telemetry message. It is possible for more Modules to write their data under the same module key. For example if another Module responds with telemetry data with module key `arduino` and this key will hold a key `delay` with value 150, the resulting telemetry message will look as following:

```
{
  "arduino": {
    "connected": True,
    "delay": 150,
    ... other data...
  }
}
```

Figure 4.2: Resulting Telemetry with More Modules Under Same Module Key

The telemetry data from Modules are being deeply merged into the resulting telemetry message. In case of an conflict (there are two different values under same key, and this key is not a dictionary) an Exception is risen.

4.1.3 Startup and Initialization

The startup is quite complex process. The following must be executed:

- **discover** and **instantiate** all Modules
- **initialize** the Modules
- **startup** the Modules

Modules Discovery and Automated Load

The task to discover and load Modules is handled by the `UpLoader` class. Its responsibility is to **discover** relevant Modules, **instantiate** them and give them to the Up framework which will integrate them into the whole autopilot solution. This implies some requirements on the directory structure which has been discussed in Section 3.3.6. The UpLoader does the following:

- **scans** the files in the relevant directories
- **loads** the Modules from these files
- **creates instance of the Up** which can use the discovered Modules

Initialization and Startup

Once the Up Loader loads the Modules, the whole system enters the **initialization phase**. During this phase the Modules should execute required initialization tasks such as checking the environment (for example whether the Internet connection is available, obtaining reference to other Modules) etc. They should *not* carry out tasks such as opening the sockets. The initialization phase can be considered as a primary configuration phase.

Once the initialization phase is completed, the system enters the **startup phase**. During this phase the Modules are ordered to execute all necessary startup tasks such as *opening* the sockets, *establishing* connections etc. Each Module *must report* whether its startup has been successful or not and if not, whether the system should continue in the startup process. For example if the Arduino Provider which creates a bridge between the Raspberry Pi and Arduino, *fails* to start up, the system *might continue*, although in a Black Box mode. In Black Box mode the flight controlling abilities are not available and the system only serves for telemetry transmission purposes. If the Android Provider (similarly as Arduino Provider, it creates a bridge between the Raspberry Pi and Android) also fails to start, the system *should not continue.* It is already in the Black Box mode (due to the failed startup of the Arduino Provider) and without Android Provider there is no telemetry data source.

After the startup phase is completed the system enters the regular *run mode.* What tasks Modules perform during the phases is dependent on their purpose and implementation. Refer to the Section 4.2.3 for more information.

4.1.4 Intra-framework Communication

The intra-framework communication is very closely related to the **Commands** component from the design. See the Section 3.2.1 for more information.

We can describe the overall pattern of the intra-framework communication as following:

- read the data sources
- publish read data - **send them as a Command**
- **deliver** the Commands to the parts of the system which are interested in them
- let these parts carry out their tasks - **execute the Command**

After the previous statements it is of no surprise we have chosen the **publish-subscribe** model as the core model for the communication inside the framework.

For better understanding, an example of such communication pattern in context of Up follows:

- the Flight Controller **registers** for the GPS updates
- the Location Provider **receives** the new GPS location (from the Android device)
- after receiving the new GPS location the Location Provider **publishes** the location
- after publishing the GPS location the Flight Controller **receives the update** and **carry out** the requested operations (for example change the heading of the airplane)

Figure 4.3 depicts the previous example in form of a diagram.

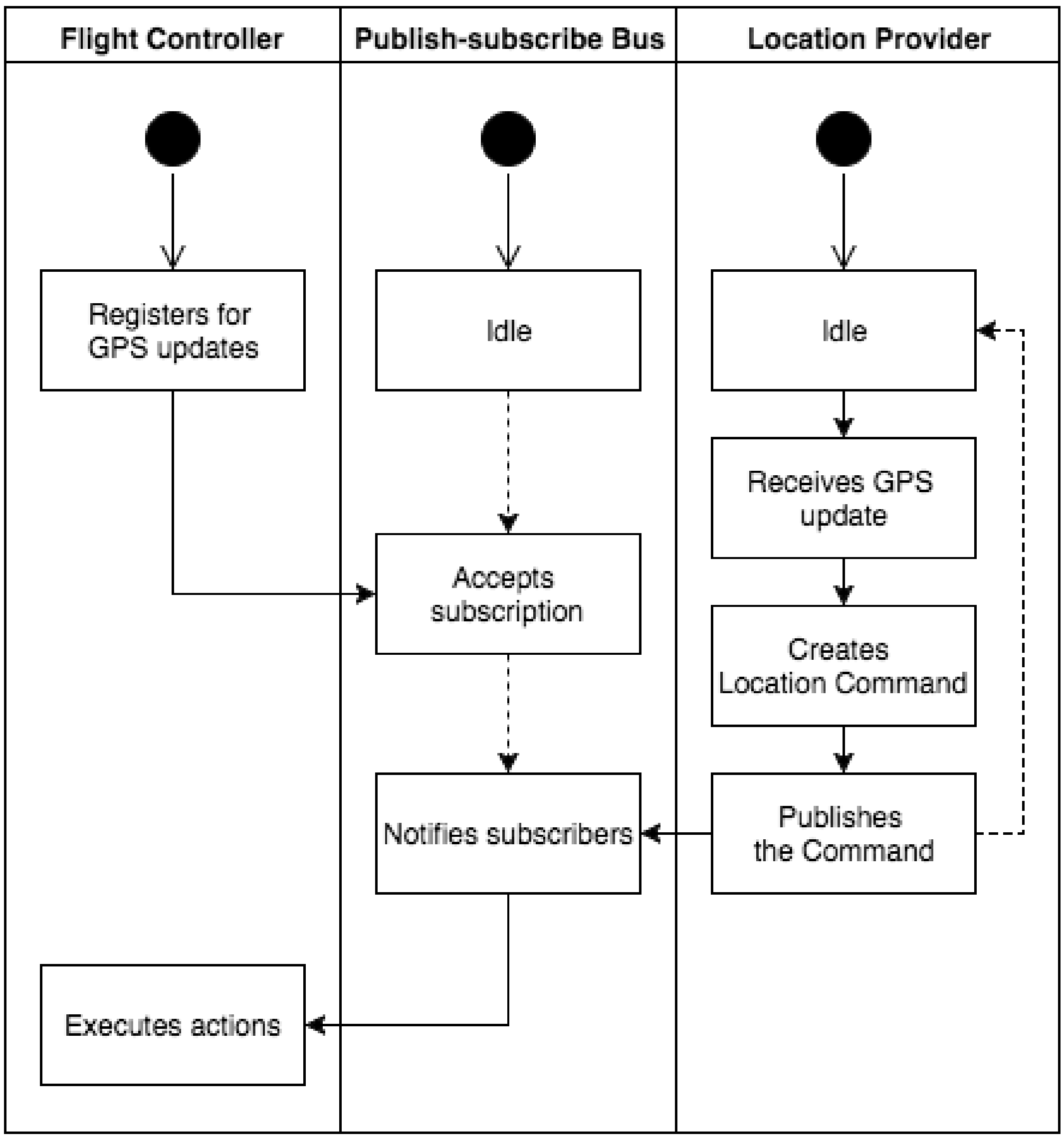


Figure 4.3: Intra-framework Communication Example

The previous example illustrates problems which must be solved:

- how to **subscribe** for particular updates
- how are the types of updates **distinguished**
- how to **execute** actions upon receiving of an update

The following Sections discuss these questions.

The Publish-Subscribe Bus

The idea of the publish subscribe bus is following: The senders of messages (publishers) sends the *identifiable*[1] messages to the bus. The receivers (subscribers) subscribe for particular message types and the bus is responsible for delivering all of the requested messages. The abstract idea of the publish-subscribe

[1]It must be possible to distinguish between different message types.

bus must be transformed into concrete implementation. This is handled by the `CommandReceiver` and `CommandExecutor` classes. Both of these are started during the start of the framework. Please not that the messages are called Commands in the Up.

Classes which have *direct control* over communication channels forward the messages to the Command Receiver. The Command Receiver can transform the data if necessary and invokes the Command Executor which executes the appropriate action. The idea of Command Receiver is to serve as a mediator. Currently this isnot used but might be in the future versions.

Subscribing and Publishing

All classes can subscribe and publish to the bus[1]. Usually the classes should subscribe during their *initialization*, but this is not mandatory. Classes subscribe for messages by specifying **the Command** and the **the action**.

Upon subscribing the *unique **Handler** identifier* is returned[2] and must be stored. Purpose of this identifier is explained in the following paragraphs.

The Commands are distinguished by unique **Command** idetifiers[3]. Specifying and setting the action is explained in the following paragraphs. Once the Command is received, the Command Receiver invokes the Command Executor which executes the particular action.

Setting the action is achieved by providing an *instance* of a **Command Handler** class which is *derived* from the `BaseCommandHandler` class (to ensure it has the required method). The Command Executor invokes the `run_action` method of this class. This method contains the parameter which holds the received Command. The method should execute all the actions required to execute the required Command.

It must be possible to register and unregister more subscribers for the given Command. Thus the **unique Handler identifier** is returned after the subscription to ensure the correctness of the unregister process. When unregistering, the class must provide the unique identifier issued during the subscription.

4.1.5 Telemetry

In order to transmit a telemetry update following tasks must be executed:

1. obtain the telemetry data of the aircraft (discussed in Section 4.1.2)
2. create a special Telemetry Update Command (see Table 4.1 for more information)
3. deliver this Command to the Ground Station

The Telemetry Controller Module which is responsible for obtaining the telemetry data invokes the `transmit_telemetry` method of the `Base Mission Control`

[1]In future versions a permissions system required.

[2]Up uses the UUID - Universally Unique Identifier as a unique Handler identifier. UUID generator is part of the Python.

[3]For example the identifier of the location update Command has identifier `up.location.update`.

`Module`. This Module manages the communication channel between the airborne and ground devices.

However, the Up does not specify how the data should be transferred, therefore it is up to the custom application to define how the data are delivered.

4.2 Raspberry Pi Application

In this Section we will discuss the details of the Raspberry Pi portion of the Raspilot and we will focus mainly on the details of implementation of the abstract parts of the Up framework as well as custom Modules, and Flight Controller.

4.2.1 Technical Details

Because the Raspberry Pi application is build on top of the Up framework it uses the same language as the framework which is the Python.

4.2.2 Design Modifications

As well as the Up, the Raspilot has also undergone some design changes.

The most significant one is related to the Flight Controller and the orientation source. The first versions uses the Android device as a source of the orientation data, which turns out to be not powerful enough. The later versions uses the dedicated hardware component (see the Section 4.4.5 for more information) connected to Arduino for this purpose.

Similarly the Flight Controller was initially present on the Raspberry Pi, but the latency between reading the orientation data, processing them and executing required actions was very long. Therefore, in the latest version, (in the time of writing this thesis) the Flight Controller is hosted on the Arduino. The reaction times to alter the attitude of the airplane are significantly lower.

On the other hand, the GPS must travel from the source (Android) via the Raspberry Pi to the Arduino in this setup. The frequency of the GPS updates is much slower than the frequency of orientation updates, therefore this flow is acceptable.

We consider the ability to move the functionality (such as the mentioned Flight Controller) freely as a proof of concept of the Up framework (more precisely of the ability of the framework to be extended and changed).

Another significant change are the plug-in modules which are discussed more in detail in the Section 4.3.

4.2.3 Modules

In this Sections the implementation of the custom Modules is discussed more in detail. Refer to the Section 3.3.3 and later for more information about design of these Modules.

Arduino Provider

The Arduino Provider is a **custom Thread Module**. This Module creates a bridge between the **Arduino and Raspberry**. One of the important responsibilities of this Module is *handling* the Serial line between Arduino and the Raspberry. All communication between these platforms *must* be transmitted via this provider (to avoid conflicts).

The Arduino Provider uses the *Command Receiver* and the *Command Executor* quite extensively. For example the Arduino Provider cooperates with the Orientation Provider in the following way:

- Arduino **reads the orientation** and **sends** it via the Serial line
- the Arduino **Provider reads the received data**
- based on the Command type (see 3.1.2 for more information) the Arduino **Provider distinguish between types** of received data and creates a Command from these data
- Arduino **Provider forwards the received Command** to the Command Receiver
- Command Receiver invokes the Command Executor and because the Orientation Provider **has registered** for this type of Commands, it **receives the Command** and can **update** the orientation

In the other direction, the other Modules uses the methods of the Arduino Provider to send the data to the Arduino. The Figure 4.4 depicts the flow of the orientation data from the Arduino the Orientation Provider.

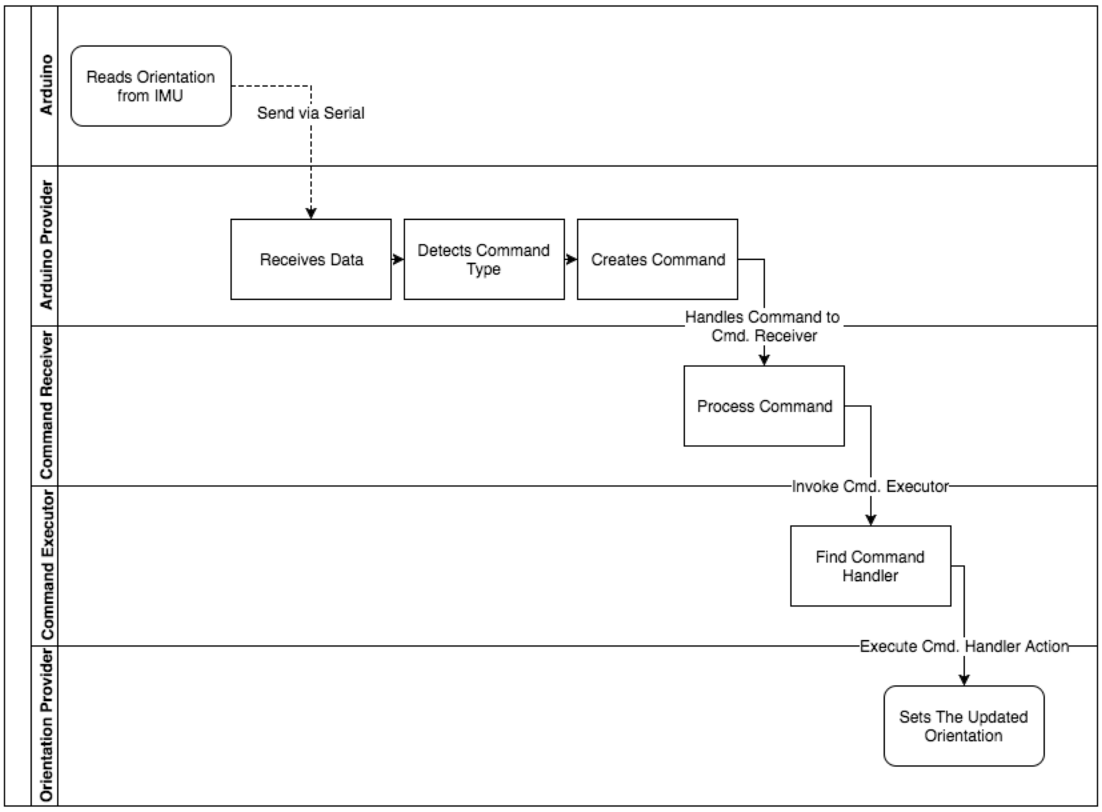


Figure 4.4: Flow of the Orientation Data From Arduino To Orientation Provider

For more information about processing the received data from the Raspberry Pi on the Arduino refer to the Section 4.4.3.

Android Provider

The Android Provider is quite similar to the Arduino Provider. Its purpose is to create a bridge, but in this case between the **Android and Raspberry**. As well as in case of the Arduino Provider it handles the connection channel and all communication between these platforms should go through this provider.

The Android and Raspberry are connected via the *USB Tethering* function of the Android devices. This function creates an TCP/IP network between the platforms. Therefore the network sockets are used for communication. This provider uses the **General Message Protocol** which has been presented in the Section 3.1.3 and the **Orientation Forward Protocol** as well (details about this protocol can be found in the Section 3.1.3).

The Android Provider tries to **restore the connection** if the connection has been lost. Connection drops are *not anticipated* when the Android and Raspberry are both on board, but in case of the Black Box mode the connection drops can be *quite frequent*. Especially in places with bad signal coverage.

Orientation and RX Providers

The Orientation Provider (see Section 3.2.2 for more information) makes the attitude of the airplane available to the rest of the framework. This provider went through many implementation changes. The Section 4.2.2 presents more details about the differences in this provider amongst the versions the Raspilot. The latest version (in the time of writing this thesis) uses the Android and Arduino Provider as a source of data.

If the Raspilot is in the **Black Box** mode then the orientation is read from the **Android** device via the Android Provider. The flow in this situation is following and uses the **Orientation Forward Protocol**:

1. **Android** reads the sensors
2. read data are sent to the **Raspilot** (run on a remote server) via the Internet
3. the **Android Provider** receives the data
4. special **Orientation Command** is created from the received data
5. the Command is **sent** via the Command Receiver and processed in Command Executor
6. **Orientation Provider** receives the new orientation (it is subscribed for the Orientation Update Command)
7. based on the received Command, the orientation in the Orientation Provider is set

This example shows an **important pattern** in the Raspilot: *one Provider can be source of the data for the another Provider*. In fact the Providers can be organized into a hierarchical structure.

If the Raspilot is in the **Flight Controller Mode** then the orientation is provided by the Arduino. The Android device does not produce orientation data in this mode. The flow is very similar to the previous one:

1. **Arduino reads** the orientation
2. orientation **data are sent** over the Serial line
3. **Arduino Provider receives** and processes the data
4. new Orientation **Command is created** and broadcasted

Here we can observe another possibility of the framework. Without changing anything in the Orientation Provider, the orientation data can originate from *multiple sources*. In fact it is possible to use the Android device in this mode as a **redundancy** component. However, the Raspilot currently does not utilize this.

The **RX Provider** is another example of the Provider which data source is another Provider. The RX Provider makes the RC receiver output available to the rest of the system. To do so the Arduino must read the PWM pulses, send them via the Serial line, the Arduino Provider reads and processes the data, creates a special Command and publishes this Command. RX Provider receives the Command and updates its data. For more information about how the PWM signal is read by the Arduino refer to the Section 4.4.4.

Mission Control Provider

The purpose of the Mission Control Provider is to create and manage communication channel between ground device and the Raspberry. Mission Control Provider connects to the proxy server via the Internet. This Provider utilizes the **General Message Transfer Protocol**. The Mission Control Provider is also responsible for transmitting of the telemetry updates and receiving Commands from the ground device. The Up framework contains abstract base class, the `BaseMissionControlProvider`, which should be extended when creating custom Mission Control Provider.

Other Custom Modules

In this Section the custom Modules are presented more in detail.

The Android Battery Provider serves as a source of information about the battery level in the Android device. This information is crucial because without the Android device, the Raspilot will **loose** the Internet connectivity or the telemetry data source (if the Raspilot is being run in Blackbox mode). Information about the battery level is also contained in the telemetry updates. Data are passed via the Android Provider similarly as in the case of the orientation data.

The Discovery Service is in fact also a Provider. Its purpose is to reply to discovery requests made by the Android device when connecting to the Raspilot in the Flight Controller Mode. Communication between this Provider and the Android devices is based on *broadcasting* of the *UDP* packets. Refer to the Section 3.3.3 for more information about design of this Module.

4.2.4 Commands and Command Handlers

The Commands and their Handlers are very important part of the Raspilot. As has been shown in the Section 4.2.3 the Commands does not serve only as a controlling mechanism but also as a mechanism of communication.

For each Command there must be a Command Handler. The Command Handler executes the action after the Command is received. For more information about the Commands, their Handlers and intra-framework communication refer to the Section 4.1.4.

The Table 4.1 lists the Commands which are recognized by the Raspilot:

Name	**Source**	**Action**
Android Battery	Android[1]	sets the current battery level for the Android Battery Provider
Location Update	Android[1]	sets the current GPS location for the Location Provider
Orientation	Arduino[1]	sets the current orientation for the Orientation Provider
Panic	Load Guard	enables or disables the Panic Mode
RX Update	Arduino[1]	sets the current PWM readings for the RX Provider
Telemetry Update	System State Recorder	used to transmit the telemetry
Telemetry Frequency	Mission Control	used to alter frequency of telemetry updates
PID Gains	Mission Control	used to alter the gains of PID Controllers (see Section 4.4.6 for more information)

Table 4.1: The Raspilot Commands

4.3 Plug-in Modules

During the development a simple problem has been encountered several times: *What should be a part of the framework and what should be left upon the custom applications?*

If we place *everything* in the Up, then the Raspilot will be very lightweight. On the other hand, the Up will become harder to maintain and will contain large

[1]If the source is noted as Arduino or Android in fact it means the Command originates form the Android or Arduino Provider which creates the Command based on the received data.

amount of code, which will be very specific and the majority of users will not use it. Such large framework will also require more hardware resources (such as CPU, RAM, disk space, etc.).

Therefore we have selected the following approach:

- the Up framework will be as **lightweight** as possible and will contain *only* functions which are anticipated to be used in majority of use cases
- the functions which are somehow *specific* (for example the communication over the Serial) but are anticipated to be used by many users, will be packed into the plug-in modules, which can be integrated into the Up framework
- *specific* functions which are not present in the plug-in modules will be implemented by the *custom* applications

In the Up framework the plug-in modules are being called the **Cogs**. This approach rises several questions which must be answered:

1. How to distribute and install the Cogs?
2. How to integrate the Cog into the Up framework?
3. How does the custom application specify which Cogs are required?
4. How to handle dependencies of Cogs?

The answers for these questions are presented in the next Sections.

4.3.1 Distribution and Installation of Cogs

The Up framework is written in Python. Python language itself has many libraries. These libraries can be installed in several ways. Most used ways are:

- installation from **source** using the Setuptools [21]
- installation from the **the Python Package Index** [22]

If properly designed, the Cog can be handled as an *ordinary Python library*. We could use the Python utilities and mechanisms of library handling, which includes package managers, dependencies handling, etc. The ways how Python processes the libraries are not important in context of this thesis and will not be discussed here.

4.3.2 Cogs Integration

Custom Modules uses the Automated Load feature to be discovered and loaded during the startup phase. However, the Automated Load *requires* a defined *directory structure*. Because Cogs are installed as libraries, the required directory structure cannot be ensured, and therefore the Automated Load cannot be used to discover the Modules defined in the Cog. Some other mechanism must be implemented.

The general idea of Cog integration is following:

- user **downloads** and **installs** the required Cog
- within the context of the custom application user **registers** the Cog
- from now on, the Cog is properly **integrated** into the Up, which means, all Modules will be discovered and loaded during startup, all classes can be instantiated, etc.

To meet these requirements, each Cog *must* fulfill the following:

- it **must** contain the `registered_modules.yml` file (discussed in the next paragraph), it specifies which Modules should be loaded by the Automated Load
- it **must** contain the `Registrant` class (discussed later)
- it *may* define custom Modules, Commands, Handlers, etc.

The Cog must contain an enumeration of Modules which should be loaded with the Automated Load. To create this we can use the Automated Load.

The Automated Load must at first **discover** all the Modules which are then processed. The output of the Modules discovery can be used to create the required enumeration. This enumeration is then written into the `registered_modules.yml` file and packed into the Cog's distribution. This file is then later used by the Registrant class during Cog integration.

Now we will discuss how to integrate the Cog into any custom application which uses the Up. This uses the previously mentioned `Registrant` class and the `registered_modules.yml` file. The Registrant is responsible for:

- integrating all required Modules within the context of the custom application (so they will be discovered and loaded during startup)
- creating all required configuration and data files which the Cog requires

When the user wishes to integrate a Cog into custom application, the following flow is applicable:

- user specifies the **name** of the Cog and runs the **Cog Integration Utilities**[1]
- the Cog Integration Utilities **instantiate** and invoke the Registrant class of the Cog
- the Registrant class **writes** the Modules which should be loaded by the Automated Load into the proper configuration file of the custom application - `external_modules.yml`
- the Registrant class **creates** all the **configuration files**, data files, etc. required by the Cog

[1]Cog Integration Utilities are a set of utilities which eases the process of Cog integration, installation, etc.

After these steps, the Cog is *properly integrated* and the custom application can use it.

To make possible for the Cog Integration Utilities to instantiate the Registrant it must be present in **the root Python module** of the Cog. For example if we have a Cog with name *Foo*, then it must have the following structure:

```
foo_cog/
    __init__.py
    registrar.py
    registered_modules.yml
    modules/
    commands/
    ...
```

The `registrar.py` file contains a definition of a valid Python class `Registrant`. This class extends the `UpRegistrant` base class which defines the required methods. These requirements make it possible to load the Registrant class from any custom created Cog.

Functions of the Registrant Classes

The Registrant reads the packed `required_modules.yml` file (containing enumeration of Modules which should be automatically loaded) and writes its content (in proper format) into the `external_modules.yml` file (present in the root of the *custom application*) and creates the required configuration scripts, data files, etc[1].

The `external_modules.yml` file specifies *all* Modules from *all* Cogs required by the application which should be loaded by the Automated Load. This file should not be edited by users. The answer for question *How to specify required Cogs?* is presented in the next paragraphs.

The Automated Load reads the `external_modules.yml` file and loads all of the Modules specified in it. This enables the Up to integrate and load the Modules of any custom build Cog.

Specifying Required Cogs

The last question which must be answered is how to specify Cogs which the custom application depends on. The `external_modules.yml` file contains Modules which should be loaded by the Automated Load. However, the Up anticipates that all of this Modules *can* be imported. If the Module cannot be imported (for example the Cogs which contains the Module is not installed), it results in an exception. For that reason we have decided to install the `Cogfile.yml` file.

The `Cogfile.yml` file specifies which Cogs are *required* by the application. User can run the Cog Integration Utilities which will process this file and **install** all of the missing Cogs.

With this approach we can solve situations like:

- install custom application with all required Cogs

[1]One can think of the Registrant as a post-installation script.

- the application adds/removes some Cog dependencies (newer version of the application updates the Cogfile, user runs the Cog Integration Utilities which will install and integrate the missing cogs)

One can think of the Cogfile as a dependency specification of the application. The Raspilot also uses this approach.

4.3.3 Cogs Dependencies

The Cogs themselves can have dependencies. The dependency resolution is quite *complex* task. Luckily the Python *contains* utilities which can solve this. It will be redundant to create another package manager, if Python already contains one.

The Cogs can specify the dependencies as any other Python library would. Python will gather and install all the required dependencies during installation process, as it would do with any other library. This means that the Cogs can depend on some other Cogs, other Python libraries, etc.

4.3.4 Existing Cogs

The Table 4.2 contains Cogs which are created by extraction from the original Up and Raspilot projects.

Name	**Purpose**
Android Cog	encapsulates the communication between Raspberry and airborne Android, contains Modules for GPS location and battery level forwarding
Arduino Cog	contains Modules required for communication with Arduino and forwarding of orientation, RX PWM values, heading, etc.; depends on the Serial Cog
Discovery Cog	implements the Discovery Provider
Load Guard Cog	implements the Load Guard, which checks the utilization of Raspberry and handles the Panic Mode
Mission Control Cog	implements the communication with the ground station
Serial Cog	implements and manages the Serial communication protocol, used by the Arduino Cog for example

Table 4.2: Cogs Extracted from Original Up and Raspilot

4.3.5 Summary

The addition of the Cogs was quite significant change. However, it was fairly easily integrated into architecture and implemented. We find this to be another evidence of the extensibility and modifiability of the Up framework.

For comprehension we state a simple overview of the most important classes and files related to Cogs in Tables 4.3 and 4.4:

Name	**Placement**	**Content**
`registered_modules.yml`	Cog	specifies Modules which should be integrated from **this** Cog
`external_modules.yml`	Custom Application	specifies Modules which should be loaded from **all** Cogs
`Cogfile.yml`	Custom Application	specifies Cogs which the application **depends** on

Table 4.3: Most Important Cog and Cogs-related Files

Name	**Purpose**
`Registrant`	writes the required Modules to `external_modules.yml`, creates configuration, data files, etc.
`UpRegistrant`	the base class for all Registrants, contains methods for creating configuration files, writing to `external_modules.yml`, etc.
`Cog Integration Utilities`	the set of classes which ease the process of generating the `registered_modules.yml`, integration and installation of Cogs

Table 4.4: Most Important Cog and Cogs-related Classes

4.4 Arduino Application

This Sections explains the implementation of the Arduino application, how are the Commands between Raspberry and Arduino being handled, how is the PWM input read, how are the servos controlled and how is the Flight Controller implemented.

4.4.1 Technical Details

Most of the Arduino Boards are build on top of the ATMega microcontroller [12], [23]. More information about Arduino are stated in the Section 2.4.3. There is and IDE available for developing Arduino applications[24]. This IDE eases the whole process of the development including compiling, building and uploading of the application. We have therefore decided to use this IDE which *implies* the usage of the C/C++ language. Most of the operations executed by the Arduino are time critical which must have been taken into account when selecting appropriate algorithms to implement the particular requirement.

The Arduino application consists of three main parts:

1. **Setup** Routine

2. **Loop** Routine
3. **Interrupt Service Routine** (abbreviated as ISR), sometimes called Interrupt Handler

Setup Routine

The Setup Routine is used to **initialize** the whole application. Tasks such as *initializing the Serial communication*, *I^2C communication*, *attaching interrupts*[1] are executed here. This routine is called **only once** during the startup.

Loop Routine

Unlike the Setup Routine, the Loop Routine is called *over and over*. Usually most of the **application logic** is implemented in this routine. The Raspilot executes the following in the Loop Routine:

- reads the RC receiver outputs (refer to Section 4.4.4)
- reads the orientation data from the IMU (refer to Section 4.4.5)
- reads and handle the incoming data from the Serial (refer to Section 4.4.3)
- invokes the Flight Controller (refer to Section 4.4.6)
- sends required data to the Raspberry

Interrupt Service Routine

The Arduino enables us to attach interrupt to various pins. We specify the pin, the Interrupt Service Routine (ISR) and when the ISR should be invoked (possibilities are when the voltage of the pin is falling, raising or changing). The system then invokes the specified routine when the specified change on the specified pin is detected. Raspilot utilizes this mechanism for measuring the PWM output of the RC receiver.

There are several **limitations** imposed on the ISR [25, 28]:

- they **cannot have parameters** and should not return anything
- they **should be** as **fast** as possible
- if an ISR should be invoked while other ISR is being executed, the second will be executed after the current one finishes

Because the ISR should not return anything, we have to use *shared variables.*

[1]Attaching an interrupt is a process, when in case of the change of the voltage of particular pin, the system executes specified method. This is used for example to measure the width of the PWM pulse.

Shared Variables

When the main program wants to read the global variable (variable used by the main program and ISR as well) the ISR might be **invoked** during the read. This can lead to **malformed data**.

For example we want to assign the value of shared global variable (for example a float, which is 32bit in size) to a local variable. But after assigning the first 16 bits, the ISR is fired and the shared global variable is changed. Then the main program resumes and copies the remaining 16 bits. The **first 16 bits are from the previous value** and **the second 16 bits are from the current value** which is certainly **not desired**. Therefore, when we want to access the shared variable we have to turn off the interrupts, copy the variables and turn them back on. We can think of this as of a **critical section**.

However, when the interrupts are turned off, the ISRs are not being invoked even if the pin state changes, which leads to missing some changes. There is no other way how to implement this behavior and the possibly missed pin state changes must be taken into account. This is a significant problem when reading the PWM pulse. The Section 4.4.4 explains how the Raspilot solves these difficulties.

4.4.2 Arduino Model

Arduino comes in many variations[23]. The following is required to run the Raspilot:

- at least **6 pins** where the **pin change interrupt** can be attached (for reading the PWM output of the receiver)
- at least **4 pins** which can **generate the PWM** output (to control the servos and ESCs)
- at least **1 pin** where the **external interrupt** can be attached (required by the IMU unit)
- its size and power consumption should be as low as possible

All of these requirements are satisfied by the Arduino Nano [26] and the Arduino Uno [27]. Both these boards are very similar in terms of hardware and capabilities.

Arduino Uno is very widely used board, which determines that there are many libraries for this board. Arduino Nano shares the same microcontroller, therefore the libraries created for Arduino Uno can be used on Nano as well. The advantage of the Nano over the Uno is its much smaller size. We have therefore decided to use Arduino Nano.

4.4.3 Commands, Command Receiver and Command Executor

Arduino needs to *communicate* with its surroundings, more precisely with the Raspberry Pi. The Section 3.1.2 specifies the communication protocol used between the Arduino and Raspberry Pi.

Similarly to Up framework, the Raspilot application for Arduino also contains Commands and Command Handlers.

Commands are being distinguished by the Command type, and after the Command Receiver receives the particular Command, it notifies Command Executor which invokes the specified Command Handler.

The Command Receiver has two states:

1. **awaiting Command**

2. **receiving Command payload**

The Command Receiver starts in the **first** state. During each run of the Loop Routine, the Command Receiver checks (if it is in the *awaiting Command* state), if there are any data available. If yes, then it reads the **first byte**, which determines the **type of the Command** and checks if the Command type is valid (whether there is a Handler specified for this type). If the Command type is **not recognized**, the Receiver **discards** the byte and continues to the next available byte.

If the Command **type has a Handler**, the Receiver enters the **receiving Command payload** state and invokes the Executor. The Command Executor invokes the required Command Handler. The Command Handler at first checks if there are all **required data available**. If some data are missing (they might be not transferred via Serial yet) the Command Handler **notifies** the Command Executor that the action **has not been carried out** yet.

On the next invocation of the Loop Routine, the Command Receiver is in the receiving Command payload state and directly invokes the Command Executor which will again invoke the Command Handler. Now there are **all data available**, the action is **carried out**, the Command Handler notifies the Command Executor that the action has been carried out, the Command Executor notifies the Command Receiver which enters the **awaiting Command state**. This pattern is analogous to all subsequent invocations of the Loop Routine.

The Figure 4.5 depicts the handling of Commands. The Figure contains also the timeout mechanism, which is explained in the next Section.

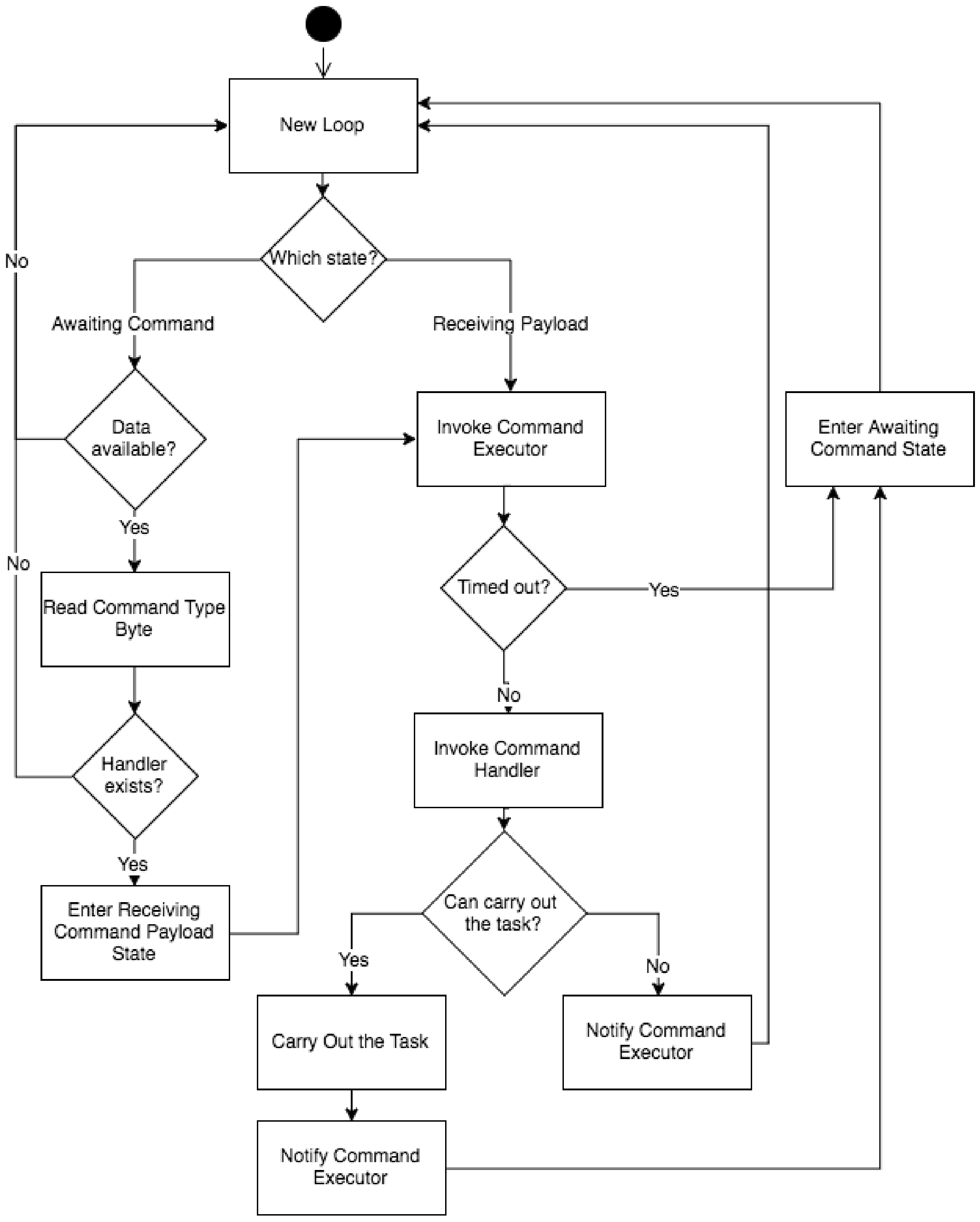


Figure 4.5: Arduino Command Handling

Handling defective payload

Because of the **unreliable** nature of the Serial communication there are couple of problems to solve:

1. the Raspberry Pi application might **crash** while sending data to Arduino
2. the data from the Raspberry Pi might be **lost** (particular bytes or whole group of bytes which compose the Command or its payload)

With the proposed patter of handling communication both of these situations will result in **infinite receiving** of the payload. In the first case it *might be* not crucial (because the Raspberry is crashed and will not send any other data).

But in the second case the Arduino will **stop responding** to all of the following Commands sent by the Raspberry. This is the reason why the Command Executor includes a **timeout** mechanism.

If the Command Handler does not carry out the task in specified time interval, the Command Executor **supposes data loss** and notifies the Command Receiver which enters the **awaiting Command state**.

However, there might be some data in the buffers which are **part of the malformed payload**. We can add special **marker** bytes to identify the start and end of the message. This will imply the need of **escaping** the bytes in the message (to avoid having the marker byte in the payload) and there is still possibility that the marker bytes will be lost. This approach solves the problem only *partially*, more precisely the probability of misinterpreting the data is reduced. On the other hand the communication will require more computational power.

The experiments has shown that when sending data between Arduino and Raspberry if something is lost, it is the **whole message**, not particular bytes from the message. Both of Arduino and Raspberry sends at first the Command type byte and then the payload, which might have more bytes. The payload is send via one method call. If the transmission fails, it will *most likely* will fail when transmitting the *first byte*. That might be caused by disconnection of the receiver, internal error in the transmitter, etc. We have executed many tests to analyze this situations and based on these tests we have concluded that the following pattern how to handle missing payload is applicable.

Upon **timeout** detection by the Command Executor, the Command Receiver enters the **awaiting Command state** and treats the first byte of the buffered data (if any) as a **new Command type** byte. There is still a possibility that this byte is from malformed payload of the previous Command. However, experiments have shown that the possibility of such situation is acceptably small. If this situation becomes a relevant issue of the communication other methods must be analyzed and implemented (such as acknowledgments).

4.4.4 PWM Reader

The PWM reader is responsible for reading the PWM output of the RC receiver. More details about the PWM pulse are stated in the Section 2.4.1.

During the development the PWM readings cause many difficulties, mainly because of the need of *very* precise time measurment. The Arduino (in the basic setup) can measure time with the **precision of** 4 µs. Therefore, if the RC receiver outputs a stable PWM pulse with the width of 1500 µs the Arduino reads a value between 1496 µs and 1504 µs. Because of the problems with the ISR stated in the Section 4.4.1 sometimes the readings are off by more than just 4 µs.

We have anticipated that the servos have precision of 10 µs (only changes greater or equal to 10 µs causes the movement of the servo). This is an example of the faulty anticipation. We have used the *cheaper* servos in the initial phases of the project and we did not await that such characteristic as PWM frequency precision differs between particular types of servos. Also some other solutions of the PWM generation for the servos have a step of 10 µs which only confirms our expectations related to required precision. However, the more capable servos have much higher precision.

The inaccuracy of the Arduino with the combination of the capable servos leads to **unacceptable servo jitter**. The jitter causes unwanted **power consumption** but also **impairs the hardware** of the servos. There are two options how to solve this problem:

- **increase the accuracy** of the Arduino
- **filter** the readings

The first solution requires either a **dedicated** hardware timer or modifications of the Arduino and then there are still the problematic readings of the shared variables.

The second solution on the other hand **adds only the filtering** of the read data. We have decided to select the second solution which should be sufficient for our case.

The Raspilot can read **6 channels**. **4** are used **to control the airplane** and the remaining **2** are used **to control the Raspilot** itself (for example change of the flight mode). For each channel we have the ISR, one *variable A* (used only by the ISR) and one *shared variable B*. Besides these there is a shared flag used to determine whether there are new readings available. All of the ISRs do the following:

- if the pin state is **high** (there is at least 3.3 V) the Raspilot **remembers current time** to the global **variable A**
- otherwise (there is 0 V on the pin) the Raspilot **subtracts** the global **variable A** from the **current time**, **stores the result** to the shared variable B and **sets the flag**

This gives us the **time** for which the pin state **had been high**, which is what we wanted to measure. However, it is not practical to access the global variables, therefore the Raspilot has to do the following in each execution of the Loop Routine:

- if the flag is not set, do nothing
- otherwise:
  - disable the interrupts
  - copy the shared variables to not shared variables
  - enable the interrupts

After this operation we have the read PWM frequencies available in variable which cannot be changed by the ISR and therefore can **be securely read**.

Low-pass Filter

The filter must be easy to calculate because it will be calculated after each PWM reading and we want to minimize the latency between reading PWM and executing action (setting position of servos).

We can conclude that the noise is high frequency noise. To remove the high frequencies from the signal we can use the Low Pass Filter [29]. This type of filter has many applications and one of them is smoothing the input data which is exactly what we need.

The simplest formula which implements the idea of the Low Pass Filter is following:

$$y[i] = y[i-1] + \alpha \cdot (x[i-1] - y[i-1])$$ [29]

Where *y[i]* is filtered output of the i-th sample and *x[i]* is the i-th sample. The α constant is used to tune the filter and will be explained in the next paragraphs.

We can see that this formula requires only addition and multiplication which are quite fast operations. We can transform the input of this formula so we can use integers instead of floats, but this should not be necessary.

With the α constant we can tune the filter. If we want the filter to make the data smoother, we set low α, but with low α the output will react slower. That means if the user changes the PWM **very fast**, the output will change **much slower**. This in fact creates **latency** between user input and positioning servo, which can have fatal consequences[1]. Therefore the α must be tuned precisely.

But the tuning of the filter proves itself not powerful enough. On one hand we have **stable PWM input** and on the other hand there are **fast responses**. We did not manage to tune the parameter to meet our requirements therefore, the following mechanism has been introduced: if the PWM changes **rapidly** (more than a specified constant) we **disable** the Low Pass Filter and use the **raw value**. This means if the user suddenly pulls the elevator, the PWM of the elevator channel changes dramatically. The inaccuracy of the Arduino is however relatively small (compared to the change) so it can be ignored in this case.

4.4.5 Determining Orientation

Unlike the initial versions of the Raspilot the current version (in time of writing this thesis) uses the **dedicated hardware** to determine the orientation. Because we do not have any special requirements on the hardware, we decided to use the mostly used one by the rest of the Arduino community, which is the *MPU6050 Inertial Measurement Unit* [30, 31] (abbreviated as IMU).

The IMU gives us access to all required data such as **rotation** and **rotation rate** around three axes. As has been stated in the Section 4.2.2 this change influenced the decision to implement the Flight Controller on Arduino instead of the Raspberry. There are more details about the Flight Controller in the Section 4.4.6.

With this change the Arduino is required to forward the orientation to the Raspberry because these data are required for the telemetry.

MPU6050

The MPU6050 has tri-axis gyroscope and accelerometer. The data from this IMU can be read via the I^2C (refer to the Section 2.3.3). The orientation can be read in form of the rotation matrix, quaternion or Euler Angles [32]. The Raspilot uses the Euler Angles format because this representation is more human readable than the others, therefore it is more easy do debug.

[1]If the airplane is heading towards ground, and you pull the elevator the airplane must react as fast as possible. The delay can determine if the airplane will hit the ground or not.

4.4.6 Flight Controller

When enabled, the Flight Controller is responsible for keeping the aircraft in the air and also for the navigation. To do so it utilizes the readings of the **PWM** output from the receiver and **orientation** from the IMU. The **GPS location** is used to execute the navigation tasks such as navigating to a waypoint.

The Table 4.5 describes available modes in which the Flight Controller can operate as well as required data for the controller to be operable in these modes.

Mode	**Functionality**	**Required Data**
Acrobatic Mode (ACRO)	Pilot directly controls the aircraft.	PWM
Stabilize Mode, Fly By Wire (FBW)	Stabilizes the attitude of the aircraft, pilot can control the airplane.	PWM, orientation

Table 4.5: Available Flight Controller Modes

Absence of the Return Home

These two modes are proof of concept, that the Up framework and Raspilot is in fact *able* to control the attitude of the airplane. The Return Home flight mode[1] is missing in the Raspilot because of the following reasons:

- the Return Home is an extension of the Stabilize Mode
- reliable Return Home is a question of good *configuration* if the Stabilize Mode is implemented correctly, it is not much interesting in terms of design and implementation
- the Return Home is much less computationally complex than the Stabilize Mode

To support the previous statements let us consider a following situation. In the Stabilize Mode, if the user have all transmitter sticks in neutral positions, the aircraft should be leveled. Therefore, the *rate of change* around all three axes should be *0*. Also the yaw, pitch and roll *angles* should equal *0*.

The idea of Return Home is following. Based on the *current* GPS location and the GPS location of the *take off point* the required bearing is calculated. The actual bearing and the required bearing are then used with conjunction of the Navigation PID to calculate the required rate of change in the yaw angle. This enters the Stabilize PID.

Now let us comprehend the previous paragraphs. In the Stabilize Mode, the required rate of change is calculated based on the user input. In the Return Home this is calculated from the actual and required bearing of the aircraft. The process of attitude control is otherwise identical. The Return Home feature must

[1]In the Return Home Mode, the aircraft returns to the take off location and pivots around this location in specified altitude.

calculate the required bearing after each update of the GPS position. These updates are however much less frequent than the updates of the orientation - it is much less computationally complex. To achieve a reliable Return Home we must have a reliable Stabilize Mode and a well tuned Navigation PID - the Return Home is a matter of configuration.

The GPS updates frequency might be a problem if it is too low. However, the result of such issue will be following. The airplane follows the previously calculated course till new course is available. This could lead to jagged flight path. However, this is an extreme case. All todays Android devices should be possible to navigate the aircraft. As a proof of this we can consider the existence of the car navigation systems which are present on these devices.

We have decided not to implement the Return Home feature also because of the already large scope of the thesis.

Implementation of the Flight Controller

Based on the design (see Section 3.2.3) there are *3* PID Controllers per *controlled* channel. Those 3 PID Controllers are Rate, Stabilize and Navigation PID. The Raspilot is capable of controlling 3 channels. Because the Raspilot does not support the Navigation PID it contains *6* PID Controllers altogether. The Table 4.6 presents which PID Controllers are enabled in particular mode and which are disabled.

Mode	**Rate**	**Stabilize**
ACRO	×	×
FBW	✓	✓

Table 4.6: Enabled PID Controllers Based on the Flight Mode

The Rate Mode might use the Rate PID in the future. We have decided to implement the Rate Mode as a forwarding of read PWM values without changes. This should ease the testing. If something goes wrong, the autopilot can be completely disengaged which should reduce the possibility of crashes.

Tuning of the PID Controllers

The parameters of PID Controller (also called gains) must be tuned. PID Controller has **three** parameters: K_p for the proportional part, K_d for the derivative part and K_i for the integral part. See section 2.3.3 for more details about the gains. There are many techniques how to tune the PID Controller [8, p. 302-306]. There are also **automated** techniques of the PID Controller tuning (for example the Naze32 board presented in the Section 2.2 has such feature), however our experience with those are not positive. With some time investment and basic knowledge of the PID Controller the **user can tune** the PID Controller *much better* than the automated mechanisms.

Choice of the tuning method is in this case a matter of personal preference. We have decided to use the method which is widely used amongst other RC hobbists and that is the **Trial and Error** method.

The Trial and Error method is based on setting a particular value and observing the changes which this value causes in the system. The process of the tuning starts with the following values:

$$K_p = 1$$
$$K_d = 0$$
$$K_i = 0$$

K_d and K_i are initially set to 0. The initial value of the K_p is *not important* and will be adjusted in the following steps.

At first the K_p will be tuned. The value of the K_p is being increased until the whole aircraft becomes **unstable** and **starts oscillating**. Once this behavior is reached the K_i gain is tuned, to **stop** the oscillations.

Increasing the K_i decreases the oscillations but **increases the overshot**. The overshot can be explained on the following example.

The aircraft is heading towards earth - its pitch angle is *less than* 0. We want to *level* the aircraft, therefore we *pull* the elevator stick of the transmitter. But we have pulled the elevator stick *too much* and the airplane is heading towards sky - its pitch angle is greater than 0. We have overshot the desired pitch angle of the airplane.

Consider this example in terms of the PID Controller. The setpoint is the desired pitch angle, which is 0. Initially the pitch angle was *less than* 0, which determines the *error*. Based on the error we have *pulled* the elevator stick, but we have pulled too much and the process variable is now *greater than* the setpoint. Later we have pushed the stick and so on, until the aircraft becomes leveled and the difference between actual pitch and desired pitch is acceptable. **Overshot is situation** when the process variable becomes greater than setpoint, if it has been initially smaller, or smaller than setpoint, if it has been initially greater. The K_i must be tuned to minimize the overshoot while suppressing the oscillations. Certain amount of overshot is required, otherwise the PID Controller will adapt to changes of the setpoint very slowly.

At last the K_d gain is tuned. Increasing the K_d gain **decreases the overshoot** but the system is much more sensitive to noise (in our case vibrations from the engines). Usually the K_d and K_i are *significantly smaller* than K_p.

The theory of PID Controller is quite simple, however implementing this theory is quite time consuming task. The Raspilot therefore have a special Command to alter the gains of the PID Controller from the ground (see Table 4.1).

4.4.7 Servo Control

To control the servos the Arduino must generate the appropriate PWM pulse. More details about PWM pulses are present in the Section 2.3.3. We are using the *default* Servo library which is available for Arduino [33]. With this library we can set either the required **angle** of the servo (from interval from 0° to 180°) or we can set the required **PWM frequency**. We have decided to set the PWM frequency as we find this approach slightly more user friendly (the majority of other autopilot systems presents the PWM frequency instead of servo angle). Also it can be easier to imagine the meaning of a PWM frequency in case of the ESC.

The servos however require certain amount of electrical current (in order of amperes) which cannot be provided by the Arduino [34]. Therefore, they must be powered by the **external power source**. Refer to the Section 4.7 for more information about interfacing and powering the servos and Arduino.

4.5 Android Application

In this Section we discuss the implementation details of both Up Library for Android and the Raspilot for Android. At first we will present the technical details and then the Up Library and the Raspilot for Android.

4.5.1 Technical Details

The Android framework uses the Java language, therefore also the Up Library and Raspilot for Android are written in Java. Both Up Library and the Raspilot can be divided into three main parts:

1. sensor reading
2. communication
3. graphical user interface - GUI

The *sensor readings* are implemented mainly in the Up Library, while the *GUI* is implemented mainly by the Raspilot (or any other custom application which uses the Up Library). The larger portion of the communication part is implemented in the Up Library, but some implementation is also required by the Raspilot. The following Sections discusses these parts more in detail.

4.5.2 Up Library for Android

The Up Library encapsulates the common functionality which is required in all applications using the Up Library. High level requirements are:

- Commands, Command Handlers, Commands Receiver and Executor
- Providers of the sensor data and Controller of Providers (discussed later)
- communication with the Raspberry (in case of airborne device) and the aircraft (in case of ground device)

All of these three requirements are discussed more in detail in following Sections.

Up Singleton Class

Core of the Up Library for Android is present in the `Up` class which is a **Singleton**[1]. This class contains references to Command Executor and Receiver.

[1]Singleton is a design pattern. A Singleton class can be instantiated only once. In our context to obtain a reference to such class we could call something like `Up.getInstance()`

Commands

The Commands are handled similarly as in the Up framework. We can create Command Handler for particular Command which defines an action. This action is executed each time the Command is received. Command Handlers are registered in the Command Executor (present in the `Up` singleton).

Providers

For comprehension we state the main purpose of the Provider, which is reading, processing and transmitting sensor data to the airborne Raspberry Pi.

The Android framework works with the sensors in following manner:

1. obtain reference to sensor (*might fail* if the device does not contain required sensor)
2. register for updates (usually contains some configuration parameters and the listener for the updates)
3. unregister the updates

This flow is reflected in the `UpProvider` base class, which all Providers extend.

Providers are registered in the `ProvidersController`. The Providers Controller implements the registration, unregistration process and it starts and stops the Providers.

During the **start** the Provider should **subscribe** to receive updates of the sensor. During the **stop** it should **unsubscribe** the updates. Once the Provider is started, it registers for sensor updates and each time the sensor updates its value, the provider processes the new value and sends it via the Base Up Service to the Raspberry. The Base Up Service is discussed later. One of its purposes is to manage communication between the airborne Android and Raspberry (or remote server).

The Up Library for Android contains the following basic Providers:

- Altitude Provider - reads the altitude from the barometric sensor
- Battery Level Provider - reads the battery level
- Location Provider - reads the GPS location
- Orientation Provider - reads the accelerometers and gyroscopes to obtain the orientation of the device and rotation rates

Communication

The communication functionality can be split into two parts:

1. communication between the **airborne Raspberry** (or remote server) and **airborne Android** device
2. communication between the **ground Android** device and the **airborne Raspberry** (or remote server)

At first we will discuss the communication channel between airborne devices. This is implemented in the `BaseUpService`. This **started** Service manages socket to the Android Provider, which is part of the Raspberry application. The socket is implemented in the `UpSocket` class, which is presented in the next paragraph. Via this socket the JSON encoded Commands (using the General Message Transfer Protocol) or raw binary data in case of orientation forwarding (using the Orientation Forward Protocol) are being transported.

The sockets are administered in the `UpSocket` class. This class takes care of:

- creating, connecting, reconnecting (in case of connection loss) and closing the socket
- deserialization of the received data and forwarding the received Commands to the Command Executor
- serialization of Commands being sent

Apart from the Commands handling the Base Up Service also has the reference to the `ProvidersController` class. The purpose of the Base Up Service is to **encapsulate** the **process of reading, processing and transmitting** the sensor data to the Raspberry.

The channel between ground Android device and Raspberry Pi is handled similarly. The `TelemetryService` class is implementing this feature. The Telemetry Service utilizes the previously mentioned `UpSocket` to connect to the proxy server between airborne and ground devices. Via this Service we can send Commands to the aircraft and receive telemetry. The Telemetry Service is a **bound** service.

4.5.3 Raspilot for Android

The Raspilot for Android is an Android application which uses the Up Library for Android. With the help of the library we are able to create application with features required from the airborne and ground device as well. Next Sections will discuss implementation details of these application more in detail.

One Application

We need two set of features in the Android application. One set for the airborne devices (sensor readings, etc.) and one set for the ground devices (telemetry reading, Commands transmission, etc.). We have decided to create only one application which contains both of these sets.

The motivation is quite simple. We do not want the user to install two different applications, rather we want the user to install one full featured application. On the other hand this implies more complexity in the resulting application. However, if the source code is well-structured, this is not an issue.

Graphical User Interface

The main portion of the implementation of the Raspilot for Android is the Graphical User Interface - GUI. The GUI implementation follows the guidelines of the Android framework. We found it unnecessary to discuss the GUI guidelines of the Android framework in this thesis as they are not interesting in terms of the thesis goals. The readers can refer to the Android framework documentation [35] for more information.

Airborne Component

The portion of implementation required for the airborne devices uses the Providers available in the Up Library. The Providers which are part of the Up Library covers all the basic sensors which are required. These Providers are registered within the Providers Controller. The transmission to Raspberry is done with the help of the Base Up Service. Of course other Providers might be implemented and added to the Providers Controller in the future.

In addition to the usage of components from Up Library, the Raspilot add a simple GUI, which displays the current status of the aircraft.

Ground Component

The main purpose of the ground device is to read and display telemetry. This is handled with the help of the Telemetry Service. It receives the Telemetry Commands produced by the Raspberry.

We need a Handler for these Commands. The Up Library contains an abstract Handler - the `BaseTelemetryHandler`. This class is a *generic class* and we can use the *type parameter* to specify what telemetry data we are interested in. Besides this we need to specify the action and the Handler is all set and can be registered.

Apart from telemetry display, the ground device has the ability to send Commands to the airplane. For each Command which we wish to send, we have to create appropriate GUI. It usually contains some kind of input where the user enters desired value. The value is then wrapped into the particular Command data and transmitted via the Telemetry Service.

For example if we wish to set the frequency of the telemetry updates, the Raspilot for Android follows this scenario:

- user opens the Fragment which contains list of all Commands which can be sent
- user enters the desired frequency
- after selection the Fragment creates the `TelemetryFrequencyCommand`, containing the desired frequency
- created Command is sent via the Telemetry Service to the proxy server, which forwards it to the airborne Raspberry
- the Command Executor on the Raspberry executes the appropriate action

4.6 Server Application

This Section describes the difference between the Up and Raspilot run on the Raspberry and on the remote server.

The Raspilot run on the remote server is used in the Black Box Mode. It makes no sense to try to connect to Arduino because there is no Arduino to connect to. But many of other Modules can work in the same way as if they are being run on the airborne Raspberry. Therefore, the mechanism of **conditional Module load** is introduced to the Up.

4.6.1 Conditional Module Load

Each Module is asked during startup (via a method) whether it should be loaded or not. It is then a matter of *configuration* if the Raspilot is run in Black Box or Flight Controller Mode. This configuration can be saved for example in a configuration file or can be specified via the environment variables. There are of course more techniques how to represent configuration. The Up utilizes the configuration file[1]. It is also possible to alter the loaded Modules by editing the `Cogfile.yml`.

This mechanism can be used not only to control the availability of the Modules, but can be used to **alter the behavior** of the Modules. The functionality which should differ in the Black Box Mode and the Flight Controller Mode can be extracted into separated Modules. We can then specify, using the conditional Module load, which Module should be loaded in which mode. Another usage of the conditional load is for example in case of an airplane with special capabilities such as landing gear.

4.6.2 Up Runner

The Up Runner (see Section 3.4 for more information) is the special application run the remote server when Up is in the Black Box mode. Its purpose is to **spawn** the Up application upon request and *manage* the spawned instance.

The Runner should run indefinitely. Because the remote instance of the Raspilot is run on the Linux servers provided by the university, we have decided to implement the Runner as an application which is managed by the **Upstart** manager [36]. However, this decision is absolutely implementation dependent and neither the Raspilot or Runner, nor Up framework is affected by this choice.

Upon receiving a spawn request, the Runner checks if the Raspilot is already running. If so, it does nothing and notifies the requester that the process is already running. Otherwise it spawns the Raspilot and notifies the requester about the result of the Raspilot startup. The requester can connect to the Raspilot once the response from the Runner is obtained.

Currently only **one instance** of the Raspilot can be run at the time. In the future versions this might be changed so **more instances** of the Raspilot can be run concurrently. Changes of the Runner will be necessary. Presently the response contains only some **basic information** such as flag whether the spawn was successful and PID of the spawned process. The Raspilot itself needs some ports for communication with the airborne Android device. Also the Ground Proxy consumes some ports. If multiple concurrent instances should be possible,

[1]The Modules can read the configuration file during startup and respond based on the content of this file.

these ports cannot be specified in the code, but must be **dynamically** assigned and **advertised** to the spawn requester.

Some other possibilities of the Runner changes are being analyzed such as **specifying the configuration** in the spawn request. The Runner will then spawn the Raspilot with the provided configuration. This can be useful also in the Flight Controller mode (which implies the presence of the Runner also in the airborne Raspberry), when the user can specify the type of the aircraft before the startup of the Raspilot and there will be no need to alter the configuration file manually. With this option enables us to use **one set of the hardware**[1] components between different airplanes, which reduces the costs.

4.6.3 Ground Station Proxy

The Ground Station Proxy is application run on the remote server and it serves as proxy between the Ground Station and the airborne Raspberry. Its implementation is quite simple. The Ground Station Proxy manages **two communication channels** bound to two *different* ports. Data received from the first port are forwarded to the second and data received from the second port are forwarded to the first port. With this setup the Ground Station connects to one port, the Raspilot to the second port and with the help of the Ground Proxy they can communicate.

4.7 Hardware

In this Section we explain the necessary hardware components such as the Printed Circuit Board (PCB) onto which the Arduino is connected and other hardware related details such as how are the devices powered and protected while airborne.

4.7.1 Printed Circuit Board for Arduino

The Printed Circuit Board (abbreviated as PCB) makes it easier to **avoid wire junctions** and **short circuits**. Also the PCB is much more space efficient than point-to-point wire connection. Only the Arduino requires at least 10 connections for the PWM readings (6 channels input, 4 channels output). But then there is the IMU, we need to power up the devices, all of these requires more and more wire connections and we will be soon overwhelmed with the wires. The PCB brings organization and makes the whole process of interconnecting components much easier.

The PCB used for the Raspilot have more assignments:

- power up the servos, Raspberry, Arduino and MPU6050
- interconnect the servos and Arduino
- interconnect the MPU6050 and Arduino

[1]One Raspberry, one Arduino, one IMU and one Android device can be shared between different aircrafts. This however implies some sort of simple interconnection of the Raspilot hardware and the aircraft hardware (servos, motors). Otherwise such sharing will not be practical.

For more details about the designed PCB (including diagram) refer to Attachment 3 - Printed Circuit Board.

4.7.2 Powering the Devices

The system must be somehow powered while airborne. We can use the present on board batteries, however this might require some sort of filtration due to the noise from the engines which are also powered by the battery. Other option is to **add another battery** which does not power the engines (therefore filtration is not required). However, this adds another **weight** which the airplane must carry. Yet on the other hand this might serve as a redundant power source (if properly connected) which can be used in case of failures of the main battery. Therefore we have decided to use additional battery. The weight of the battery should be reduced as much as possible. With the current setup (Raspberry Pi, Arduino, Android, 4 servos to control the airplane) we are using the 1300 mAh 2S Li-Poly battery. This battery powers the PCB which powers the Raspberry Pi, Arduino, IMU, Android, the RC receiver and the servos. The main battery powers only the engines. With this setup if we run out of the power in main battery, the airplane is still **maneuverable** although without the option to throttle up the engines.

4.7.3 Physical Protection

In case of a catastrophic crash we **cannot** provide reasonably light and inexpensive casing for the hardware components. However, it makes sense to try to protect the hardware in less serious cases such as **rough landing** or **light crashes**. Materials such as plywood should be considered when creating such encasement.

5. Experiment and Discussion

This Section presents the experiment which should validate the results of this thesis, how this experiment has been conducted and a short discussion of the results. At first the tests scenarios are presented, and then the results of these tests are reviewed.

5.1 Validating the Communication Channel

The first test is designed to validate the communication channel between the ground device and the airborne devices. It focuses on user experience, mostly in terms of latency.

The latency of this channel has been tested during the development, including *various* connection types (all components on the same Wi-Fi network; airborne devices connected to Wi-Fi network, ground device to cellular network; everything connected to cellular network), however, this needs to be tested in the real environment. In this test, the test aircraft (presented in Section 5.2) carries the Android device, operating in Black Box mode.

5.2 The Test Aircraft

In order to test the Up and Raspilot in real situations an aircraft is required. Because of the higher risk of crash during the test flights, the aircraft with following characteristics has been chosen:

- the aircraft is a glider[1]
- it is reasonably inexpensive
- it is big enough to carry all the required hardware

For this purpose the EasyGlider by Multiplex has been chosen. The fact that this airplane is a glider reduces the risk of injury with propellers (which might be unintentionally spun up or spun up because of a bug). The particular aircraft chosen for this experiment has been crashed and repaired several times before the testing of the Up, which decreases its economical value (however its personal value is quite high).

For illustrative photos of the test aircraft refer to Attachment 2 - Photos of the Test Aircraft.

5.3 Validating the Autopilot

Purpose of this test is to validate the ability of the autopilot to control the aircraft.

[1]The glider does not have an engine. Therefore, another airplane is required to tow the glider to some altitude.

Because of the airplane's size, only the Arduino and MPU unit are on board during this test. However, this does *not influence* the validity of the experiment, because of the following.

When the Arduino is not connected to Raspberry, the *only* difference is the lack of communication between these devices. The communication might make the cycle time[1] longer. Longer cycle time could affect the attitude control ability of the autopilot.

However, the tests have shown *no significant increase* in cycle time when the devices are connected and when they are not. Also there has been tests on ground and there has been no observable change when all devices has been connected or only Arduino and MPU has been connected.

This test as a side effect created a **new possible mode of use**, when only Arduino and MPU unit are on board. This mode has not been considered before, but the architecture of the Up and Raspilot makes it possible.

5.4 Running Raspilot on Raspberry

So far the proposed tests does not include testing the performance of the Raspilot on Raspberry Pi. The computational power of Raspberry might affect the overall performance of the framework.

It is not considered necessary to run the Raspilot on Raspberry in a flying aircraft because it will bring nothing relevant in context of whether the Raspilot will run on Raspberry or not. Therefore this test is conducted on the ground.

The Raspilot is run on Raspberry with all other on board devices (Arduino and Android) connected. The Raspberry and the ground station devices are connected via the *cellular network*.

5.5 Results of Communication Channel Validation

In this Section the results of the communication channel tests are reviewed. These tests have shown that the latency is more than acceptable (below 250ms) in areas with *good signal coverage*. The delay between the orientation change of the aircraft and the change of relevant user interface component is surprisingly low. In fact it *should be possible to pilot the aircraft* using the data from Raspilot's telemetry.

The only problem is the occasional lags which are caused by the cellular network (sometimes the are several messages delivered at once). However, in critical situations, for example when the aircraft flies behind a building or into the sun, it is possible (although not ideal) to pilot it using only the data from the telemetry.

Also the reconnection mechanism proves itself. In areas with poor signal coverage the telemetry feed has been reestablished after a connection loss. In these areas the lags were more often.

[1]Cycle time is duration of execution of the *loop* method.

5.6 Results of Autopilot Tests

The autopilot tests were more complex. At first the airplane had to be properly trimmed[1] because of the newly added weight. Also the center of gravity must be set accordingly.

Once these tasks had been completed, the airplane was taken into tow. The first flights were a bit bumpy, because the PID controller had not been tuned properly. More specifically, the K_p tuning parameter had been set too high. After lowering this parameter the aircraft had become more stable. Currently it is set to $K_p = 6$ for the stabilize PID controllers and $K_p = 0.7$ for the rate PID controllers. Please note that this is relevant only for the test aircraft, in other aircrafts the values might differ.

During one of the test flights the autopilot saved the aircraft. The pilot had lost orientation because the airplane had flown into the sun. When the sun stopped blinding the pilot, the aircraft was headed towards the ground. The autopilot had been activated, and the aircraft was leveled. If the autopilot had not leveled the aircraft, it would have crashed without any doubts.

The controlling of attitude is the most critical task the autopilot must handle. Because it succeeded in this task, we conclude it can also achieve other tasks, such as the Return Home function, which is not implemented in the Raspilot because of the reasons stated in the Section - Current Limitations. The Return Home function relies on GPS data in addition to the orientation data.

In order to support the statement that the Return Home function is possible within the Raspilot, let us at first review how the Return Home function works. Based on the current GPS location and the GPS location of take off point the *required heading* is calculated. Required heading with the actual heading enters the *Navigation PID* controller. The Navigation PID controller outputs the required heading which then enters the Stabilize and Rate PID controllers pipeline. The situation in current version is following: the *required change of heading* is always set to 0. One can think of this as if the aircraft is required to hold its current heading. Therefore the only change which the Return Home feature brings to the existing situation, is the calculation of the required heading and based on the required heading, the calculation of the required *rate* of change. If it is currently possible to hold the rate of change in heading at 0, it is possible to hold it at some other value as well.

5.7 Results of Raspilot Run on Raspberry

During the evaluation of this test, there were some difficulties with installing libraries, etc. All of them are discussed in the **Troubleshooting** Sections of the Up's Documentation.

Once these has been solved, the delays between forwards of orientation and receiver data from Arduino must be increased, as they overloads the Raspberry. Besides these the Raspilot had been run on Raspberry without any complications.

Current values of delays between forwarding are following:

[1]Trimming is a process when the pilot alters the neutral position of a control surface. The goal is that when all controller sticks are in neutral, the airplane should fly leveled (if there are no external effects such as wind).

- 50ms delay between RX data forwarding
- 50ms delay between orientation data forwarding

The utilization of the B+ model during the tests[1] has been following:

- CPU ca. 40%
- free RAM ca. 80%

5.8 Summary

The tests have shown that the Raspilot is able to execute all required tasks when deployed to *real* environment. Some of the default values needs to be adjusted, but there is no need to make extensive changes to make the whole system functional and operable as requested.

Because of the capability to implement the required design changes (discussed through the whole Chapter 4) and the results of the tests we conclude, that all of the requirements (both functional and non-functional) have been met and satisfied.

[1]Both Android and Arduino were connected.

6. Conclusion

In this Chapter we present the overall results of the thesis. The Chapter is divided into four sections which discusses the following topics:

1. the problem which this thesis addresses
2. the solution of the problem
3. current limitations of the Up framework and the Raspilot
4. future research

6.1 The Problem

The main problem which this thesis is addressing is the *lack* of modular autopilot system for the RC airplane models. Many superb autopilots already exists, but none of them can be modified or extended easily. This need is satisfied by the **Up framework** which is used in the **Raspilot application**.

6.2 The Solution

The Up framework has been designed with these key requirements:

- modularity
- extensibility

Meeting both of these requirements *has been validated* during the development of the Raspilot. Many versions of the Up framework and the Raspilot has been created till the final version for this thesis has been reached. Both the framework and the Raspilot has undergone many changes between the versions (refer to the Section 4.2.2).

The **concepts** of the *Command Receiver* and *Command Executor* as well as the concept of *Modules* has proven themselves as a good set of instruments for creating an autopilot. These concepts has also proven themselves as an apparatus which *support* the later **changes** and **modifications**. Therefore we consider the Up framework useful when solving the stated problem.

In the beginning, the Raspilot was intended to be a single application running on the Raspberry Pi with an Android counter part (which however is quite separated and does not utilize the Up framework). But based on the analysis, development and tests this initial concept has been changed quite dramatically. The Raspilot is currently a system consisting of the following applications:

- the Raspberry Pi application (which uses the Up framework)
- the Arduino application
- the Android application

- the proxy application on the remote server[1]

All of these applications needs to cooperate, exchange messages to make the following possible:

1. send message from ground Android device
2. the message is transmitted via the proxy application to the airborne Raspberry
3. the Raspilot running on the remote Raspberry receives the message
4. with usage of the instruments of the Up the appropriate action is executed
    - an action is executed directly in the Raspberry application (such as change of the telemetry updates frequency)
    - an action is executed on the Arduino device
    - an action is executed on the airborne Android device

These steps work of course also in reverse order, when the message is originating from the airborne device. With this stack we have the instrument to exchange data between ground Android device and airborne devices.

As has been stated in the Section 4.2.2 the Raspilot has undergone many changes. We consider the ability of the Raspilot to accommodate these changes without huge ripple effect as a proof of its modularity and extensibility.

6.3 Current Limitations

Despite meeting the key requirements there are currently a certain number of *drawbacks* in the Raspilot.

One of the main drawbacks is the *space* and *weight* requirements of the airborne devices. However, we consider the currently used hardware a *prototype* and the possibility of optimizing these requirements are being examined. Refer to the next Section for more information.

Probably the weakest element of the Raspilot is the Flight Controller. The current flight controller serves as a proof that the Raspilot *has* the ability to control the airplane reliably. But it lacks some functions which has become common amongst other autopilots such as gimbal support, automatic return home in case of low battery, etc. However, we consider the implementation of the superb Flight Controller *not absolutely crucial* for the purpose of this thesis. During the analysis and development many autopilots have been examined. The scope of creating the autopilot is quite large and we consider creating new autopilot useless when there is a number of already developed autopilots. Therefore, we have decided to choose another approach, which is integrating the existing autopilots so they will cooperate with the Raspilot. This is a very hot topic in terms of future development.

[1]The proxy application on the remote serves as a mediator between the Raspberry Pi and the ground Android application.

6.4 Future Development

The space and weight requirements stated in the previous Section are *not crucial* as the Raspilot can be used in reasonably large aircraft (for example aircrafts with wingspan larger than 1.8m can accommodate the required hardware without much struggle). However, this limits the usage to bigger aircrafts only.

There are many possibilities how to lower these requirements. For example the special **Raspberry Pi Zero**[37] might be used instead of the regular Raspberry Pi. Other opportunity to reduce the weight is to remove the additional battery powering the Raspilot devices. This however implies research whether the airplane batteries can be used without any filtrations, if the filtration is required, which to use, etc.

The Android device initially serves as a source of the sensor data. However, during the development it has become a provider of the Internet connectivity more than a source of sensor data. Currently only the GPS data are being read from the Android. In terms of space and weight it is not very efficient. The dedicated GPS sensor will be much better. The removal of the Android device will imply two facts:

1. loss of the Internet connectivity
2. loss of the Black Box mode

The *loss* of the Internet connectivity means the Commands must be transfered via *another communication channel*. Either a new channel will be created or we have only the RC radio left. However, if we do not want to loose the Internet connectivity it must be provided by some other device. For this purpose the **USB modem** might be used. This modem will contain the SIM card with active data plan. The space requirements and weight are much better than in the case of the Android device. Of course these devices must be examined in order to select the best for our case.

If the loss of the Internet connectivity is acceptable, then we have to use the RC radio to transmit our data. Currently there are *not many* RC radios capable of this. An example of such RC radio are the radios running the OpenTX [38] software. For more information about the OpenTX refer to following Section.

The Black Box mode has shown itself as *not very useful*. If we have aircraft big enough to carry the Android phone, it is probably big enough to carry the rest of the devices as well. Also the Android device is the biggest and the heaviest of all airborne devices. Therefore, the future versions of the Raspilot will probably abandon the Black Box mode completely.

OpenTX

The OpenTX [38] is the open source firmware for RC radios. One of its features is the ability to transmit telemetry from the aircraft. We can use this and feed the telemetry with our data.

But this can be brought further. The principle of the OpenTX telemetry is a **bus of sensors**. Each sensor has its ID. The radio receiver in the aircraft requests the *update* of the sensor with specified ID and transmits the sensor data to the transmitter [1]. There is a component which can be added to the sensors bus and which serves as a bridge between the **OpenTX and UART** protocol. With this sensor it might be possible to establish the Serial communication channel between the airborne RC receiver and RC transmitter. The OpenTX powered transmitters have the ability to output the received telemetry via their Serial port. This Serial port can be read by the Ground Station device. If our previous assumptions about the OpenTX and the OpenTX and UART protocol bridge are correct than the following should be possible:

- the Raspilot application will create a Command
- the Command will be transmitted via the OpenTX and UART protocol bridge
- the OpenTX telemetry will transmit the data to the RC transmitter
- received data will be outputted to the Serial port of the RC transmitter
- an application will read these data and execute relevant action (for example update the dashboard, etc.)

This is currently more of a concept than a reality. However, if we manage to achieve this usage, it might bring a *whole new set of possibilities* to the Up and the Raspilot.

Flight Controller

As has been stated in the Section Current Limitations the Flight Controller is somehow limited. Because there are many superb autopilots already we find it useless to create a new one. Therefore, the future versions of the Raspilot and the Up will probably more focus on integrating existing autopilots rather than creating a new one.

For example the DJI has created and released a SDK while this thesis has been developed. A possible integration of DJI's solutions and Up should be analyzed.

6.4.1 Summary

After all we consider the thesis to successfully achieve the goals. The Up framework has been designed an implemented in such manner, that all functional and non-functional requirements are satisfied. The Raspilot proves the concept that the Up is usable as intended. These statements are supported by the results of the tests, which has been stated in the Section 5. These tests also supports the fact, that the Raspilot is usable in real RC models.

[1] It is a bit confusing that the radio receiver transmits data to the radio transmitter. However, the radio transmitter is usually called the device with which the user controls the airplane. The receiver is the device present in the aircraft.

Bibliography

[1] On Screen Display Autopilot for Model Airplanes User Manual. RangeVideo `https://www.rangevideo.com/index.php?controller=attachment&id_attachment=23`. Accessed: 2016-10-19.

[2] Naza-M V2 Mutlirotor Autopilot System. DJI `http://www.dji.com/naza-m-v2/feature`. Accessed: 2016-10-19.

[3] Naze32 User Manual, AfroFlight `www.abusemark.com/downloads/naze32_rev3.pdf`. Accessed: 2016-10-19.

[4] MultiWii Wiki, MultiWii `http://www.multiwii.com/wiki`. Accessed: 2016-10-19.

[5] ArduPilot Home Page, ArduPilot `http://ardupilot.org/`. Accessed: 2016-10-19.

[6] What Is Arduino, Arduino `http://www.arduino.org/learning/getting-started/what-is-arduino`. Accessed: 2015-04-10.

[7] Arduino Home Page, Arduino `http://www.arduino.org`. Accessed: 2015-04-10.

[8] Aström, Karl Johan, and Richard M. Murray. Feedback systems: an introduction for scientists and engineers. Princeton university press, 2010: 293-314.

[9] Kaur, Amanpreet, and Amandeep Kaur. "An approach for designing a universal asynchronous receiver transmitter (UART)." simulation 2.3 (2012): 2015-2016.

[10] Official I^2C specification, NXP `http://www.nxp.com/documents/user_manual/UM10204.pdf`. Accessed: 2015-04-11.

[11] Barr, Michael. "Pulse width modulation." Embedded Systems Programming 14.10 (2001): 103-104.

[12] megaAVR Microcontrollers, Atmel `http://www.atmel.com/products/microcontrollers/avr/megaavr.aspx`. Accessed: 2015-04-10.

[13] Raspberry Pi - Teach, Learn, and Make with Raspberry Pi, Raspberry Pi Foundation `https://www.raspberrypi.org/`. Accessed: 2015-03-23.

[14] Raspberry Pi - FAQs, Raspberry Pi Foundation. `https://www.raspberrypi.org/help/faqs/`. Accessed: 2015-03-23.

[15] Raspberry Pi Model 3, Raspberry Pi Foundation `https://www.raspberrypi.org/products/raspberry-pi-3-model-b/`. Accessed: 2015-03-23.

[16] Build your own Quadcopter Flight Controller, Dr. Gareth Owenson
https://www.raspberrypi.org/products/raspberry-pi-3-model-b/.
Accessed: 2016-04-15.

[17] Activity — Android Developers, Google Inc.
https://developer.android.com/reference/android/app/
Activity.html. Accessed 2015-04-16.

[18] Fragments — Android Developers, Google Inc.
https://developer.android.com/guide/components/fragments.html.
Accessed 2015-04-16.

[19] Services — Android Developers, Google Inc.
https://developer.android.com/guide/components/services.html.
Accessed 2015-04-16.

[20] IntentService — Android Developers, Google Inc.
https://developer.android.com/reference/android/app/
IntentService.html. Accessed 2015-04-16.

[21] Welcome to Setuptools' documentation! - setuptools 34.3.3 documentation, Python Software Foundation.
https://setuptools.readthedocs.io/en/latest/. Accessed: 2016-11-10.

[22] PyPI - the Python Package Index : Python Package Index, Python Software Foundation.
https://pypi.python.org/pypi. Accessed 2015-04-28.

[23] Arduino - Compare, Arduino.
https://www.arduino.cc/en/Products/Compare. Accessed: 2015-07-03.

[24] Arduino - Software, Arduino.
https://www.arduino.cc/en/Main/Software. Accessed: 2015-04-10.

[25] Arduino - AttachInterrupt, Arduino.
https://www.arduino.cc/en/Reference/AttachInterrupt. Accessed: 2015-07-03.

[26] Arduino - ArduinoBoardNano, Arduino.
https://www.arduino.cc/en/Main/ArduinoBoardNano. Accessed: 2015-07-03

[27] Arduino - ArduinoBoardUno, Arduino.
https://www.arduino.cc/en/Main/ArduinoBoardUno. Accessed: 2015-07-03

[28] Gammon Forum : Electronics : Microprocessors : Interrupts, Nick Gammon
http://gammon.com.au/interrupts. Accessed: 2015-07-03.

[29] Mitra, Sanjit Kumar, and Yonghong Kuo. Digital signal processing: a computer-based approach. Vol. 2. New York: McGraw-Hill, 2006.

[30] MPU-6000/MPU-6050 EV Board User Guide, InvenSense.
`https://www.invensense.com/wp-content/uploads/2015/02/MPU-6000-EV-Board1.pdf`. Accessed: 2016-08-02.

[31] Arduino Playground - MPU6050, Arduino.
`http://playground.arduino.cc/Main/MPU-6050`. Accessed: 2016-07-30.

[32] Diebel, James. "Representing attitude: Euler angles, unit quaternions, and rotation vectors." Matrix 58.15-16 (2006): 1-35.

[33] Arduino - Servo, Arduino.
`https://www.arduino.cc/en/Reference/Servo`. Accessed: 2015-04-12.

[34] Arduino Playground - Arduino Pin Current Limitation, Arduino
`http://playground.arduino.cc/Main/ArduinoPinCurrentLimitations`. Accessed: 2015-04-12.

[35] Android Developers - Google Inc.
`https://developer.android.com/index.html`. Accessed 2015-04-16.

[36] Upstart - event based init daemon, Canonical Ltd.
`http://upstart.ubuntu.com/`. Accessed: 2015-11-16.

[37] Raspberry Pi Zero - Raspberry Pi, Raspberry Pi Foundation.
`https://www.raspberrypi.org/products/pi-zero/`. Accessed: 2016-09-07.

[38] Welcome to OpenTX - OpenTX.
`http://www.open-tx.org/`. Accessed: 2016-09-17.

[39] Twisted, Twisted Matrix Labs.
`https://twistedmatrix.com/`. Accessed: 2016-11-10.

[40] Fritzing Fritzing, Friends-of-Fritzing Foundation.
`https://fritzing.org`. Accessed: 2017-03-02.

List of Figures

List of Tables

Attachments

This thesis contains several addenda. This text contains the following attachments:

	Name	**Content**
Attachment 1	Installation Guides	installation Guides for all applications and libraries
Attachment 2	Photos of the Test Aircraft	ohotos of the test aircraft with the Arduino and IMU installed
Attachment 3	Printed Circuit Board	diagram of the printed circuit board

Attachments Overview

The burned DVD contains the following appendices:

Content	**Relative Path**
this text in a PDF document	`text/thesis.pdf`
sources of the Up Framework	`src/up/`
sources of the Raspilot Application	`src/raspilot/`
sources of the Android Application	`src/android/`
sources of the Arduino Application	`src/arduino/`
sources of the Arduino Application libraries	`src/arduino-libs/`
sources of the Proxy Application (for remote server)	`src/remote/proxy`
sources of the Runner Application (for remote server)	`src/remote/runner`
sources of the Raspilot (for remote server)	`src/remote/raspilot`
APK file of the Android Application	`build/android/raspilot.apk`
Up's Documentation	`docs/up-autopilot/`

Content of the Attached DVD

Attachment 1 - Installation Guides

This Chapter contains the brief installation guides. More detailed guides are available in the **Up's Documentation** which is burned on the attached DVD and is also accessible at:

`https://up-autopilot.github.io/`

Up

This Section contains basic installation guide for the **Up framework**.

Python 3.5

The Up requires *Python 3.5*. Many operating systems have a dedicated installer of this Python version. However, there are some operating systems which do not contain such installer. In this case Python 3.5 must be installed from sources[1].

Installation of Up

The Up contains the Setuptools[21] script which handles all the required dependencies. This script is present in the file `setup.py`. The Up should be installed as any other Python library. The following example, which shows how to install a Python library, is usable also in case of Raspilot and Cogs. The following should be executed in the root directory containing Up's (or Raspilot's or Cog's) sources:

```
$ python setup.py develop
```

Figure 6.1: Installing Python Module

Please refer to Setuptools' documentation[21] for more details about usage of Setuptools.

There might be an error while installing the `Twisted` library[2][39] on Raspberry, Macbooks and possibly other platforms. For troubleshooting, refer to the Up's Documentation. The solution is to install the Twisted from sources at first and then install the Up.

Raspilot

This Section contains basic installation guide for the **Raspilot** application which is run on the Raspberry. Please note that it is *expected* that the Up has been installed successfully.

[1]The installation guide for installing Python from sources is out of scope of this thesis, but is available in the Up's Documentation.

[2]Twisted Library is used by the Up for managing the network communication.

Installing Application

The Raspilot as well as the Up contains the Setuptools script which should be handled the same way as the Up's installation script.

Installing Cogs

The Cogs which are required by the Raspilot are defined in the `Cogfile.yml` file. These Cogs can be installed one by one from sources, or can be installed all at once. To install them all at once we can use the Cog Integration Utilities. Invoke the following in the root of the Raspilot:

```
$ up gather
```

Figure 6.2: Installing All Required Cogs

Configuring Cogs

Some of the Cogs need to be configured. All of the configuration files are placed in the `config` folder. However, the version of Raspilot on the attached DVD contains all required configuration files. For more information about the Cogs and their configuration files refer to the Up's Documentation.

Running the Raspilot

Once Raspilot and all Cogs are properly installed and configured, Raspilot can be run. The main script is present in the `main.py` file. This script does not require any other arguments.

Arduino Application

This Section contains basic installation guide for the **Arduino application**. The Arduino application should be build and uploaded via the Arduino IDE.

Dependencies

The Arduino IDE does not provide (at the time of creating this thesis) any utility for specifying dependencies. They must be therefore installed manually. The Arduino application required two additional libraries:

1. **PinChangeInterrupt**
   - created by NicoHood
   - tested with version 1.2.2
2. **PID**
   - created by Brett Beauregard
   - tested with version 1.1.1

Sources of both libraries are present on the attached DVD. Please refer to the following web page for detailed guide about installing libraries in Arduino IDE:

`https://www.arduino.cc/en/guide/libraries`

The required libraries can be installed from provided sources or via the Library Manager of the Arduino IDE.

Android Application

This Section contains basic installation guide for the **Raspilot for Android**.

The attached DVD contains the `raspilot.apk` file, which can be installed on the Android device. There are few requirements which must the Android device meet:

1. Android version 4.1 and higher
2. presence of GPS
3. presence of barometric sensor (required for airborne devices)
4. presence of accelerometers and gyroscopes (required for airborne devices)
5. sufficient storage

Server Applications

This Section contains basic installation guide for the Up framework components run on **remote server**. It is anticipated that there is properly installed Up on the remote server.

The configured Raspilot which can be installed on the remote server is present on the attached DVD. It should be installed using the Setuptools script. Also the Cogs must be installed. Installation process of Cogs is the same as in case of the Raspilot run on Raspberry Pi.

Besides the Up and the Raspilot the remote server contains the **Runner**[1] and **Proxy**[2] applications. Both should be installed via the Setuptools scripts.

Both the Runner and the Proxy should be run on system startup. Please refer to the system's documentation about how to run a program on startup.

[1]The Runner spawns the Raspilot instance upon request.

[2]The Proxy exchanges data between the Raspberry and ground device.

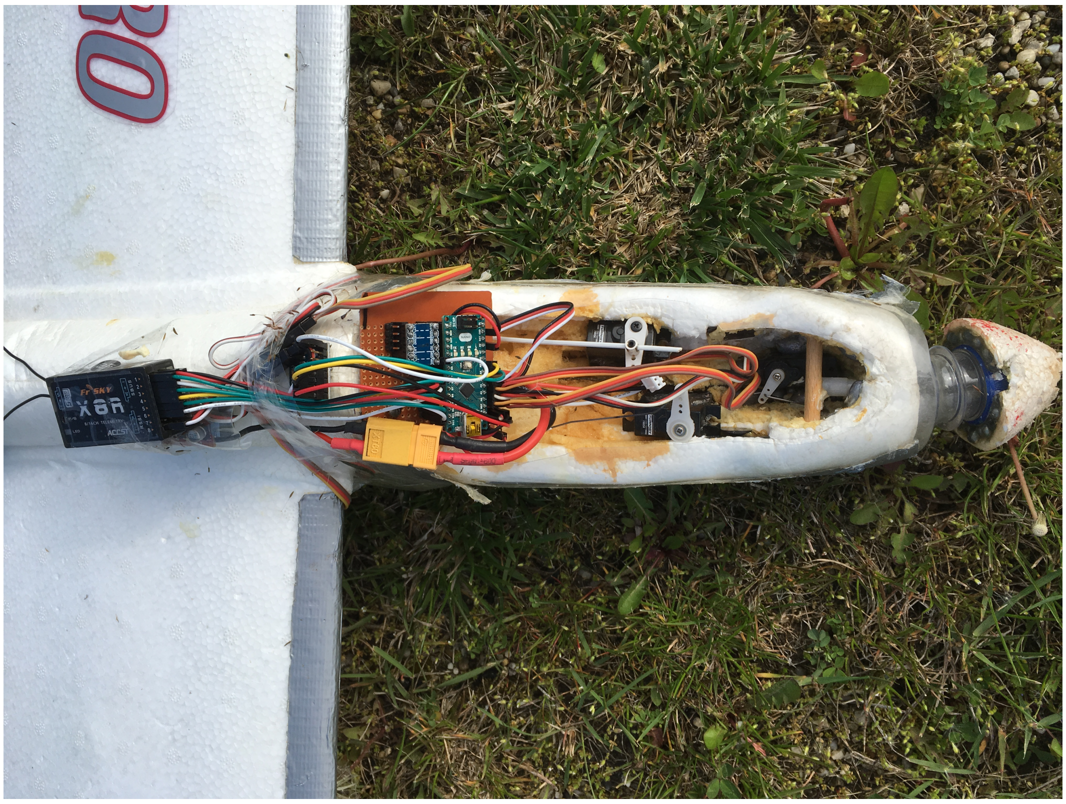
Attachment 2 - Photos of the Test Aircraft

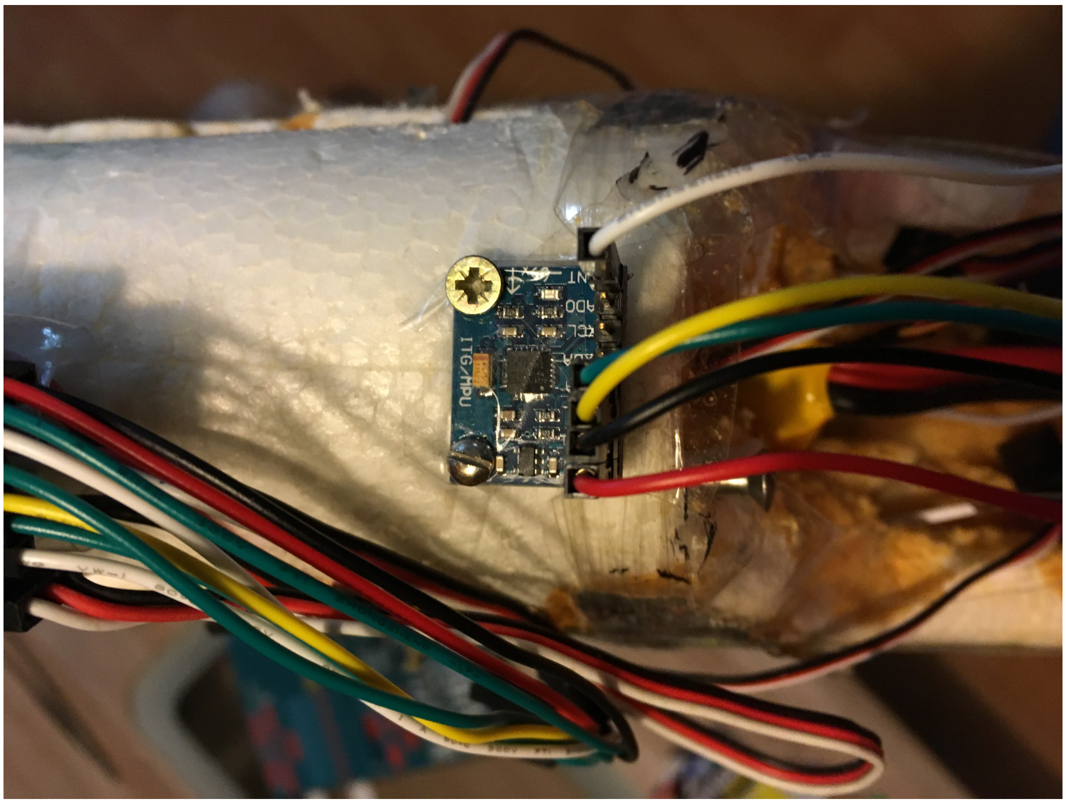
This attachment contains the photos of the test aircraft with the Arduino and IMU unit equipped.

The succeeding figures contain the following:

- the overall aircraft
- picture of the cockpit part of the aircraft where we can see the PCB containing Arduino
- the IMU unit mounted in such fashion, it is leveled with the ground

The Test Aircraft


The Cockpit with the Arduino on Printed Circuit Board

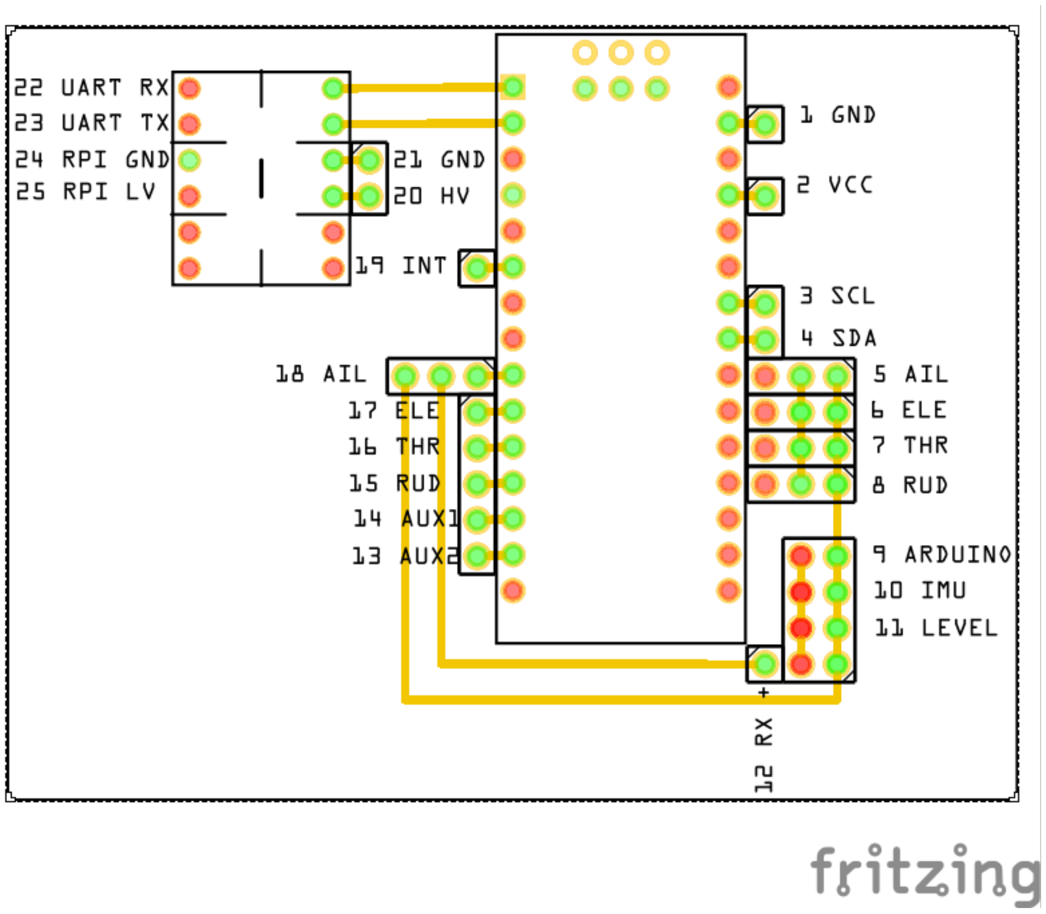


The IMU Unit

Attachment 3 - Printed Circuit Board

The third attachment contains the diagram of the printed circuit board (PCB) and some brief connection instructions.

The following Figure depicts the diagram of the printed circuit board. This figure has been created by the Frizting[40] program.



The Diagram of Printed Circuit Board

The yellow lines are PCB connections. The pins touching the Arduino, are connected with the nearest Arduino pin. In case of a servo connector (a connector with three wires, *(5)-(8)* and *(18)*) the pin nearest to the Arduino is the signal and the one on the other end is ground.

The designed PCB is capable of following:

- powering up the logic level shifter (discussed later), the servos, ESC, the Arduino and the IMU
- connecting RX and Arduino
- connecting Arduino and servos
- connecting Arduino and ESC

- connecting Arduino and logic level shifter
- connecting Arduino and IMU

Required Connections

The professionally designed and printed PCB should not need any additional connections. But because this is a home made PCB, there are some connections which must be added if the PCB should work correctly.

The pin *(1)* must be connected to the **right** pin of *(9)*. The pin *(2)* must be connected to the **left** pin of *(9)*. The whole power connectors section (four connectors from *(9)* to the bottom) have the **IC power supply - VCC pin on the left** and the **ground - GND pin on the right**.

Connecting the Level Shifter

Logic level shifter is a dedicated hardware component responsible for translating signal between two voltage logic schemes. The Arduino uses a 5V logic, while the Raspberry uses the 3.3V logic. If these devices are connected directly, the Raspberry could be damaged.

However, the Arduino and Raspberry might be connected via the USB cable. Then the logic level shifter is not required. Logic level shifter is required only if the Arduino connects *directly* to the UART pins of the Raspberry.

If there is a level shifter, the VCC pin of *(11)* should be connected to *20* and the GND of *(11)* to *(21)*.

The Raspberry should connect to the pins *(22)* - *(25)*. Connect pins *(22)* and *(23)* to the proper UART pins of the Raspberry. Connect the *(24)* to the GND pin of Raspberry and the *(25)* to the 3.3V pin of the Raspberry.

If we now power up the Raspberry and then the Arduino, the devices should communicate. The Raspberry **must** be powered first, otherwise the Arduino will not be discovered.

Connecting the IMU

The IMU unit requires several connections. The **SDA** pin of the IMU should be connected to the *(4)*, the **SCL** pin to the *(3)*. The **VCC** of the IMU should be connected to the VCC pin of the *(10)* and the **GND** pin of the IMU to the GND pin of the *(10)*. The **INT** pin of the IMU should be connected to the *(19)*.

If we now power up the Arduino, the IMU should power up as well and these two devices should communicate.

Connecting the RX, the Servos and the ESC

The generic servo connector has three wires:

1. **ground**, usually black or brown
2. **vcc**, usually red
3. **signal**, usually white or yellow

On the designed PCB the signal wire should be always nearest to the Arduino and the ground wire should be on the opposite site.

(13) - *(18)* are **input** connectors, while *(5)* - *(8)* are **output** connectors. A servo connector from the ailerons RX output should be connected to *(18)*. This connector will power the section of PCB which powers the servos. The signal wires from the remaining RX channels should be connected to the pins *(13)* - *(17)*. The servos should be connected to pins *(5)* - *(8)*.

With this connection, if we power up the RX and the Arduino, we should be able to control the servos and motors.

Powering the Arduino

The Arduino can be powered in two ways:

1. from the RX
2. via the USB cable connection

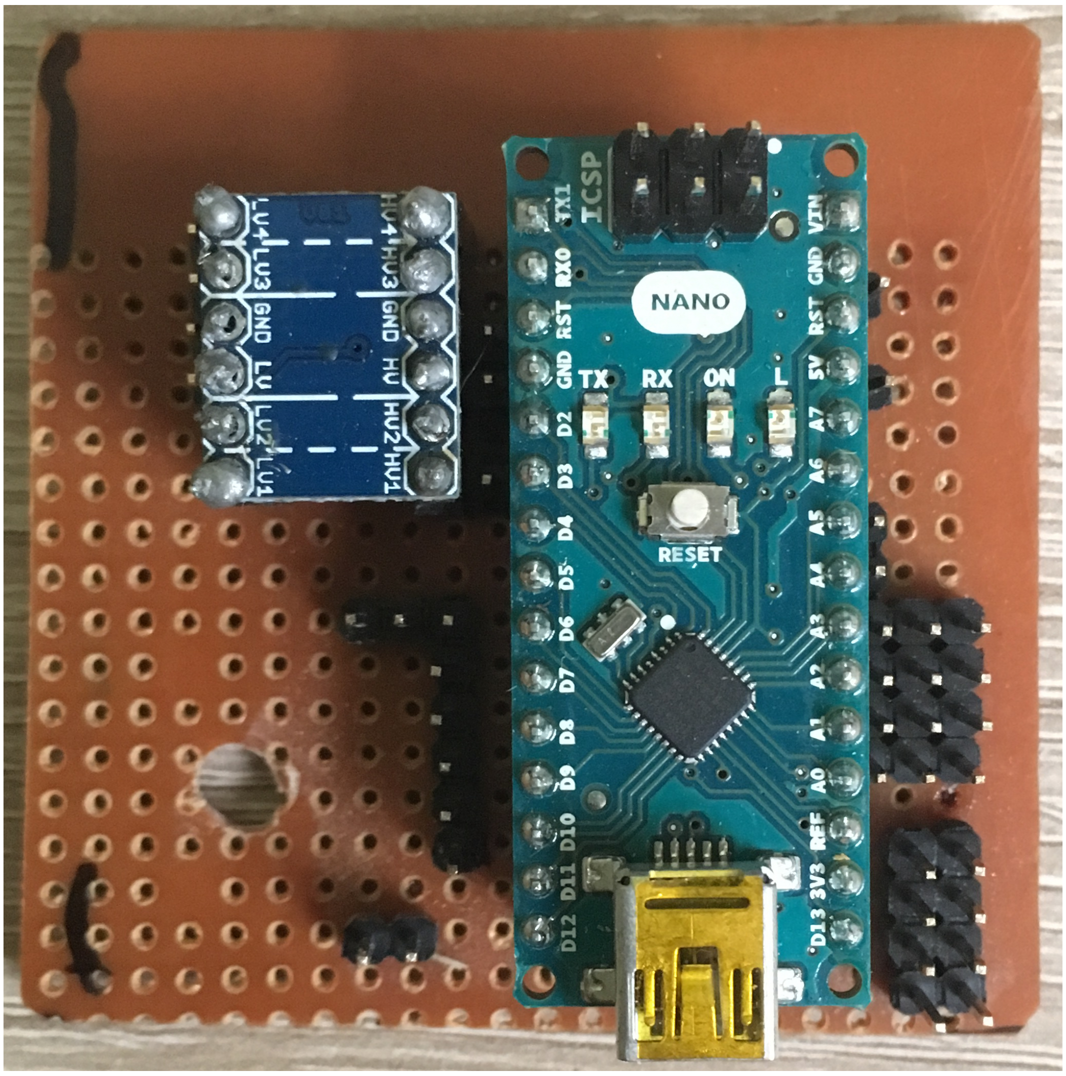
If we want to power the Arduino from the RX, pin *(12)* should be connected to its neighbor. If we now power up the RX, it should power up the Arduino, logic level shifter, IMU, servos, ESC and the whole PCB. But, if we disconnect the battery from RX and power up the Arduino via USB connection, then also all devices will be powered up including servos and ESC.

The pin *(12)* should not connected to its neighbor if:

- **the Arduino is powered via USB connection and RX is powered by another power source** - damage risk to RX
- **the Arduino is powered via USB connection and RX is powered by the PCB and there are servos and ESC connected** - damage risk to Arduino and USB host

Photo

The following figure depicts the real PCB created by the proposed design:


The Printed Circuit Board