Politecnico di Milano
Scuola di Ingegneria Industriale e dell'Informazione
Corso di Laurea Magistrale in Ingegneria Informatica
Dipartimento di Elettronica, Informazione e Bioingegneria

Re-engineering of Aerial Drone Autopilot Firmware with Reactive Programming

Relatore: Prof. Luca Mottola

Tesi di Laurea di:
Endri Bregu
Matricola 804686

Anno Accademico 2014–2015

To my family

Acknowledgments

It is a pleasure to thank those who made this thesis possible with advices, critics and observations.

I would like to thank my supervisor Prof. Luca Mottola. Without his help and support, this thesis would not have been possible.

A special thanks to the students of the lab and especially to the «Salotto Sibaritico» group for these three years.

A profound thanks to Fabjola for the affection and care shown during these years and especially in life.

I owe my deepest gratitude to my families and friends. They were always supporting me and encouraging me with their best wishes.

Abstract

Nowadays, drone-related technologies are growing rapidly. Although they are widely used in the military field, their lower costs are enabling their spread both in the professional and in the hobbyist fields. At the same time, a rapid growth of developer communities contributing to drone-related software development is overtaking. The largest open-source community in this field is Ardupilot. Every 5-6 months a new version of their software is released with new features up-to-date with the always evolving new boards.

The board, along with the rotors, is the main piece of hardware that composes a drone. The board is equipped with a firmware which is the component in charge of collecting the sensors' output and use it to plan engine adjustments in order to achieve a stable flight.

The Ardupilot firmware uses a control loop, which manages sensor readings and executes tasks *periodically*. These tasks process the sensor data and send information to the engines. The computation on incoming data is performed in any case, even if pieces of data detected by sensors do not vary. We would rather like to run a system that *reacts* only to new sensor readings, thus optimizing the system response and decreasing the computation time. This improvement enables the execution of a larger number of control tasks giving a further increase in the drone's flight stability.

The Reactive Programming Paradigm, with its peculiarity of dynamically reacting to changes in input values, seems to be a natural fit for drone control loops; but nobody applied it to them before.

In this thesis we explain how we applied this paradigm to the Ardupilot firmware and how we overcame hardware and software limitations of the existing reactive programming libraries when integrated with the Ardupilot firmware by developing a brand-new open-source library.

We evaluate the performance of our custom firmware comparing two different flight plans executed with the original and custom firmware. Thanks to the data obtained, we are able to demonstrate that our custom firmware is more precise at positioning the vehicle and more efficient in terms of computation time compared to the original one.

Sommario

L'uso più comune dei droni ad oggi è stato quello militare, ma i recenti sviluppi tecnologici in questo campo con la conseguente diminuzione dei loro costi ha reso accessibile al pubblico il loro acquisto. Inoltre, questo fenomeno ha dato luogo ad un sostanziale aumento del software per il controllo di droni da parte delle comunità di sviluppatori. Una delle più importanti e più attive oggigiorno è Ardupilot.

Un drone è generalmente costituito da due componenti principali: i motori rotazionali e la scheda di controllo. Su quest'ultima viene installato il firmware del drone: un software di controllo che garantisce un volo stabile leggendo i dati forniti dai sensori e utilizzandoli per governare i motori.

Nel caso di Ardupilot, il firmware svolge le sue funzioni per mezzo di un ciclo di controllo eseguito *periodicamente*, che quindi raccoglie dati anche in assenza di una loro variazione. Ci siamo resi conto che sarebbe meglio avere un sistema che, anziché agire in modo periodico, *reagisca* alle variazioni dei dati in ingresso.

Ciò porterebbe ad ottimizzare la risposta del sistema stesso a ridurre il tempo richiesto per la computazione rendendo possibile l'esecuzione di altre funzionalità di controllo che migliorino a loro volta la stabilità del drone durante il volo.

Il *Reactive Programming Paradigm* (Paradigma di Programmazione Reattiva) ci è sembrato essere particolarmente adatto allo scopo, proprio per la sua intrinseca capacità di reazione dinamica alle variazioni dei dati in ingresso.

In questa tesi spieghiamo come abbiamo applicato questo paradigma al firmware di Ardupilot e come abbiamo sviluppato nella sua interezza una libreria *open-source* per il Reactive Programming date le incompatibilità sia software che hardware di quelle già esistenti con le più utilizzate schede di controllo.

La valutazione del nostro firmware modificato è stata fatta tramite l'analisi di due piani di volo, il primo eseguito con il firmware originale, il secondo con quello modificato. Dai dati ottenuti si evince che il nostro firmware garantisce, rispetto a quello originale, una maggiore precisione nel posizionamento del drone e una maggiore efficienza per quanto riguarda i tempi di computazione.

Contents

1 Introduction **1**

1.1 General context 1

1.2 Motivation 2

1.3 Outline 3

2 State of the art **5**

2.1 Reactive programming 5

2.1.1 Basic abstraction 6

2.1.2 Evaluation model 7

2.1.2.1 Pull-based 7

2.1.2.2 Push-based 8

2.1.2.3 Push versus Pull 8

2.1.3 Glitch avoidance 8

2.1.4 Lifting operations 10

2.1.5 Multidirectionality 10

2.1.6 Support for distribution 10

2.1.7 Tools 11

2.2 Multicopters 12

2.2.1 Overview 12

2.2.2 Autopilot suite 14

2.2.2.1 Hardware 16

2.2.2.2 Firmware 18

2.2.2.3 GCS (Software) 19

2.2.2.4 SITL Simulator 22

3 Analysis of Ardupilot Implementation **25**

3.1 Code structure 25

3.1.1 Main loop 26

3.1.2 Fast loop 26

3.1.3 Scheduler 26

3.1.4 Running new code 29

3.1.5 Flight modes 31

4 Re-engineering of Ardupilot's Flight Control 35

4.1 ReactiveCpp 35

4.1.1 Problems 35

4.1.2 Solution 36

4.2 Example in a control loop 43

4.3 Ardupilot re-engineering 46

4.3.1 Overview 46

4.3.2 Setup 53

4.3.3 Main Loop 54

4.3.4 Fast Loop 56

4.3.5 Scheduler 57

5 Evaluation 59

5.1 SITL 59

5.1.1 Yaw 60

5.1.2 Pitch and Roll 63

5.1.3 Error altitude 66

5.2 Code execution 70

5.3 Summary 72

6 Conclusions & Future Work 77

Bibliography 81

List of Figures

1.1 Drone configuration . . . 2

2.1 Push-based versus Pull-based evaluation model . . . 7
2.2 Glitch avoidance . . . 9
2.3 Quadcopter configuration . . . 14
2.4 Quadcopter . . . 15
2.5 Flight controllers . . . 16
2.6 Pixhawk . . . 18
2.7 Mission planner . . . 20
2.8 APM Planner . . . 21
2.9 DroidPlanner 3 . . . 21
2.10 SITL simulator . . . 23

3.1 Auto flight mode . . . 32
3.2 RTL flight mode . . . 33

4.1 Propagation example . . . 42
4.2 Example solution . . . 42
4.3 Example of control loop . . . 44
4.4 Example of control loop with reactive library . . . 45
4.5 Arducopter . . . 47
4.6 Fast loop . . . 48
4.7 Sensors dependency . . . 50
4.8 Dependency tree between signals . . . 55

5.1 Flight plan . . . 61
5.2 Pitch, Roll and Yaw . . . 62
5.3 Error in yaw of the original Firmware . . . 62
5.4 Error in yaw of the reactive Firmware . . . 62
5.5 Error in yaw on original and reactive firmware . . . 64
5.6 Error in roll and error in pitch of the original firmware . . . 65
5.7 Error in roll and error in pitch of the reactive firmware . . . 66

5.8 Amplitude of the Roll and DesRoll parameters 67
5.9 Error in pitch on original and reactive firmware 68
5.10 Error in roll on original and reactive firmware 69
5.11 Altitude graph 70

List of Tables

5.1 Average and standard values of the error in yaw, first scenario. . . . 63
5.2 Average and standard values of the error in yaw, second scenario. . . 63
5.3 Average error in pitch and roll values 65
5.4 Standard deviation of pitch and roll parameters 66
5.5 Average error of the altitude of full flight plan 67
5.6 Average error of the altitude of modified flight plan 70
5.7 Main loop duration 71
5.8 Fast loop duration 72

List of Listings

2.1 Example reactive programming 6
2.2 Glitch avoidance example 8
2.3 Lift example 10
2.4 Start SITL 23
3.1 Main loop 27
3.2 Fast loop 28
3.3 Scheduler 30
3.4 Ten hz loop 31
4.1 ListOfFunctions.h 37
4.2 ListOfFunctions.cpp 38
4.3 ReactiveCpp.h 40
4.4 ReactiveCpp.cpp 41
4.5 Declaration of objects 49
4.6 Barometer and GPS reading 49
4.7 AHRS update 51
4.8 Attitude control 51
4.9 Inertial navigation and current location objects 52
4.10 Modified setup code 53
4.11 Method related to sensor check 56
4.12 Fast loop modified 56
4.13 GPS and barometer signals 58
5.1 DataFlash logs 71
5.2 Original ardupilot example 74
5.3 Reactive ardupilot example 75
6.1 Using C++11 standard 78

Chapter 1

Introduction

"Flying is learning how to throw yourself at the ground and miss."

Douglas Adams

1.1 General context

In recent years, we have seen a rapid growth of UAV's (Unmanned Aerial Vehicle) technology. Primarily this technology has only been used in the military field. Because of the creation of autopilots at a reasonable price and the fast growth of developer communities that are constantly developing drone firmwares, there was a big spread of this technology in the professional and hobbyist fields. A drone is controlled by a remote control (allowing pilots to control the vehicle manually) or by a ground control station (allowing pilots to create a flight plan that includes a set of waypoints and some flight modes). Some of the most used applications (using ground control station) include surveying, maintenance and surveillance tasks, transportation [27], search and rescue [2].

The two main components that make a drone fly are the autopilot and the firmware. Other components can be connected in order to permit the controlling vehicle (as the two control tools mentioned above) or the accurate detection of vehicle information (using additional sensors). In Figure 1.1 a simple configuration of a drone suite is shown. This configuration includes an autopilot with some components installed (RC receiver and GPS module), a radio control transmitter or a ground control station, a battery and finally the quadcopter which contains all these components except the control tools. In the autopilot is also installed the firmware which manages sensor readings and executes tasks that make the drone fly.

Ardupilot is an open source autopilot platform created from the DIY Drones community, able to control autonomous multicopters, fixed-wing aircraft and ground rovers. It is based on the Arduino platform. Its tools for altitude detection evolved from using thermopile technology to the use of Inertial Measurement Unit (IMU) that

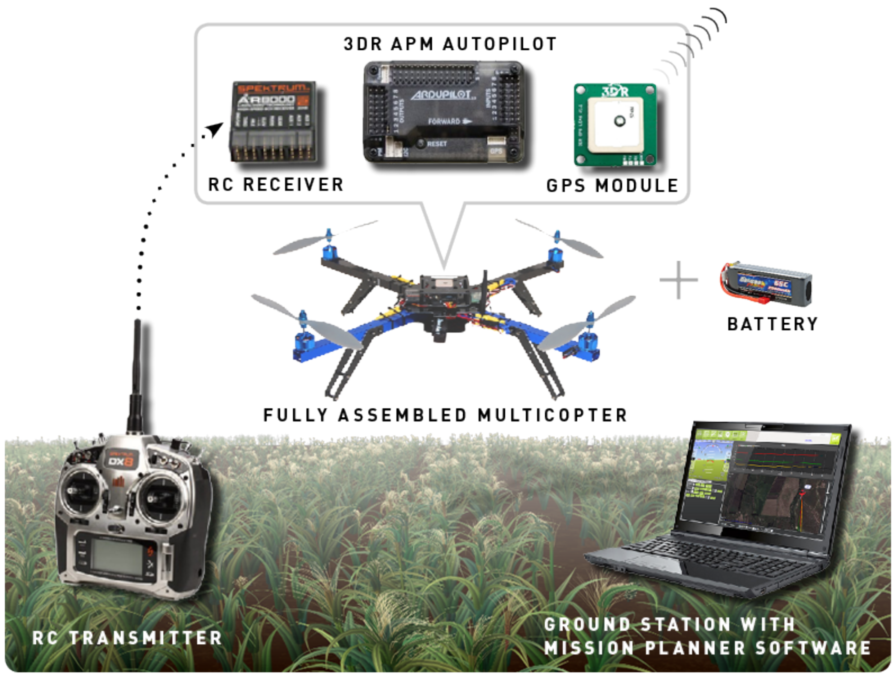

3DR APM AUTOPILOT
RC RECEIVER
GPS MODULE
BATTERY
FULLY ASSEMBLED MULTICOPTER
RC TRANSMITTER
GROUND STATION WITH MISSION PLANNER SOFTWARE

Figure 1.1: Drone configuration. *The main components are shown for a general overview [4].*

combine the accelerometer, gyroscope and magnetometer sensors. In addition other sensors can be installed such as GPS or sonar to provide more accurate information about position or altitude.

Referring to the firmware, a very important aspect is how the system manages these sensor values. We want to have vehicles that react immediately to new sensor readings. For example, when external factors are so strong as to affect the desired position, the system should notice it and adjust the vehicle position. Another fundamental aspect is that no computation must be done when the sensors do not detect new values (or at least the computation time must be decreased). Decreasing the computation time, the system has more time to execute other tasks. This implies an optimized use of battery (one of the limitations of drones is the autonomy, for a simple quadcopter the autonomy is about 20 minutes)

1.2 Motivation

The goal of this thesis is to modify an existing firmware (Ardupilot) using the reactive programming paradigm [5]. This paradigm consists in systems able to directly react to input changes or input environment changes in general. When a value changes, all the values depending on it are updated according to a dependency tree between variables or objects (starting from the sensor values). We can compare this paradigm with spreadsheet programs like Microsoft Excel. When a value changes, all the cells depending on that value directly change.

Reactive implementations of control loops, especially on a drone autopilot, are absent in the state of the art, although the reactive programming paradigm is taking more and more attention in the programming community. The motivation of this thesis is thus to apply it in the control loop of a drone to explore if it could bring to any sort of improvements.

First of all we are going to study the existing tools based on this paradigm. All these tools use libraries not suitable for Ardupilot due to hardware limitations such as the lack of multithreading or software limitations such as the lack of support for libraries like Boost or STL (both based on the latest standard of C++ not yet supported from Ardupilot). The only solution is to create a library of reactive programming from scratch (see ReactiveCpp in Section 4.1). Using this library we have modified the data flow that passes by the sensors up to the engines. The Ardupilot project supports a "pull-based" model in which data are pulled from sensors only when they are needed. With the reactive programming paradigm, Ardupilot is going to use a "push-based" model where data are pushed from sensors only when there are new values. The problem with the original solution is that the data request is done with a predetermined frequency, thus if one sensor does not change its state, the objects that require its values will get the same data and will recompute the same results.

The purpose of this thesis is to avoid unnecessary computation in case of unchanged sensor values. The data must be transmitted only when the sensors detect new data to avoid wasting time on unnecessary computation.

1.3 Outline

The first part of the implementation phase consists in the creation of a new library in C++ using reactive programming. The existing ones, such as TBB (Threading Building Blocks) and Boost , use libraries not suitable for Ardupilot firmware due to hardware limitations. They also use STL (Standard Template Library of C++) and C++11 standard that are not yet fully supported from Ardupilot. ReactiveCpp is one solution created for the implementation phase. It simply creates «signals» that have two lists of function pointers. When a signal is alerted, it executes all the functions on the first list (functions related to the signal value) and all the functions on the second list (functions that alert signals depending from it).

In the second part of implementation phase we present how this tool can be introduced in a control loop and then how we use it on Ardupilot project. We consider only the part of the project that refers to the data flow from sensors (gyroscope, magnetometer and barometer) to the main objects that make use of it, such as AHRS object (Attitude and Heading Reference System). We use the "pull-based" model to check if the sensors have detected new values, then with the reactive library (using "push-based" model) the propagation of new data takes place.

Finally we evaluate this custom firmware comparing its log files with those of the original firmware. We study the behavior of the vehicle in altitude, pitch, roll and yaw variables comparing the error (difference between the desired and real value of each parameter) related to these two generated flight logs. Another test consists in seeing how the code execution change from the original firmware to the custom one (time execution of the main loop). It is useful to understand if the reactive library has affected the time execution of the main loop after the modifications and see if the computation time is decreased.

From now on we give the general context and the general goals of this thesis with a brief description of this work.

In Chapter 2 there is a description of the actual state of the art related to this work. In Section 2.1 we describe what is the reactive programming paradigm and also we give a brief introduction of its properties and tools that make use of it are introduced. In Section 2.2 we describe in what the multicopter technology consists focusing the attention on the hardware, software and firmware that characterize a multicopter.

Chapter 3 presents the code structure of the Ardupilot firmware. It is divided in five subsections that explain how the main loop works (3.1.1), in what the fast loop consists (related to the critical part of the main loop, see Subsection 3.1.2), in what the scheduler consists and how it manages the tasks (3.1.3), an alternative way to run new code without using directly the scheduler (3.1.4) and eventually in the last Subsection (3.1.5) some of the most important flight modes used by a multicopter are shown.

Chapter 4 is focused on the main problem encountered during the implementation phase. It is divided in three sections. In Section 4.1 we show the main problem using the existing tools (4.1.1) and one possible solution (4.1.2) that consists on an implementation of an alternative reactive programming library in C++. In Section 4.2 is shown how this library can be used on a generic control loop and eventually in Section 4.3 it is shown how this library can be used on Ardupilot project.

Chapter 5 presents the evaluation phase. It describes the behavior of the system using the reactive programming library implemented in Section 4.1. To compare the original system with the modified one we compare two flight plans generated by the two systems. In Section 5.1 is shown how we compare the flight plans using only the simulator and the respective results. In Section 5.2 are compared two log files generated from tests done on physically drones. In Section 5.3 there is a summary of the entire evaluation chapter.

Finally, Chapter 6 draws the conclusion, recaps the results obtained and suggests some ideas that can be done as extensions of this work.

Chapter 2

State of the art

"I do not fear computers. I fear the lack of them."

Isaac Asimov

In this chapter we are going to present the state of the art related to this work focusing the attention on the two main topics, the reactive programming paradigm and the Unmanned Aerial Vehicle (UAV).

We have divided this chapter in two sections to have a clear overview about the two main topics. In the first topic we are going to describe in what reactive programming paradigm consists, some of the most important properties and a list of tools using this paradigm. In the second section we are going to describe what an UAV is and what is needed for a complete developing suite.

2.1 Reactive programming

Reactive programming describes systems that are reactive in the sense that they directly adapt to changing inputs or a changing input environment in general. If an input value is updated, reactive systems directly recompute the depending output values, so that the output always reflects the current input. We can compare it with spreadsheet programs like Microsoft Excel. When a value changes, all the cells depending on that value directly change. For years, implementing reactive systems, has been done with the observer pattern, which has several drawbacks (it is error-prone, update-triggering code is scattered throughout the system, it is not possible to compose different reactions, etc.). Therefore, better solutions, like event-driven programming (EDP) [9] or aspect-oriented programming (AOP) [14], have been developed. These solutions have their advantages over the observer pattern, but some problems still remain (e.g. dependencies must be manually encoded and event handlers have yet to implement updates explicitly). Using reactive programming, the output of the example in Listing 2.1 would be 10, because line 3 would be considered as a constraint instead of an assignment.

Listing 2.1 Example reactive programming

```
1 val a = 2
2 var b = 3
3 val c = a + b
4 b = 8
5 println(c)
```

Now we are going to describe the properties that constitute the taxonomy [5] of reactive programming paradigm along six axes:

1. basic abstractions
2. evaluation model
3. glitch avoidance
4. lifting operations
5. multidirectionality
6. support for distribution

These properties are discussed in detail below.

2.1.1 Basic abstraction

The basic abstractions in a reactive language are reactive primitives that facilitate the writing of reactive programs, just like primitive operators (assignments) and values (numbers) are basic abstractions in an imperative language. Most of reactive languages provide behaviors and events:

- In reactive programming literature, behavior is the term used to refer to time-varying values. It continuously changes over time and a basic example of a behavior is time itself. Most reactive programming languages express behavior in time units (a primitive seconds to represent the value of the current seconds of a minute or a behavior whose values is 2 times the current seconds is expressed in terms of seconds as seconds*2)
- On the other side, an event is the term used to refer to the streams of value changes. Events do not continuously change over time, they occur at discrete points in time (keyboard button press, new sensor values, updated values etc.). Those two basic abstractions can be seen as dual to each other and one can be used to represent the other [8].

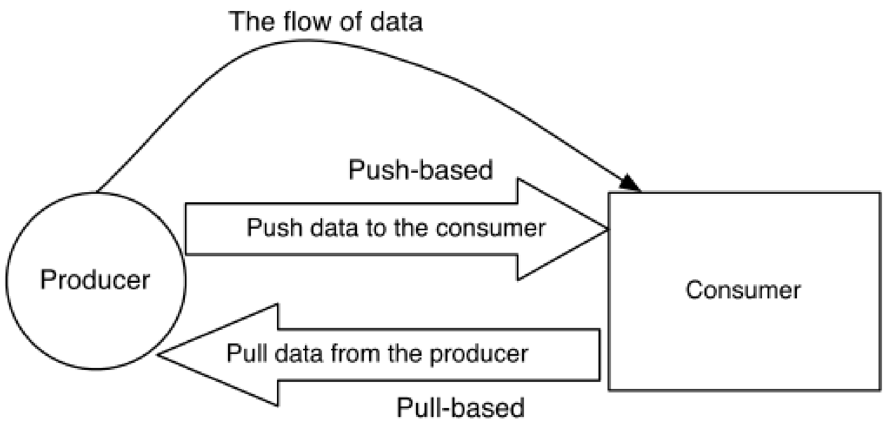


Figure 2.1: Push-based versus Pull-based evaluation model

Most languages provide primitive combinators to filter the events or just combine they. For example, a merge and filter combinators are provided by Flapjax [16] and FrTime [7].

2.1.2 Evaluation model

The evaluation model of a reactive programming language is concerned with how changes are propagated across a dependency graph of values. From the programmer's point of view, when a value changes, the propagation should happen automatically. When a change is verified, dependent computations must be notified. Therefore, the design decision is to choose which component initiates the propagation of change. In the reactive programming literature, there exists two evaluation models:

- Pull-based
- Push-based

2.1.2.1 Pull-based

In the pull-based model, the computation that requires a value need to «pull» it from the source (consumer should pull data from the producer, Figure 2.1). The first implementation of reactive programming languages such as Fran [11] use this evaluation model. It offers the flexibility that the computation requiring that value has the liberty to only pull the new values when it actually needs it. So the propagation is driven by the demand of new data (demand-driven).

2.1.2.2 Push-based

In the push-based model, when new data are available, the source «push» the data to its dependent computation (producer should push data to the consumer, Figure 2.1). The propagation is driven by availability of new data (data-driven). This approach is used by almost all implementations of reactive programming. This usually involves calling a registered callback or a method [23]. A push-based model is used by the most implementations of reactive programming such as Flapjax [16], Scala.React [15] and FrTime [7].

2.1.2.3 Push versus Pull

Each of those two models has its advantages and disadvantages. The pull-based model fits well in reactive systems where sampling is done on event values that change continuously over time. The push-based model works well in reactive systems that require instantaneous reactions. Most of the existing tool in reactive programming use only one of those two models and some other tools use both. In the pull-based model, behavior need not to be initialised, it will be initialised and then recomputed on demand. In the other model, the push-based one, the behaviors must be initialised explicitly to make sure they hold a value.

2.1.3 Glitch avoidance

This property is another one that needs to be considered by a reactive language. A glitch is verified when a value is propagated but it is not a «fresh» value. For example, when a computation is run before all its dependent expressions are evaluated [7]. Lets consider the example in Listing 2.2. The value of `var2` is expected to be always as that of `var1`. The value of `var3` is expected to be always as `var1` + `var2` (the twice of `var1`). If the value of `var1` changes to 2, the expected value of `var2` is 2 and the expected value of `var3` is 4. However, in a reactive programming implementation that does not consider the glitch avoidance, in a first instance the value of `var2` will be changed on 2 and the value of `var3` in 3. It will be necessary another computation to calculate the final value of `var3` (from `var1` and the new value of `var2`). This example is shown in Figure 2.2.

Listing 2.2 Glitch avoidance example

```
var1 = 1
var2 = var1 * 1
var3 = var1 + var2
```

In the reactive programming literature, this inconsistent view of data is known as a glitch [7]. This can happen only in languages using a push-based evaluation

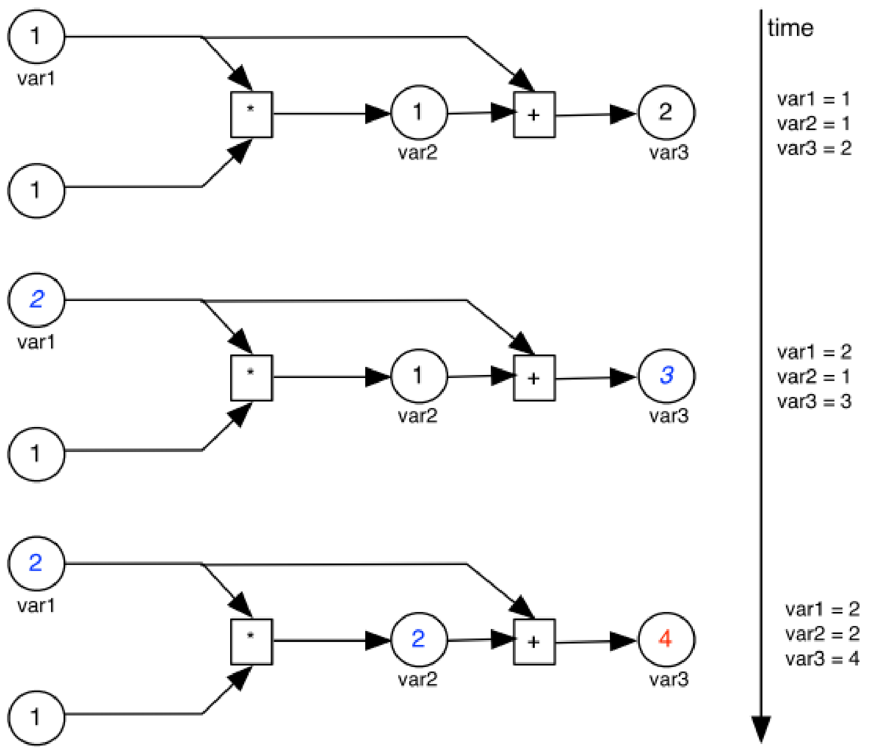


Figure 2.2: Glitch avoidance

model.

Another important aspect of this property is that a reactive implementation should avoid unnecessary recomputations of values that do not change. Referring to the example in Figure 2.2, if var1 is updated to the same value (var1 = 1), the values of var2 and var3 do not need to be recomputed.

2.1.4 Lifting operations

In the reactive programming literature the conversion of an ordinary operator to a variant that can operate on behavior is known as lifting. It transforms a function in reactive function and it registers a dependency graph in the application's dataflow graph. The reactive functions can be applied to reactive values and will return a reactive variable.

Listing 2.3 Lift example

```
lift : f(T) -> fl(Behavior<T>)

fl(Behavior<T>) -> f(Ti)
```

In Listing 2.3, «T» is a non behavior type while «Behavior» is a behavior type holding values of type «T». So, lifting an operator «f» that was defined to operate on a non-behavior value transforms it into a lifted version «fl» that can be applied on a behavior.

2.1.5 Multidirectionality

This property gives us the direction of change's propagation. It can be unidirectional when the propagation happens in one direction or multidirectional when it happens in more directions. With multidirectionality, changes in derived values are propagated back to the values from which they were derived. For example, if we have two variables that depend on each other, when one of those variables change, the other changes too. This property is similar to the multidirectional constraints in the constraint programming paradigm [25].

2.1.6 Support for distribution

Recently, many interactive applications (e.g., Web applications and mobile applications) are becoming increasingly distributed. It has motivated the need for support for distribution in a reactive language. It enables to create dependencies between computations that are distributed on multiple nodes. For example, a value depends by the sum of other two values situated in two different nodes. However, the main characteristics of distributed programming such as latency and network failures, make difficult to ensure consistency in a dependency graph based on more nodes. Only a few tools like Flapjax [16] support this property. However, in these languages glitches are avoided only locally.

2.1.7 Tools

Here we are going to give a brief summary of the libraries on C++ language using reactive programming. We are focusing only to C++ language because it is what we are going to use in implementation phase. The three main libraries are:

- cpp.react [21]
- SodiumFRP [22]
- RxCpp [17]

C++React (cpp.react) is a reactive programming library for C++11. It provides abstractions to handle change propagation and data processing for a push-based event model. It enables coordinated, multi-layered and potentially parallel execution of callbacks. All this happens implicitly, based on declarative definitions, with guarantees regarding

- update minimality - nothing is re-calculated or processed unnecessarily;
- glitch freedom - no transiently inconsistent data sets;
- thread safety - no data races for parallel execution by avoiding side effects.

The core abstraction of the library are

- signals, reactive variables that are automatically recalculated when their dependencies change
- event streams as composable first class objects.

Signals specifically deal with aspects of time-varying state, whereas event streams facilitate event processing in general.

Additional features include

- a publish/subscribe mechanism for callbacks with side effects;
- a set of operations and algorithms to combine signals and events;
- a domain model to encapsulate multiple reactive systems;
- transactions to group related events, supporting both synchronous and asynchronous execution.

RxCpp is a C++ implementation of the Reactive Extensions library. It is a branch of Reactive Extensions (Rx) implemented from Microsoft. The Reactive Extensions can be thought of as an asynchronous algorithm and collection library. Instead of using "std::iterator" pairs, RxCpp uses observable as the collection interface and observer as the iterator interface.

SodiumFRP is implemented in C#, C++, Java, Haskell and Scala (other languages are going to be added). This library is based on Flapjax [16], Yampa [13], scala.React [15] and a number of other Functional Reactive Programming tools. Some features are:

- Goals include simplicity and completeness.
- Applicative style: Event implements Functor and Behavior implements Applicative.
- Instead of the common approach where inputs are fed into the front of a monolithic 'reactimate', Sodium allows you to push inputs in from scattered places in IO.
- Integration with IO: Extensible to provide lots of scope for lifting IO into FRP logic.
- Push-based imperative implementation.

2.2 Multicopters

A multicopter is a mechanically simple aerial vehicle whose motion is controlled by speeding or slowing multiple downward thrusting motor/propeller units. In this section we are going to introduce the UAV technology.

2.2.1 Overview

Drones are more formally known as Unmanned Aerial Vehicles (UAV). Essentially, a drone is a flying robot. The aircraft may be remotely controlled or can fly autonomously through software-controlled flight plans (GCS) in their embedded systems working in conjunction with GPS. UAVs have most often been associated with the military but they are also used for search and rescue, surveillance, traffic monitoring, weather monitoring and firefighting, among other things.

More recently, the unmanned aircraft have come into consideration for a number of commercial applications. In 2013, Amazon announced a plan to use drones for delivery in the not too distant future. The service, known as Amazon Prime Air, is expected to deliver orders inside a 10-mile radius of a fulfillment center within 30 minutes.

In late 2012 Chris Anderson, Editor-In-Chief of Wired magazine [28], retired to dedicate himself to his personal drones company, 3D Robotics [1]. Personal drones are currently a hobbyist's item most often used for aerial photography, but the market and potential applications are both expected to expand rapidly.

The drones are classified in three categories

- Copters
- Planes
- Rovers

In this thesis we are going to refer only to UAVs (Copter category).

Copter is capable of the full range of flight requirements from fast paced FPV (First Person View) racing to smooth aerial photography to fully autonomous complex missions which can be programmed through one of the well-developed software ground stations. The entire package is designed to be safe, feature rich, open-ended for custom applications and increasingly easy to use even for novice users. There are some types of copters classified from their number of motors/propellers (Helicopter, Tricopter, Quadcopter, Hexacopter and Octacopter). Here we are going to describe the characteristics of a quadcopter that we are going to use later on the implementation and test phases.

The main characteristics are:

- Utilize differential thrust management of independent motor-prop units to provide lift and directional control
- Benefit from mechanical simplicity and design flexibility
- A capable payload lifter that's functional in strong wind conditions
- Redundant lift sources can give increased margin of safety
- Varied form factor allows convenient options for payload mounting.

There are two types of configuration for the quadcopter. The “X” configuration and the “+” configuration as shown on Figure 2.3. Intuitively, in the “X” configuration, the pitch and roll axes both have two counter-rotating propellers each, on each side. In the '+' configuration the quadcopter has just one on each side. Moving (pitching) forward in this configuration will leave the vehicle with lesser stability in the roll axis. From now on we will consider only the “X” configuration.

The quadcopter in Figure 2.4 is the simplest type of multicopter, with each motor/propeller spinning in the opposite direction from the two motors on either side of it (i.e. motors on opposite corners of the frame spin in the same direction).

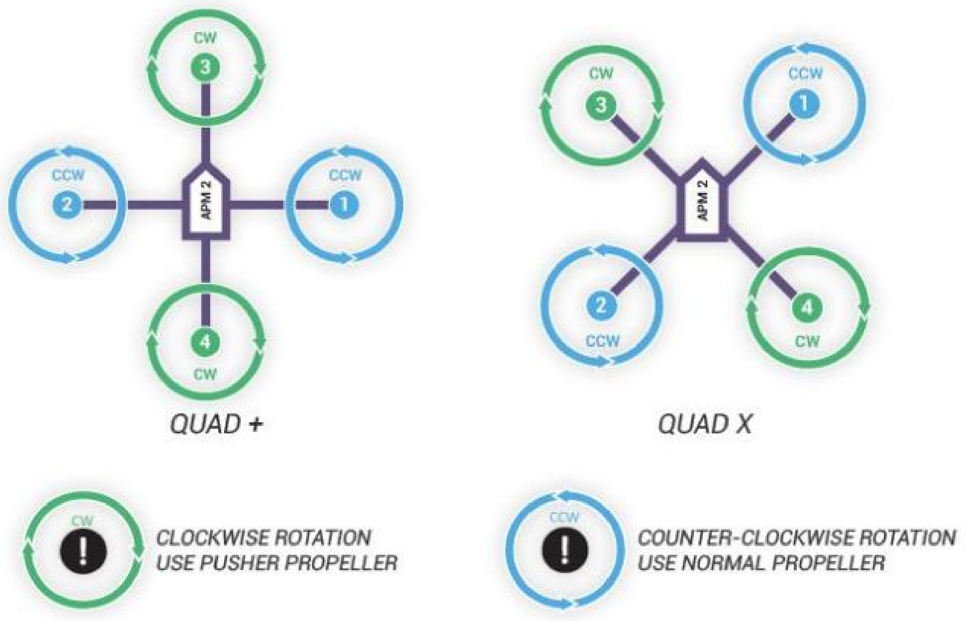


Figure 2.3: Quadcopter configuration

A quad copter can control it's roll and pitch rotation by speeding up two motors on one side and slowing down the other two. So for example if the quad copter wants to roll left it would speed up motors on the right side of the frame and slow down the two on the left. Similarly if it wants to rotate forward it speeds up the back two motors and slows down the front two. The quadcopter turns (aka "yaw") left or right by speeding up two motors that are diagonally across from each other, and slowing down the other two.

Horizontal motion is accomplished by temporarily speeding up/slowing down some motors so that the vehicle is leaning in the direction of desired travel and increasing the overall thrust of all motors so the vehicle shoots forward. Generally the more the vehicle leans, the faster it travels.

Altitude is controlled by speeding up or slowing down all motors at the same time.

2.2.2 Autopilot suite

Here in this subsection we are going to introduce two components that make a UAV fly and other two optional components that helps on controlling the vehicle or simulating a flight plan without the hardware. In Subsection 2.2.2.1 are listed three of the most common flight controllers. In Subsection 2.2.2.2 we are going to present the Ardupilot firmware. In Subsection 2.2.2.3 are listed three Ground Control

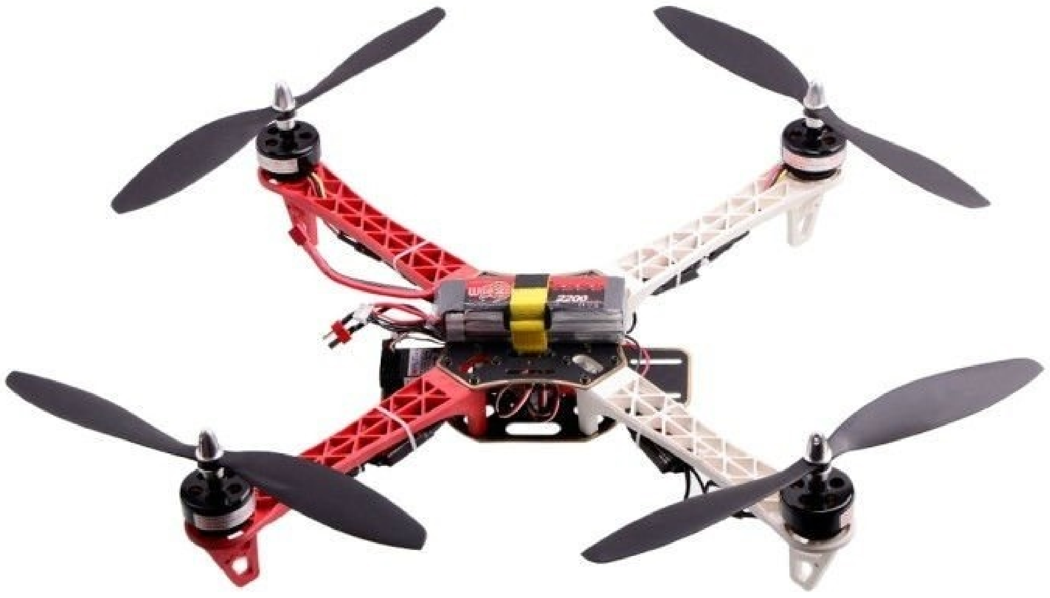


Figure 2.4: Quadcopter

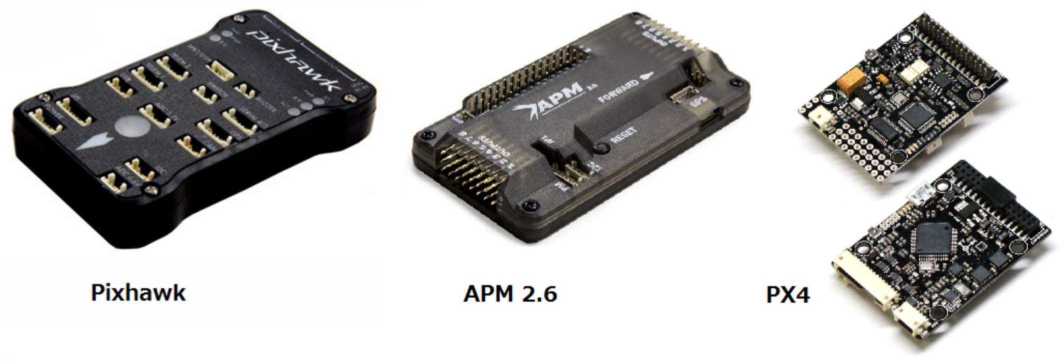


Figure 2.5: Flight controllers

Stations (GCS) and in the last one (Subsection 2.2.2.4) we are going to introduce the simulator (SITL), very useful for who want to test the firmware without the hardware.

2.2.2.1 Hardware

There are three main autopilots (Pixhawk, APM2.6 and PX4, all sold by 3DRobotics [1]) that run the Ardupilot software (Figure 2.5). In addition there are clones of those boards (like HKPilot32 [12], a clone of Pixhawk). Here are the summary of those three best choices.

Although these boards have different CPU performance and different sensors, the user experience is almost identical and supported features are still very similar. The best flight controller is the Pixhawk. This board has an ARM CPU and it is based on the earlier PX4 with more enhancements. Pixhawk is the latest and most advanced of the three boards, with the fastest CPU, most RAM, backup acceleormeters and gyros, and supports backup compass and GPS.

The APM2.6 is the final edition of the traditional AVR CPU based ardupilot flight controllers. The APM was one of the first flight controllers and now with the version 2.6 was optimized to the fullest to bring us the rich capabilities that got us so far. Unfortunately the ArduCopter firmware now consumes all the available 8 bit AVR memory and CPU performance (especially for multicopters with more than four motors from ArduCopter 3.3 and later), so we can no longer add additional enhancements.

The PX4 was the first ARM based board to run multicopters. It was developed, like the Pixhawk, from the PX4 team. It has slower CPU and less RAM than the Pixhawk and its only advantage is the price and its small size.

Pixhawk: This board (Figure 2.6) is an advanced autopilot system designed by the PX4 open-hardware project and manufactured by 3D Robotics. It features advanced processor and sensor technology from ST Microelectronics® and a NuttX real-time operating system, delivering incredible performance, flexibility, and reliability for controlling any autonomous vehicle. The benefits of the Pixhawk system include integrated multithreading, a Unix/Linux-like programming environment, completely new autopilot functions such as sophisticated scripting of missions and flight behavior, and a custom PX4 driver layer ensuring tight timing across all processes. These advanced capabilities ensure that there are no limitations to your autonomous vehicle. Pixhawk module is accompanied by new peripheral options, including a digital airspeed sensor, support for an external multi-color LED indicator and an external magnetometer. All peripherals are automatically detected and configured.

Key features

- Advanced 32 bit ARM Cortex® M4 Processor running NuttX RTOS
- 14 PWM/servo outputs (8 with failsafe and manual override, 6 auxiliary, high-power compatible)
- Abundant connectivity options for additional peripherals (UART, I2C, CAN)
- Integrated backup system for in-flight recovery and manual override with dedicated processor and stand-alone power supply
- Backup system integrates mixing, providing consistent autopilot and manual override mixing modes
- Redundant power supply inputs and automatic failover
- External safety button for easy motor activation
- Multicolor LED indicator
- High-power, multi-tone piezo audio indicator
- microSD card for long-time high-rate logging

Microprocessor:

- 32-bit STM32F427 Cortex M4 core with FPU
- 168 MHz/256 KB RAM/2 MB Flash
- 32 bit STM32F103 failsafe co-processor

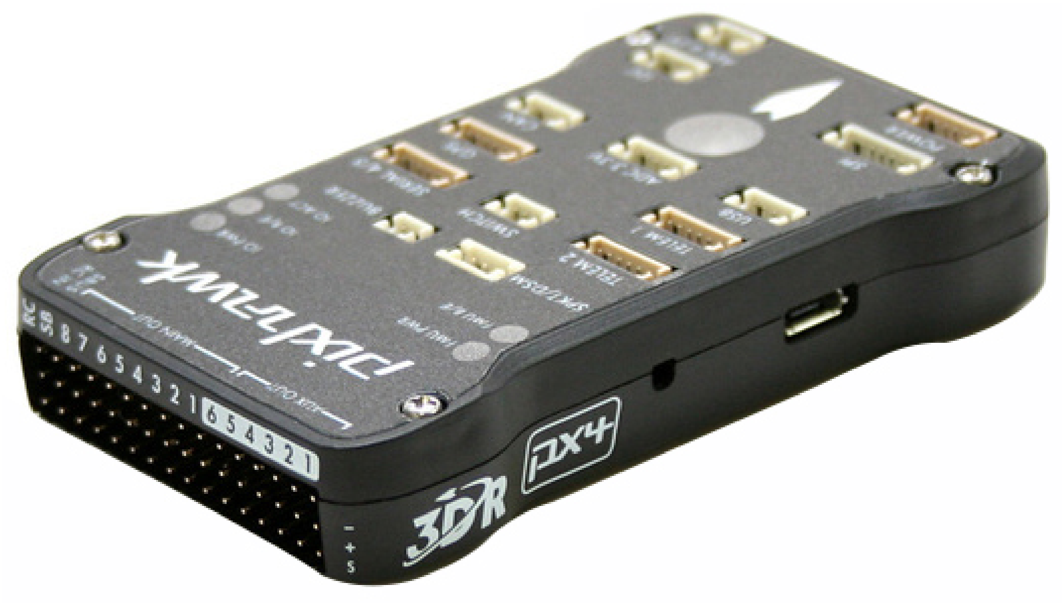


Figure 2.6: Pixhawk

Sensors:

- ST Micro L3GD20 3-axis 16-bit gyroscope
- ST Micro LSM303D 3-axis 14-bit accelerometer / magnetometer
- Invensense MPU 6000 3-axis accelerometer/gyroscope
- MEAS MS5611 barometer

2.2.2.2 Firmware

Ardupilot (also ArduPilotMega - APM [3]) is an open source unmanned aerial vehicle (UAV) platform, able to control autonomous multicopters, fixed-wing aircraft, traditional helicopters and ground rovers. It was created in 2007 by the DIY Drones community [10]. It is based on the Arduino open-source electronics prototyping platform. The first Ardupilot version was based on a thermopile, which relies on determining the location of the horizon relative to the aircraft by measuring the difference in temperature between the sky and the ground. Later, the system was improved to replace thermopiles with an Inertial Measurement Unit (IMU) using a combination of accelerometers, gyroscopes and magnetometers.

Today, the ArduPilot project has evolved to a range of hardware and software products, including the APM, Pixhawk/PX4 and HKPilot32 line of autopilots, and the ArduCopter, ArduPlane and ArduRover software projects. In this thesis we are going to use a Pixhawk board and ArduCopter software project.

The free software approach from Ardupilot is similar to that of the PX4/Pixhawk and Paparazzi Project [19], where low cost and availability enables its hobbyist use in small remotely piloted aircraft, such as micro air vehicles and miniature UAVs.

The customizability of ardupilot allows it to be very popular in the DIY field. This allows for a multitude of uses such as multicopter and fixed plane drones. This customizability also allows a variety of additional parts to be used by the use of different connectors and transmitters to allow for different uses depending on the operator preferences. The customizability and ease of installation has allowed the Ardupilot platform to be integrated for a variety of missions. The use of a ground station control (GCS like Mission Planner) has allowed the Ardupilot board to be used for mapping missions, search and rescue, and surveying areas.

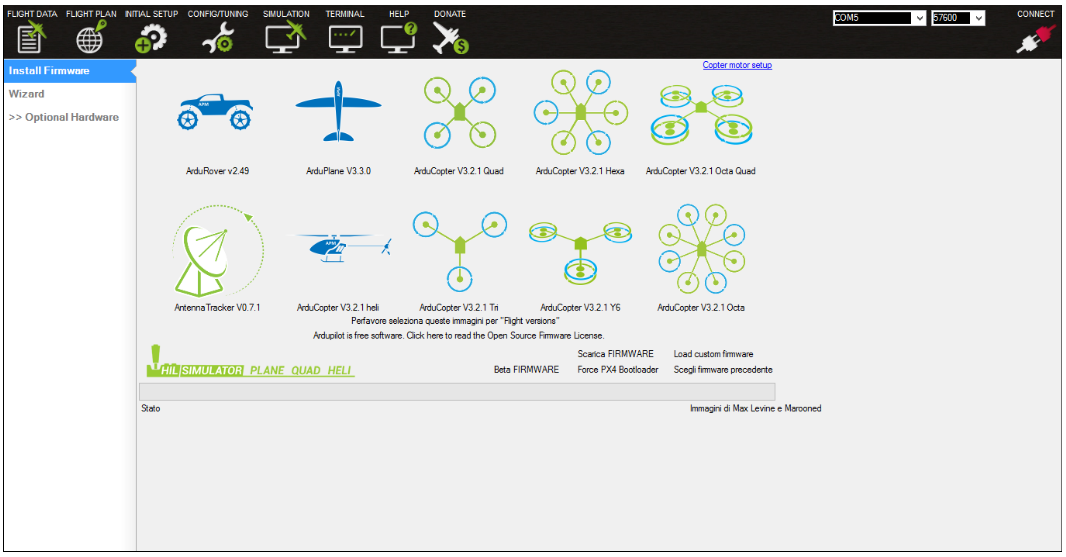
2.2.2.3 GCS (Software)

A Ground Control Station is an operator control unit for unmanned aerial vehicle. It is defined as a unit where an operator (pilot) can send and receive instruction to one or more vehicles that have been deployed and to visualize and control the vehicle during development and operation, both indoors and outdoors. With a flexible software architecture it supports multiple UAV types/autopilot projects. The purpose of the Ground Control Station is real-time monitoring of an UAV.

Those are some of the most used Ground Control Station:

Mission Planner is a fully featured GCS running on the Windows Operating System (Figure 2.7). Its features include:

- Configure APM/Pixhawk/HKPilot32 settings for UAV
- Radio control accelerometer and gyroscope calibration
- Install Rover/Plane/Copter firmware (original firmware or custom ones)
- Connection with the vehicle through telemetry
- Creations of flight plans (a set of waypoints and flight modes)
- Point-and-click waypoint entry, using Google Maps/Bing/Open street maps/-Custom WMS (Web Map Service)
- Select mission commands from drop-down menus
- Download mission log files and analyze them
- Interface with a PC flight simulator to create a full hardware-in-the-loop UAV simulator
- In flight HUD (Head-Up Display) view



Perfavore seleziona queste immagini per "Flight versions"

Ardupilot is free software. Click here to read the Open Source Firmware License.

Immagini di Max Levine e Marooned

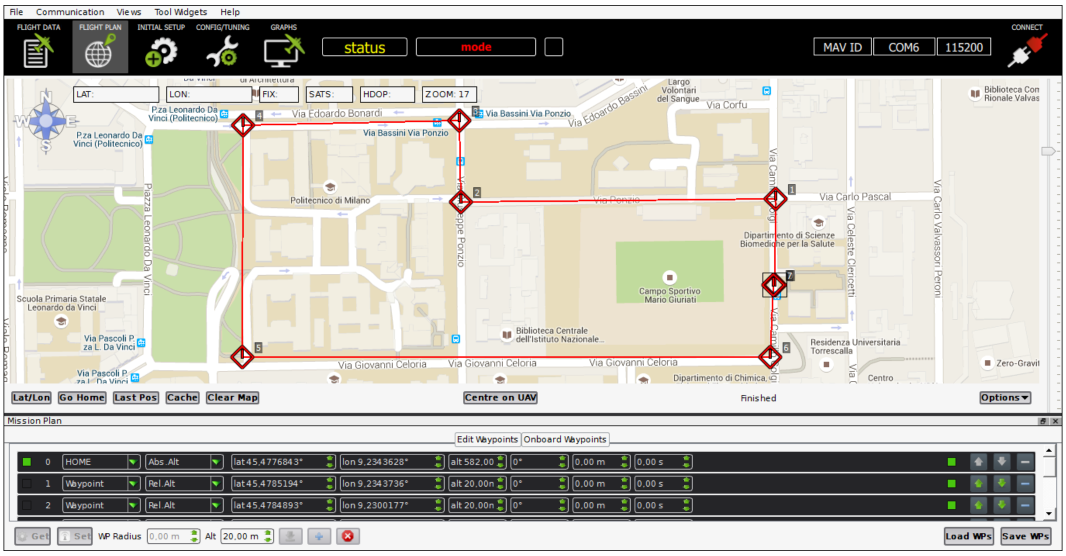
Figure 2.7: Mission planner

APM Planner is a multi OS fully featured GCS. It runs on MAC OS X, Windows and Linux (Figure 2.8). Its features include:

- Configure and calibrate APM /Pixhawk/HKPilot32 autopilot for autonomous vehicle control
- Upload the latest firmware to the autopilot
- Plan a mission with GPS waypoints and control events
- Connect a Radio to view live data and initiate commands in flight
- View vehicle status and flight data using the head-up display (HUD) area of the Flight Data screen

Tower (DroidPlanner 3) is a GCS for the Android OS 4.0 and above (Figure 2.9). Its features include:

- Specifically designed for multirotors
- New telemetry screen showing quick glanceable info: HUD, battery, RSSI, distance
- Easy to use Home, Land, and Loiter buttons
- New guided mode with changeable altitude
- Quick mode switching
- New planning screen for quick mission generation



0	HOME	Abs.Alt	lat45,4776843°	lon 9,2343628°	alt 582,00	0°	0,00 m	0,00 s
1	Waypoint	Rel.Alt	lat45,4785194°	lon 9,2343736°	alt 20,00n	0°	0,00 m	0,00 s
2	Waypoint	Rel.Alt	lat45,4784893°	lon 9,2300177°	alt 20,00n	0°	0,00 m	0,00 s

WP Radius 0,00 m Alt 20,00 m

Figure 2.8: APM Planner

- Easy and powerful mission editing tools
- Basic radio TX setup
- Preflight checklist

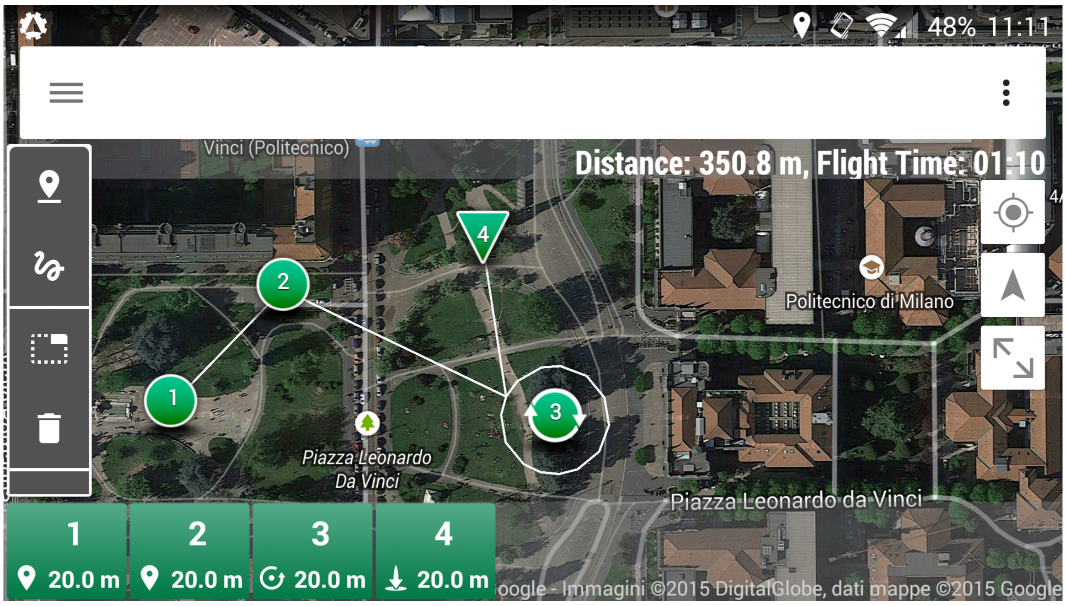


Figure 2.9: DroidPlanner 3

In the implementation and test phases we will use these GCS (Mission Planner and APM Planner) to load our custom firmware, to calibrate the compass, accelerometer and radio control and to open the log files of each flight. we will never use Tower, however we describe it because is one of the most used for Android platform.

2.2.2.4 SITL Simulator

The SITL (Software In The Loop) simulator allows us to run Ardupilot without any hardware. It is a build of the autopilot code using an ordinary C++ compiler, giving us a native executable that allows to test the behavior of the code without hardware.

HITL (Hardware In The Loop) simulation is a very useful way of testing the Ardupilot code, but it has a number of limitations that make it less suitable for some tasks. The main limitations are:

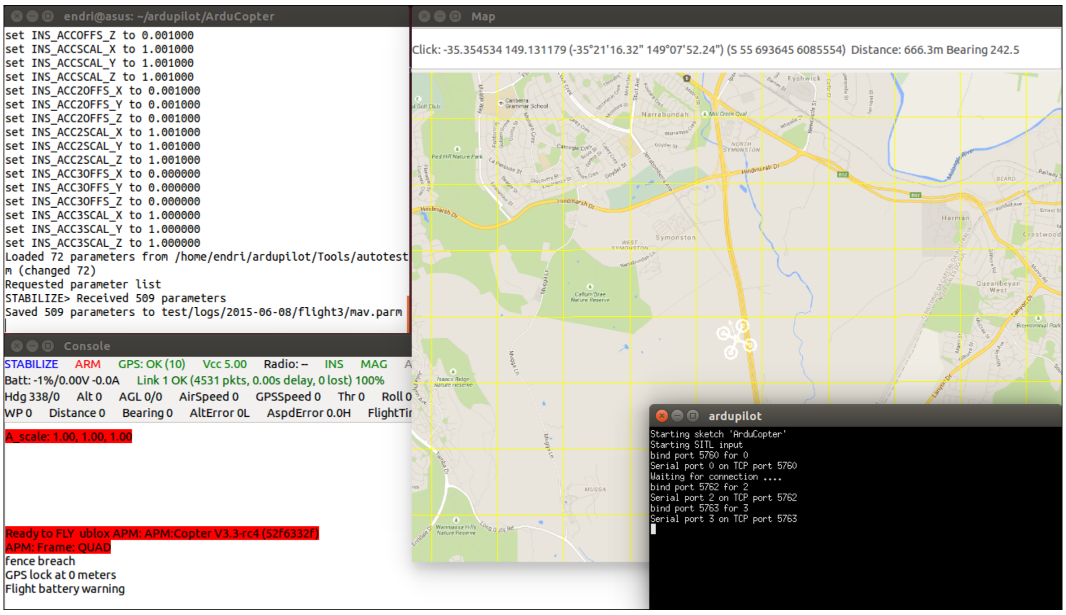
- it can not run all of the autopilot code, as the low level driver code does not see suitable inputs for a test flight when the hardware is sitting on the desk
- we can not use the sort of advanced programming tools (such as debuggers and memory checkers) that are so useful in normal C++ development

The SITL build of ArduPilot overcomes these limitations. It still runs the same code, but this time as a native executable on our PC, and uses some C++ tricks to emulate the hardware of the APM/Pixhawk board at the register level, so the key low level hardware drivers (such as the ADC, gyros, accelerometers and GPS) all run in the same way that they would run in a real flight. In Figure 2.10 is shown a screenshot of this simulator. It is composed by three windows:

- MAVProxy Command Prompt
- Console
- Map

The SITL arrange these three tools so we can observe the status and send commands at the same time to the vehicle.

To start this simulator, the user must be on the ardupilot/ArduCopter folder (referring to the copter vehicle) of the ardupilot project. An example of commands for the SITL are shown on Listing 2.4. In line 1, `sim_vehicle` sets all the parameters of the copter and than open the console and map window where the flight is shown. In line 3 it is loaded the flight plan (a set of way points that composte the flight plan). From line 5 to 8 are shown the commands given to the vehicle to start the flight (arming throttle, then passing to auto mode with throttle at 1500).



```
set INS_ACCOFFS_Z to 0.001000
set INS_ACCSCAL_X to 1.001000
set INS_ACCSCAL_Y to 1.001000
set INS_ACCSCAL_Z to 1.001000
set INS_ACC2OFFS_X to 0.001000
set INS_ACC2OFFS_Y to 0.001000
set INS_ACC2OFFS_Z to 0.001000
set INS_ACC2SCAL_X to 1.001000
set INS_ACC2SCAL_Y to 1.001000
set INS_ACC2SCAL_Z to 1.001000
set INS_ACC3OFFS_X to 0.000000
set INS_ACC3OFFS_Y to 0.000000
set INS_ACC3OFFS_Z to 0.000000
set INS_ACC3SCAL_X to 1.000000
set INS_ACC3SCAL_Y to 1.000000
set INS_ACC3SCAL_Z to 1.000000
Loaded 72 parameters from /home/endri/ardupilot/Tools/autotest
m (changed 72)
Requested parameter list
STABILIZE> Received 509 parameters
Saved 509 parameters to test/logs/2015-06-08/flight3/mav.parm
```

```
STABILIZE    ARM    GPS: OK (10)    Vcc 5.00    Radio: --    INS    MAG    A
Batt: -1%/0.00V -0.0A    Link 1 OK (4531 pkts, 0.00s delay, 0 lost) 100%
Hdg 338/0    Alt 0    AGL 0/0    AirSpeed 0    GPSSpeed 0    Thr 0    Roll 0
WP 0    Distance 0    Bearing 0    AltError 0L    AspdError 0.0H    FlightTi

A_scale: 1.00, 1.00, 1.00

Ready to FLY  ublox APM: APM:Copter V3.3-rc4 (52f6332f)
APM: Frame: QUAD
fence breach
GPS lock at 0 meters
Flight battery warning
```

```
Click: -35.354534 149.131179 (-35°21'16.32" 149°07'52.24") (S 55 693645 6085554)  Distance: 666.3m Bearing 242.5
```

```
Starting sketch 'ArduCopter'
Starting SITL input
bind port 5760 for 0
Serial port 0 on TCP port 5760
Waiting for connection ....
bind port 5762 for 2
Serial port 2 on TCP port 5762
bind port 5763 for 3
Serial port 3 on TCP port 5763
```

Figure 2.10: SITL simulator

Listing 2.4 Start SITL

```
sim_vehicle.sh -w --console --map --aircraft test

wp load ../Tools/autotest/copter_mission.txt

level
arm throttle
auto
rc 3 1500
```

Chapter 3

Analysis of Ardupilot Implementation

3.1 Code structure

In this chapter we are going to describe how the ArduCopter code runs. It is divided in five parts which we believe are useful to understanding the structure of the project and are fundamental to the implementation of a modified firmware using a library based on the reactive programming paradigm. In Subsections 3.1.1, 3.1.2 and 3.1.3 are shown the three most important parts of the firmware (the main loop, the fast loop and the scheduler) and described how they work. Subsection 3.1.4 presents an alternative mode to run a new code on Ardupilot project (instead of using main loop, fast loop or directly the scheduler). In Subsection 3.1.5 are described the main flight modes that we are going to use later on test phase.

The basic structure of ArduPilot is divided into 5 main parts:

- vehicle directories - are the top level directories that define the firmware for each vehicle type (Plane, Copter, Rover and AntennaTracker).
- AP_HAL - is a hardware abstraction layer for the ArduPilot project. The AP_HAL consists of a set of headers (.h) that define the classes and methods that should be implemented if ardupilot should run in a new device/architecture. The code contained in HAL is usually quite low level and close to the hardware.
- libraries - all the set of libraries used by the project (it includes the AP_HAL library).
- tools directories - is refereed to script files that permit to configure the computer for the ardupilot project, that permit to start the simulator or to set parameters

(like the home position and the set of waypoints nedded to the SITL simulator etc.) .

- external support code - on some platforms we need external support code to provide additional features or board support. Currently some of the external trees are PX4NuttX (the core NuttX [18] RTOS used on PX4 boards) and PX4Firmware (the base PX4 middleware and drivers used on PX4 boards).

In this chapter we refer only to copters code that is the ArduCopter directory of the ardupilot project.

3.1.1 Main loop

ArduCopter.pde is the main file of the ArduCopter project. It starts by running two subroutines, setup() and loop().

- setup() - called once at boot
- loop() called continuously

The main loop cycles in a continuous manner enabling the software to read the input and to respond appropriately. In this loop (Listing 3.1) is called the fast_loop() function and the scheduler. The main loop manages the time to give to the scheduler. At first is calculated how much time the fast_loop takes, then, after fast_loop ends, the available time is given to the scheduler that executes all the possible tasks (a task is a function that must be called at a given frequency). It also updates a counter (mainLoop_count) that is used to control how often different loops are executed.

3.1.2 Fast loop

The fast_loop() is the critical function of the main code. In Listing 3.2 we can find all the instructions that must be called at the highest priority. The fast_loop is directly called from the loop and not from the scheduler because it guarantees the execution in each cycle.

3.1.3 Scheduler

The most flexible way to run a function at a given interval is to use the scheduler. So, creating a function to ArduCopter.pde, we can simply add it to the scheduler_tasks array (an array that contains a list of tasks that are going to be called from the scheduler). There are two types of task lists. The first list is for slow CPUs like APM2 and the other list is for high speed CPUs like Pixhawk. In Listing 3.3 it is shown how the scheduler is initialized on the project. Here we can find all the tasks except the fast_loop. If there is enough time to execute a set of tasks, they will be

Listing 3.1 Main loop

```
// Main loop
void loop(){
        // wait for an INS sample
        ins.wait_for_sample();

        uint32_t timer = micros();
        // check loop time
        perf_info_check_loop_time(timer - fast_loopTimer);

        // used by PI Loops
        G_Dt = (float)(timer - fast_loopTimer) / 1000000.0f;
        fast_loopTimer = timer;

        // for mainloop failure monitoring
        mainLoop_count++;

        // Execute the fast loop
        fast_loop();

        // tell the scheduler one tick has passed
        scheduler.tick();

        // run all the tasks that are due to run. Note that we
        // only have to call this once per loop, as the tasks
        // are scheduled in multiples of the main loop tick.
        // So if they don't run on the first call to the scheduler
        // they won't run on a later call until scheduler.tick()
        // is called again
        time_available = (timer + MAIN_LOOP_MICROS) - micros();
        scheduler.run(time_available);
}
```

Listing 3.2 Fast loop

```
// Fast loop
static void fast_loop() {

        // IMU DCM Algorithm
        read_AHRS();

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

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

        // Inertial Nav
        read_inertia();

        // run the attitude controllers
        update_flight_mode();

        // update home from EKF if necessary
        update_home_from_EKF();

        // check if we've landed
        update_land_detector();
}
```

executed. The set of tasks will be chosen by the scheduler considering their priority and frequency of execution.

Each row is composed by three parameters. The first parameter is the task name. The second gives us the frequency in which the task should be called (in 2.5ms units in our case using Pixhawk, in 10ms units in case of APM2). If we need to call a function at 50hz, we must set this parameter to 8 (400hz/50hz), to call a function at 400hz we must set it at 1. The third parameter is the max time beyond which the tasks should not run (value in microseconds). This parameter helps the scheduler to avoid making the call unless there is enough time left to run the task. When scheduler.run() is called it is passed the amount of time (in microseconds) available for running tasks, and if the worst case time for this task would mean it would not fit before that time runs out then it will not be called.

The tasks in AP_Scheduler tables must have the following attributes.

- they should never block
- they should never call sleep functions when flying (an autopilot, like a real pilot, should never sleep while flying)
- they should have predictable worst case timing

3.1.4 Running new code

Another way to run a new code is to add the function in one of the existing timed loops. It can be useful when the programmer wants to add more tasks that maybe run at the same frequency. In this way it can be avoided to create other rows on the scheduler_tasks list. Those functions listed below are called at the right frequency by the scheduler except the fast_loop (called directly by the main loop).

- fast_loop : runs at 400hz (called from main_loop)
- fifty_hz_logging_loop : runs at 50hz (called from the scheduler in line 38 of Listing 3.3)
- ten_hz_logging_loop: runs at 10hz (called from the scheduler in line 37)
- three_hz_loop: runs at 3.3hz (called from the scheduler in line 24)
- one_hz_loop : runs at 1hz (called from the scheduler in line 28)

There is no real advantage to this approach over the above approach except in the case of the fast_loop. Adding the function to the fast_loop will mean that it runs at the highest possible priority (i.e. it is nearly 100% guaranteed to run at 400hz).

If we want our new code to run at 10hz we could add it to one of the case statements in the ten_hz_logging_loop() function found in ArduCopter.pde (Listing 3.4).

Listing 3.3 Scheduler

```
/*
1 = 400hz
2 = 200hz
4 = 100hz
8 = 50hz
20 = 20hz
40 = 10hz
133 = 3hz
400 = 1hz
4000 = 0.1hz
*/

static const AP_Scheduler::Task scheduler_tasks[] PROGMEM = {
{ rc_loop,                  4,     10 },
{ throttle_loop,            8,     45 },
{ update_GPS,               8,     90 },
{ update_batt_compass,     40,     72 },
{ read_aux_switches,       40,      5 },
{ arm_motors_check,        40,      1 },
{ auto_trim,               40,     14 },
{ update_altitude,         40,    100 },
{ run_nav_updates,          8,     80 },
{ update_thr_average,       4,     10 },
{ three_hz_loop,          133,      9 },
{ compass_accumulate,       8,     42 },
{ barometer_accumulate,     8,     25 },
{ update_notify,            8,     10 },
{ one_hz_loop,            400,     42 },
{ ekf_check,               40,      2 },
{ crash_check,             40,      2 },
{ landinggear_update,      40,      1 },
{ lost_vehicle_check,      40,      2 },
{ gcs_check_input,          1,    550 },
{ gcs_send_heartbeat,     400,    150 },
{ gcs_send_deferred,        8,    720 },
{ update_mount,             8,     45 },
{ ten_hz_logging_loop,     40,     30 },
{ fifty_hz_logging_loop,    8,     22 },
{ full_rate_logging_loop,1,        22 },
{ perf_update,           4000,     20 },
};
```

Listing 3.4 Ten hz loop

```
// ten_hz_logging_loop
// should be run at 10hz
static void ten_hz_logging_loop() {
        /* block of instructions */

        /* your new function call here */
        our_new_function();
}
```

3.1.5 Flight modes

In this subsection we are going to give a brief description of what is a flight mode and how it works (listing some of the most important used in the test phase). The flight mode is simply a function that manage the data to send to the motors. For example, in «*Land*» flight mode, the copter slows down the motors until the copter lands. They are divided in flight modes that requires GPS lock and flight modes that do not require GPS lock. The fast_loop function checks at every loop if there are updates about the flight mode (if there are inputs about flight mode from GCS or radio control). This check is done on fast_loop because it is considered a very important function (when a change of flight mode is required, it should always be changed).

Some of the most important flight modes are:

- Stabilize
- AltHold
- Loiter
- Auto
- RTL

Stabilize mode allows us to fly the vehicle manually. This flight mode we are going to use for the tests on the autopilot to calculate the loop time that we will explain later on Chapter 5.

In *AltHold* mode (Altitude Hold Mode), the copter maintains a consistent altitude while allowing roll, pitch, and yaw to be controlled normally. When this flight mode is selected, the throttle is automatically controlled to maintain the current altitude. Roll, pitch and yaw operate the same as in «Stabilize» mode (the pilot directly controls the roll and pitch lean angles and the heading). To calculate the altitude, the flight controller uses the barometer to estimate it or, if enabled, a Sonar will provide even more accurate altitude maintenance.

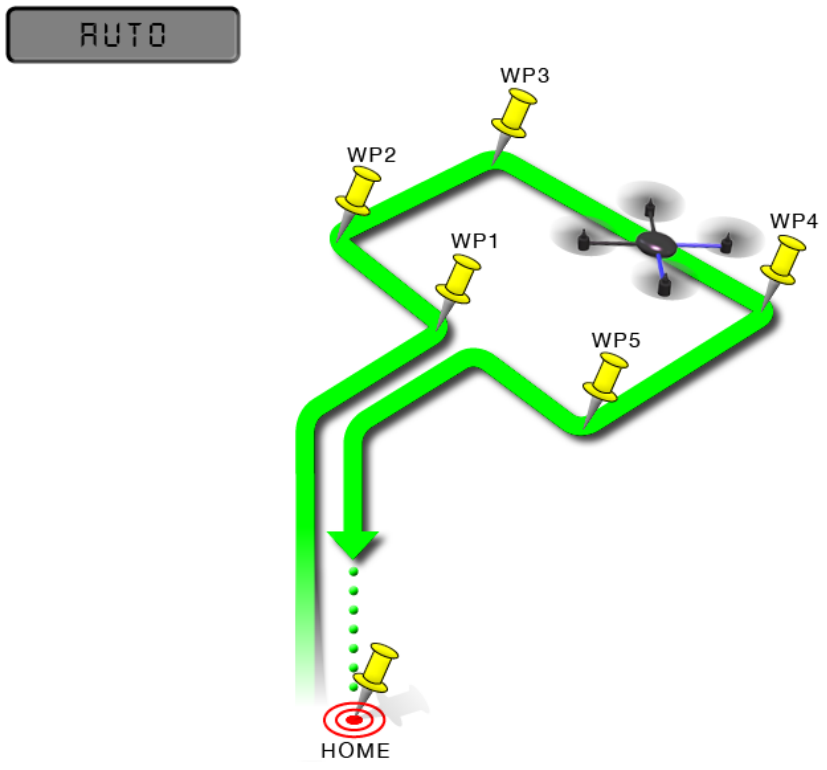


Figure 3.1: Auto flight mode

Loiter mode automatically attempts to maintain the current location, heading and altitude. The pilot may fly the copter in «Loiter» mode as if it were in manual. Releasing the sticks, the copter will continue to hold position. In this flight mode it is required a GPS installed on the quadcopter for the position maintenance.

In *Auto* mode the copter will follow a pre-programmed mission script (Figure 3.1) stored in the autopilot which is made up of navigation commands (i.e. waypoints) and "do" commands (change altitude, change flight mode, slow down or speed up etc.). «Auto» mode incorporates the altitude control from «AltHold» mode and position control from «Loiter» mode and should not be attempted before these modes are flying well.

In return to launch (*RTL*) mode, the copter navigates from its current position to hover above the home position (see Figure 3.2). The home position is always supposed to be the copter's actual GPS takeoff location or the location where tha copter was in when it was armed. So «RTL», like «Loiter» and «Auto», is a GPS-dependent. The copter will first rise to RTL_ALT before returning home or will maintain the current altitude if the current altitude is higher than RTL_ALT. The default value for RTL_ALT is 15m.

In Chapter 5 we are going to use «Auto» flight mode with «RTL» and «Loiter» incorporated to test the firmware with the SITL simulator.

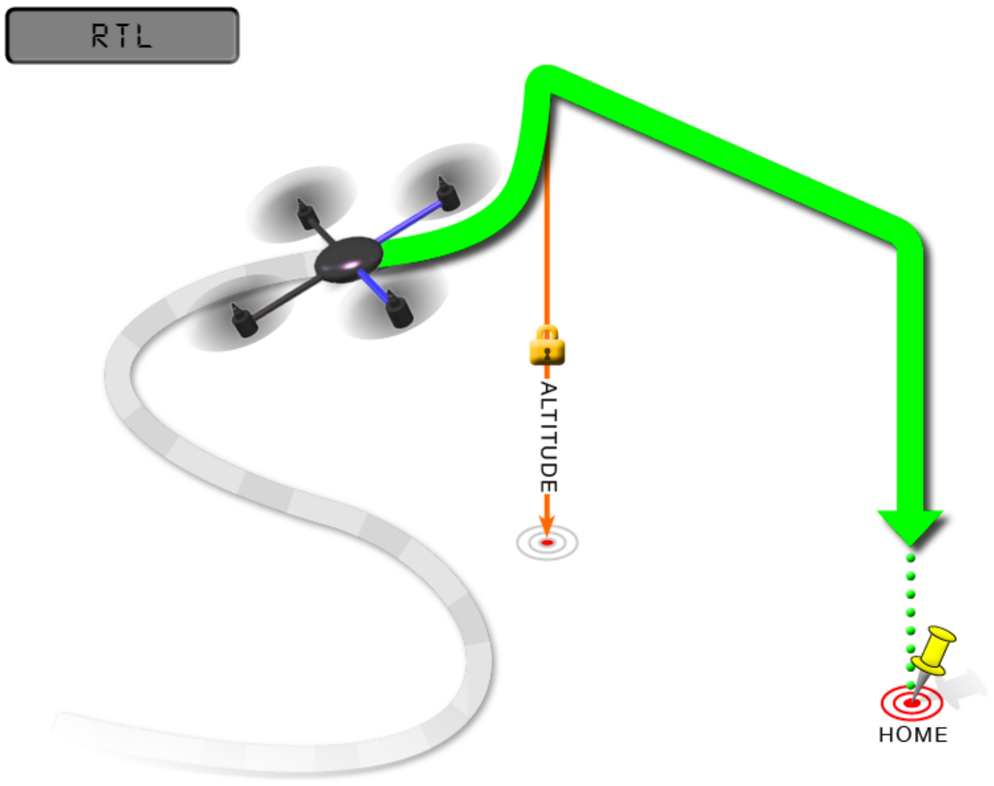


Figure 3.2: RTL flight mode

Chapter 4

Re-engineering of Ardupilot's Flight Control

In this chapter we describe the process from the problems found until the implementation phase. In Section 4.1 the main problem and one possible solution are shown. The existing tools using reactive programming paradigm can not be used on Ardupilot for hardware and software limitations that we are explaining later. The solution in this case is to implement from scratch a library (ReactiveCpp) for this language paradigm. Section 4.2 presents how this solution can be applied to a control loop and finally, in Section 4.3, we are going to apply this solution to the Ardupilot code.

4.1 ReactiveCpp

In this section we describe the main problem we have encountered on the implementation phase. It is divided in two subsections. In the first one the problem is described and in the second subsection our solution that includes the implementation of a new library is described.

4.1.1 Problems

The tools we have mentioned in the state of the art (see Subsection 2.1.7 for more information) do not work with Ardupilot on Pixhawk or APM board for some hardware and software limitations that we are going to explain now. These tools use other libraries that are difficult to port for ARM architecture (in this case for Pixhawk board). The react.cpp library uses Boost and TBB library [20]. The first one (used also by Sodium library) provides free peer-reviewed portable C++ source libraries.

TBB Intel® Threading Building Blocks (Intel® TBB) lets us easily write parallel C++ programs that take full advantage of multicore performance, that are portable and composable, and that have future-proof scalability. It widely uses C++ template for task parallelism. In this case the lack of multithreading is an hardware limitation. It is used only on cpp.react library and allows the multithreading access. This technique is used when an object depends from two or more objects and it must check if the propagation can takes place or it should wait a response from all the other objects.

Boost Boost is a set of libraries for the C++ programming language that provides support for tasks and structures such as linear algebra, pseudorandom number generation, multithreading, image processing, regular expressions, and unit testing. It provides free peer-reviewed portable C++ source libraries. Boost library is used by almost all the tools mentioned before.

STL The Standard Template Library (STL) is a software library for the C++ programming language that influenced many parts of the C++ Standard Library. It provides four components called algorithms, containers, functional, and iterators. This library is used by all the tools mentioned in the state of the art.

The problem is that porting these libraries for ARM architecture involves large costs in terms of memory. There are some guides that explain how to cross-compile these libraries but their size is not acceptable for the ARM board. The STL library is not yet supported from Ardupilot project (std template and lists are not yet implemented for Ardupilot project). Therefore another library exists, uClibc++ (micro controller C++ library), that adapts STL for ARM processor. The version is ported in the NuttX (see Section 3.1, external support code for Ardupilot) and we can find it in the NuttX repository in the misc/ directory. Anyway it is not compatible with all the boards and it is currently in development.

All these libraries on C++ make use of STL library. As further contribution we decided to create a new library from scratch to avoid these problems (the ReactiveCpp library). Avoiding these libraries, the new library can be used also in other boards. Below is shown the structure of this library and how can be used on a control loop.

4.1.2 Solution

The idea is to not use libraries like Boost, TBB and STL (or uClibc++). In our implementation we need to use lists but since they are managed from STL (std::list) we must implement them from zero. We specifically need lists of function pointers. A simple structure of our list (ListOfFunctions) is shown in Listing 4.1 where the struct is composed by a node that points to NULL or to another node and a void (*function)(void) that is a pointer to a function. This is the structure of our list of function pointers. In Listing 4.2 the two main methods that we need later for our library implementation (append and call methods) are shown:

- The append(&f) method is simply a push-back implementation. It appends to the the list a pointer to function.
- The call() method executes all the functions of the list. It scrolls the entire list and for each pointer to function runs the correspondent function.

Listing 4.1 ListOfFunctions.h

```
class ListOfFunctions {

private:
        struct node
        {
                void (*function)(void);
                node *link;
        }*p;

public:
        ListOfFunctions ();

        //Add element at the end of the list
        void append(void (*)(void));

        //Execute all the functions of the list
        void call();

}; // ListOfFunctions
```

Now, having the implementation of the list without the STL, we can use it on ReactiveCpp library. In Listing 4.3 we have considered as a «*stream*» all the functions related to one node of the graph that are going to be called when an «*alert*» is triggered. Each stream is composed by two lists and an integer value (Listing 4.4):

- bind_list is a list with all the function pointers that characterize a single stream.
- stick_list is a list with function pointers that launch other streams.
- value is an integer that permit us to know when the stream is triggered (if it changes we have new data, the call method is called)

We have implemented two types of constructs:

Listing 4.2 ListOfFunctions.cpp

```
#include "ListOfFunctions.h"

ListOfFunctions::ListOfFunctions(){
        p=NULL;
}

//Add element at the end of the list
void ListOfFunctions::append(void (*func)(void))
{
        node *q,*t;

    if( p == NULL )
        {
                p = new node;
                p->function = func;
                p->link = NULL;
        }
        else
        {
                q = p;
                while( q->link != NULL )
                        q = q->link;
                t = new node;
                t->function = func;
                t->link = NULL;
                q->link = t;
        }
}

//Execute all the functions of the list
void ListOfFunctions::call()
{
        node *q;
        for( q = p ; q != NULL ; q = q->link ){
                void (*f)(void) = q->function;
                f();
        }
}
```

- stream() creates an independent signal. It is triggered when the value is updated to a new value.
- stream(stream &s1, ...) creates a signal that depends from one or more signals. It is triggered when the value is updated (in this case it is updated from the other dependent streams)

The three main methods of an object of this library are:

- push(int) - updates the value of the stream. Then it will be valuated if the new value is different from the last one. If the value is the same the method will return nothing, if the value is different the method will call bind_list.call() and stick_list.call() to start the propagation of new data.
- bind(void (*)(void)) - adds a function pointer to the bind list
- stick(void (*)(int)) - adds a function pointer to the stick list (generally used to create a dependency between two streams)

Lets do a simple example to understand better how the library works. In Figure 4.1 is shown a dependency graph. The «S» is our signal from which depends other streams like «A», «B» and «C». It has a value (value = 23 in this case), a bind_list with s1, s2 and s3 (function pointers) and a stick_list with a, b and c (function pointers to other streams). From stream «A» depends other three streams («X», «Y» and «Z»). When we read a new value with S.push(newValue), if it is different from the last one, the value of «S» will be updated and the flow of changes will be in the order of the bold arrow, simulating the depth first search algorithm [26].

In the same example, if Z depends from A and from B, both will have on stick_list the function pointer «z». This implies the execution of stream Z twice. As we said on the state of the art, this is a common problem for reactive language tools and having the limitations described before on this section, it is impossible to manage the concurrency (Z must wait A and B to do a valuation if there is a propagation to do). One solution is to create a dependency between the sum of value of stream A and value of stream B, so Z depends on A+B and not anymore from the two streams separately. It can be done adding a pull-based style in the streams depending from more than one stream (mixing push and pull styles [24]). Another solution is to manage this dependency on the code. Considering the same example, if we know that A and B depends from S, if A run, B will run after. One solution is to cut the dependency of Z from A and leave only that from Z to B (like in Figure 4.2). In this case, every changes of Z will be propagated only once in A and only once in B. This problem we will find later on the Ardupilot code, where some objects depend from more then one object.

Here we describe the properties of our library referring to the taxonomy [5] of reactive language paradigm mentioned in Section 2.1.

Listing 4.3 ReactiveCpp.h

```
#ifndef __REACTIVE_CPP__
#define __REACTIVE_CPP__

#include <stdlib.h>
#include <stdio.h>
#include <ListOfFunctions.h>
#include <ListOfFunctions2.h>

namespace alert {

class stream {

public:
        stream();
        stream(stream &s1);
        stream(stream &s1, stream &s2);
        int getValue();

        void push(int);
        void bind(void (*)(void));
        void stick(void (*)(int));


private:
        typedef void (stream::*func_ptr_t_)(int);
        func_ptr_t_ func_ptr_;

        typedef void (*func_stream)(int);
        func_stream f_s;

        struct stream_impl;
        stream_impl *impl;
};
}
#endif // __REACTIVE_CPP__
```

Listing 4.4 ReactiveCpp.cpp

```
#include "ReactiveCpp.h"

namespace alert {
struct stream::stream_impl {

        int value;
        ListOfFunctions bind_list;
        ListOfFunctions stick_list;

        void push_value(int newvalue){
                if (this->value == newvalue)
                        return;
                this->value=newvalue;
        }
};

stream::stream(){
        this->impl = new stream_impl();
        this->func_ptr_ = &stream::push;
}

stream::stream(stream &s1){
        this->impl = new stream_impl();
        this->func_ptr_ = &stream::push;
        s1.stick(this.push(s1.getValue));
}

int stream::getValue(){
        return this->impl->value;
}

void stream::bind(void (*func)(void)){
        this->impl->bind_list.append(func);
}

void stream::stick(void (*func)(int)){
        this->impl->stick_list.append(func);
}

void stream::push(int newValue){
        if (this->impl->value == newValue)
                return;
        this->impl->value=newValue;
        this->impl->bind_list.call();
        this->impl->stick_list.call(this->impl->value);
}
}
```

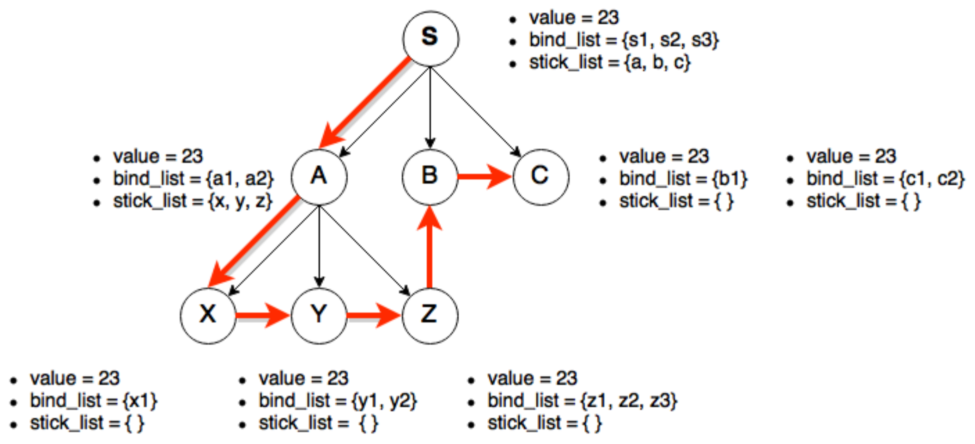


Figure 4.1: Propagation example

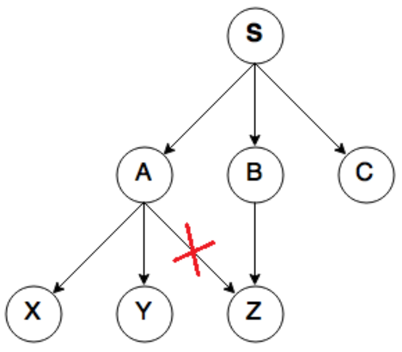


Figure 4.2: Example solution

Basic abstraction In this library we have implemented only streams of data. It is a type of signal. When something happens, the stream is triggered and then functions are executed (behavior of a signal triggered).

Evaluation model A push-based model was chosen. The data must be propagated when new values are available and not when they are required.

Lifting ReactiveCpp provides lifting operators that transform ordinary C++ functions into behaviors. The programmer has to manually retrieve the value of a behavior in order to use it with primitive operations.

Multidirectionality The library does not support the multidirectionality. Data are propagated from signals (the root of the graph dependency) to the leafs (leaf and other nodes are streams in this case).

Glitch avoidance It does not support the glitch avoidance. The programmer has to detect the critical paths and manage them on the code. Multithreading helps avoid it, but in our case we can not make use of it.

Support for distribution It is not implemented with the idea to use it on distributed systems, so it does not support this feature.

4.2 Example in a control loop

As we described in Chapter 3, the main core of the Ardupilot code is the main loop. Here we will explain how the reactive programming (especially using ReactiveCpp library) can be introduced in a control loop.

We are going to consider this simple example of control loop in Figure 4.3. It is about a vehicle with a GPS sensor installed on it. On each loop, the vehicle always calls:

- moveVehicle()
- printCurrentPos()

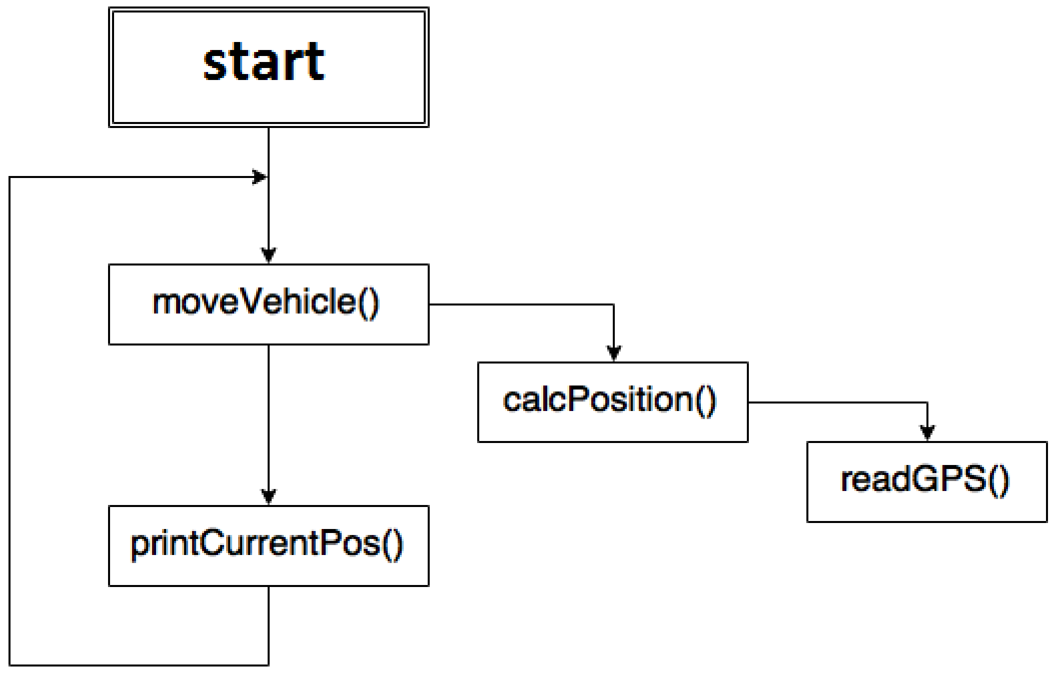


Figure 4.3: Example of control loop

The first method (moveVehicle()) needs to know the current position, so it calls the calcPosition() method. The same thing is repeated from calcPosition() with the readGPS() method. The printCurrentPos() method just writes on the log file the current position.

The goal of this control loop is to maintain always the same position of the vehicle. As we know, the vehicle can move from its desired position due to wind or other external factors. In this example it is calculated every loop where the vehicle is and if it has moved from the desired position, command are sent to turn it to the desired position. This control is done always. It makes sense if the external factors exist always and does not make sense when the vehicle stays at the same position for a long period of time. If the GPS does not detect new positions, it is a waste of memory and CPU usage to recompute unnecessarily these functions, especially in case of bigger loops.

The idea is to create a graph dependency through the reactive programming. We can set as signal the value of GPS. When it changes, we simply inform all the values depending on that value (Figure 4.4).

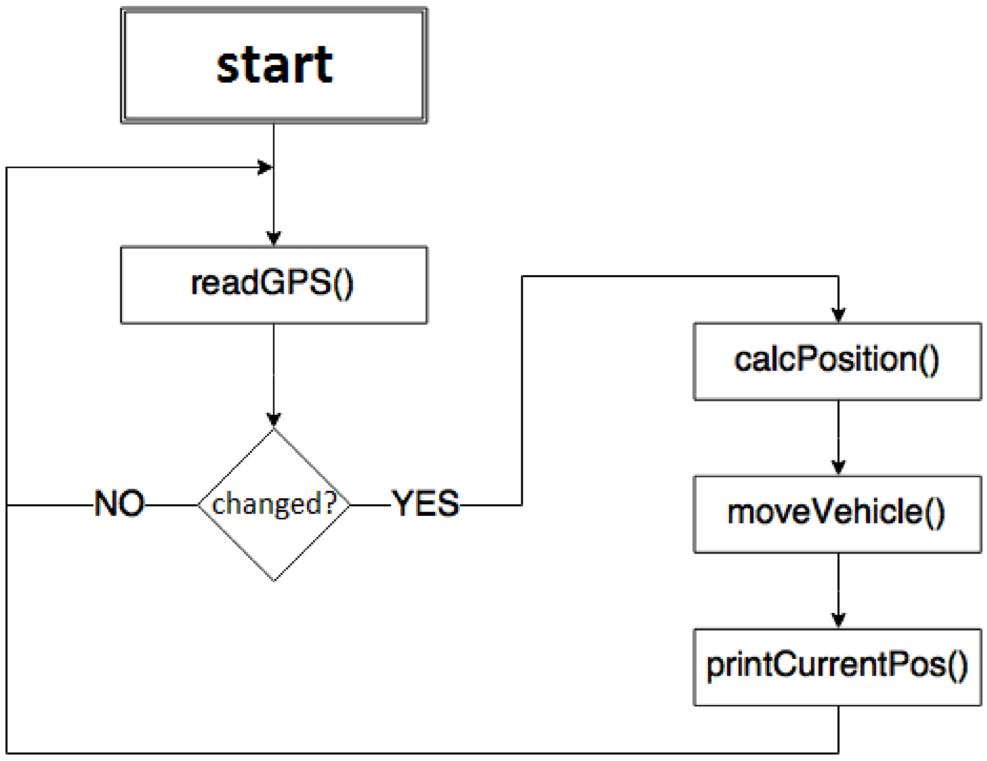


Figure 4.4: Example of control loop with reactive library

In this modified example, in each loop we can check if the GPS has read new values. We can simply use readGPS() method, in other more complex we can put together all the methods related to sensor reading in one single method. It is done with the pull-based approach because we are going to read the value of the signal. It is not the library style because we are not yet using it. When the checkGPS() realizes that new values are available, a propagation of changes is started from the GPS signal. Unlike the original style where the computation flow goes from the moveVehicle() to readGPS(), in this case we have an inverted computation flow, from GPS sensor to moveVehicle() and in addition at the end we have the printCurrentPos() method. It is added here because it depends by GPS.

The difference between the two styles is that with reactive programming we can avoid unnecessary execution of functions. If the sensor changes at each cycle (vehicle moving or external factors too strong), there is no difference. If the vehicle is stopped for a long time and external factors do not effect the vehicle moving, the moveVehicle() method is called only once until new data is read from GPS sensor. If the loop runs at 100Hz (100 times at second) and the vehicle is moving for 2 seconds and for other 2 is stopped, with the original code the moveVehicle() will be called 400 times and with the modified one this method will be called 200 times (in the last 2 seconds it will stays in listening for new data). In the modified example there will be a "waste of time" by checking for new data (a control should be performed at each cycle) but anyway the total time (especially if the "computation path" is long) will be lower then the original one. The time saved from the unnecessary execution of functions can be used in some ways:

- allowing the system to execute other additional functions permitting a more accurate control of the vehicle

- allowing the system to re-check for new values from sensors more than once at loop (in Ardupilot it can not be verified because sensors are read once at loop)

4.3 Ardupilot re-engineering

In this section we are going to explain all the changes we made on the Ardupilot project (in particular on the ArduCopter sub-project). In Subsection 4.3.1 there is a description of our choices related to the part of the code that can be personalized. There is a detailed description of all the parts that we are going to change, thus allowing us to understand the changes made. The main changes are explained on subsections related to setup (4.3.2), main loop (4.3.3), fast loop (4.3.4) and scheduler (4.3.5).

4.3.1 Overview

In Figure 4.5 is graphically shown how the ArduCopter code runs. There is an initial phase where all the variables are initialized (`setup()`). Then, the main loop (`loop()`) is executed and it runs until the execution of the entire vehicle terminates. In the main loop is called a method that waits for an INS sample (inertial sensor that mixes the accelerometer, gyroscope and magnetometer data), the fast loop (`fast_loop()`) and then with the remaining time the scheduler (the available time is passed and the scheduler choose a set of tasks that can be run). Data are sent at any loop to the motors (motors input). In the same figure, the instructions that calculate the available time immediately after the fast loop call are not shown for simplicity.

In Figure 4.6 we can see which methods are called on fast loop. As we said in Subsection 3.1.2, in fast loop are called methods with the highest priority (need to be called at any loop)

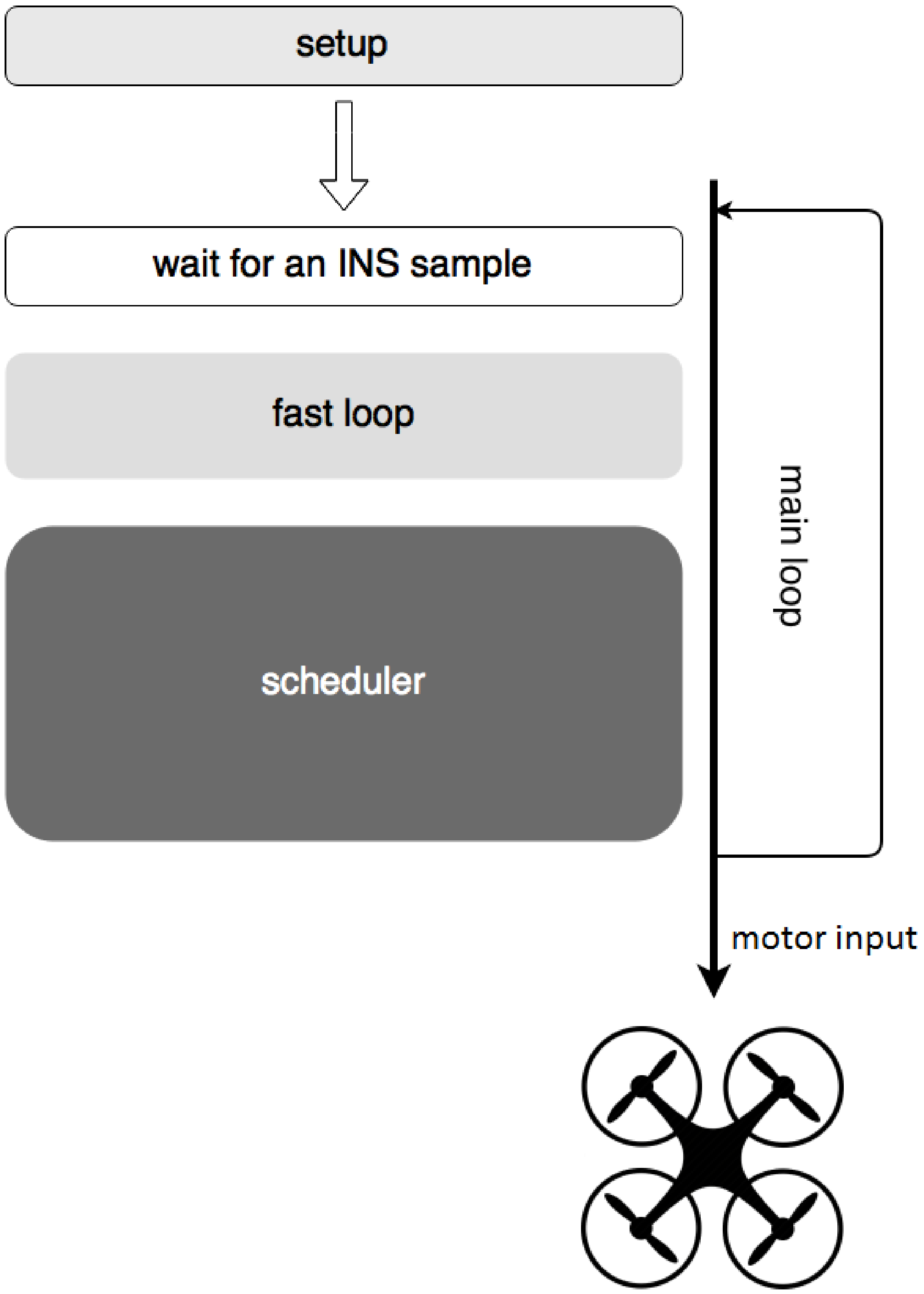


Figure 4.5: Arducopter

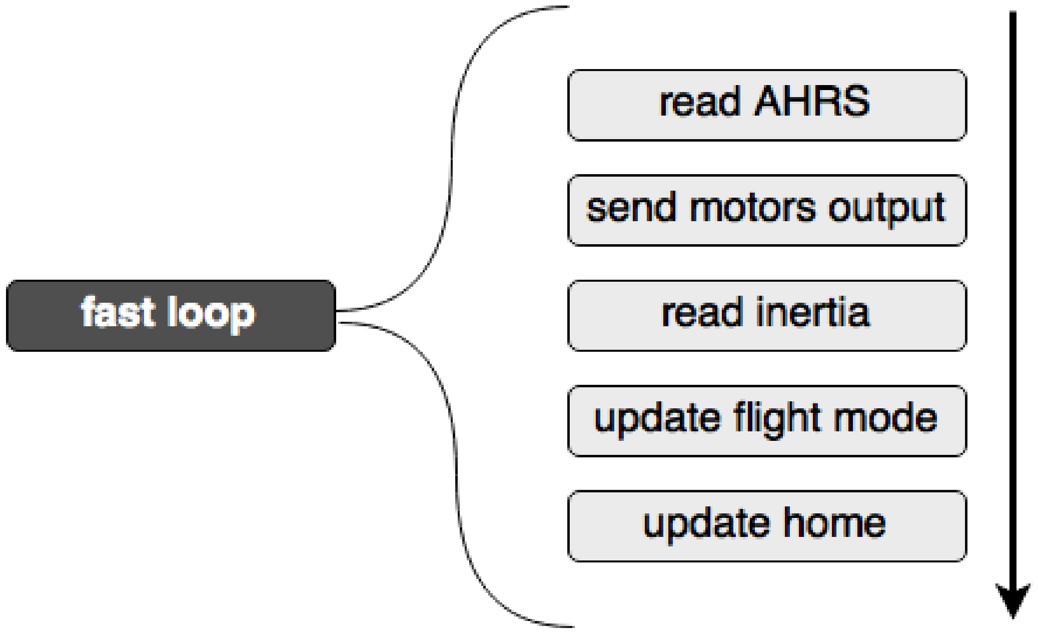


Figure 4.6: Fast loop

In Listing 4.5 are shown the variables we will manage to create a dependency flow from sensors to the final output. In lines 1, 2 and 3 there are three sensors managed from ardupilot to calculate the position of the vehicle.

- gps (if installed) gives the 2D position
- barometer is used to calculate the altitude (a sonar sensor gives more accurate information about the altitude)
- ins (inertial sensor) provides 3D orientation by integrating gyroscopes and fusing this data with accelerometer data and magnetometer data (it uses the board sensors)

In line 5 it is shown the ahrs object that is created from the combination of the three sensors listed before. An AHRS (Attitude and Heading Reference System), like an interial sensor, provides a 3D orientation. In line 6 is shown the interial_nav that gives all the information about our vehicle like position and velocity. In current location (current_loc) is stored the position of the vehicle that is used on other methods to send commands or to compare it with the destinations and so on.

Listing 4.5 Declaration of objects that manage the sensor data. *The first three objects manage the main sensors of the board (gps, barometer, accelerometer, magnetometer and gyroscope). The other* objects *(except the current location that is a struct depending on ahrs) are created passing to the constructor some sensor data.*

```
AP_GPS  gps;
AP_Baro barometer;
AP_InertialSensor ins;

AP_AHRS_DCM ahrs(ins, barometer, gps);
AP_InertialNav_NavEKF inertial_nav(ahrs);
AC_AttitudeControl attitude_control(ahrs, aparm);
AC_PosControl pos_control(ahrs, inertial_nav, attitude_control);

struct Location current_loc;
```

From these lines of code we can see the dependency between these senors and objects (see Figure 4.7)

Let us analyze now how many times these sensors are read from the Arducopter code. Referring to the scheduler initialization (see Subsection 3.1.3) we can see two methods witch refer to the barometer and gps reading (simplified in Listing 4.6). These two methods run at 50hz (second parameter is 8 and correspond to 50hz). We have new data of GPS and barometer every eight loops.

Listing 4.6 Barometer and GPS reading

```
/*
1 = 400hz
2 = 200hz
4 = 100hz
8 = 50hz
*/

AP_Scheduler::Task scheduler_tasks[] PROGMEM = {
        { update_GPS,             8,     90 },
        { barometer_accumulate,   8,     25 },
};
```

In Listing 4.7 is shown how often the ahrs (Attitude and Heading Reference System) and ins (accelerometer, magnetometer and gyroscope) are updated.

- In the first part is shown how often the ahrs is called to check for new values (see the full code in Listing 3.2 on Section 3.1.2)
- read_AHRS() called from the fast_loop is a method that call the update method of AHRS (other instructions are not shown in this method)

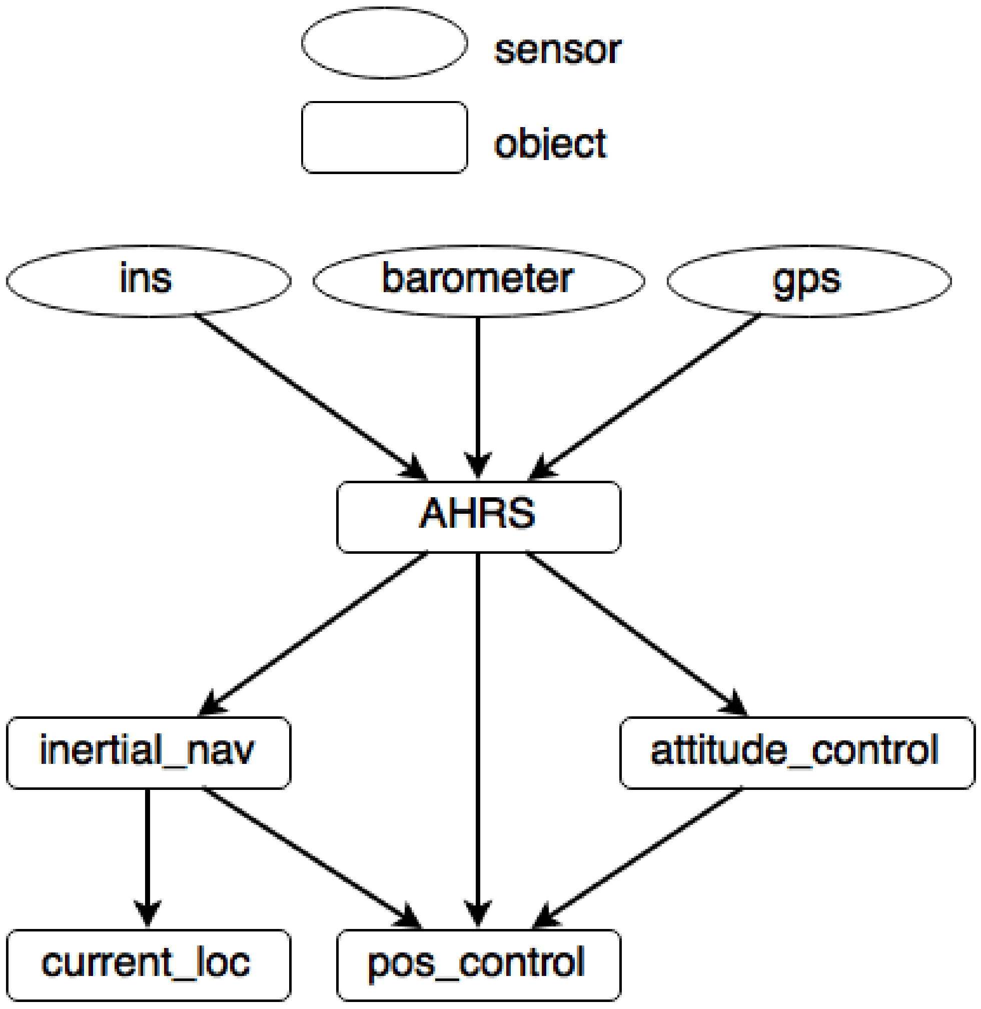


Figure 4.7: Sensors dependency. *Is shown the dependency between the sensors managed from the Ardupilot (Arducopter sub-project in this case).*

- the update method of AHRS at one point read the ins sensor and update the ahrs variable.

Listing 4.7 AHRS update. *(Attitude and Heading Reference System). Here is shown only a piece of fast_loop (see Listing 3.2 in Subsection 3.1.2).*

```
static void fast_loop() {
        // IMU DCM Algorithm
        // --------------------
        read_AHRS();
}
```

```
static void read_AHRS(void) {
        // Perform IMU calculations and get attitude info
        //-----------------------------------------------
        ahrs.update();
        /* instructions */
}
```

```
void AP_AHRS_DCM::update(void) {
        /* instructions */
        ins.update();
        /* instructions */
}
```

The attitude_control (see Listing 4.8) calls the rate_controller_run() method on fast_loop (each loop). The main function of this method is to call rate controllers and send output to motors object. To the motors is set the roll, pitch and yaw passing as input three variables that depend from ahrs object (see line 2,3 and 4 of the second part of Listing 4.8). The attitude control variable is called at each loop like the ahrs (immediately after and depending on the new values of ahrs object).

Listing 4.8 Attitude control

```
1 static void fast_loop() {
2
3         // run low level rate controllers that only require
4         // IMU data
5         attitude_control.rate_controller_run();
6 }
```

```
1 void AC_AttitudeControl::rate_controller_run() {
2         motors.set_roll(rate_to_motor_roll(_rate_target.x));
3         motors.set_pitch(rate_to_motor_pitch(_rate_target.y));
4         motors.set_yaw(rate_to_motor_yaw(_rate_target.z));
5 }
```

The last two objects we are going to describe are inertial_nav and current_loc. The first part of Listing 4.9 shows how often inertial_nav is used (called). It is read from fast loop (so at any loop) with the read_inertia() method. This method calls the update method of this object. In other words from fast_loop is updated the interial navigation object. The update consists in updates of velocity and position estimates using latest information from accelerometers augmented with GPS and barometer readings. Then the inertial_nav is used on every flight mode method which require the current position and velocity (almost all the flight modes).

The current_loc depends from inertial_nav object but it changes at 50hz (see third part of Listing 4.9)

Listing 4.9 Inertial navigation and current location objects

```
static void fast_loop() {
        // Inertial Nav
        // ————————————————
        read_inertia();
}
```

```
static void read_inertia() {
        // inertial altitude estimates
        inertial_nav.update(G_Dt);
}
```

```
AP_Scheduler::Task scheduler_tasks[] PROGMEM = {
        { run_nav_updates,        8,      100 },
};
```

```
static void run_nav_updates(void) {
        // fetch position from inertial navigation
        calc_position();
}

// get lat and lon positions from inertial nav library
static void calc_position() {
        // pull position from interial nav library
        current_loc.lng = inertial_nav.get_longitude();
        current_loc.lat = inertial_nav.get_latitude();
}
```

These objects we have just described are very related among each other. When an update method related to each of these object is called, it executes all the code even if there are no new values. For example, if ahrs does not change for 20 loops (the update method of the ahrs object is called once at loop), the inertial navigation object (interial_nav object) will execute the update code at any loop having always the same values. In this thesis we are going to avoid this behavior introducing the reactive programming paradigm on the propagation of sensor changes. Referring to Figure 4.7, the flow of changes of the sensors is not performed in order from sensors to final objects. For example, the ins.update() is called only when it is needed from ahrs even if ahrs depend from it. Ardupilot code performs so a pull-based model. The main problem is that a lot of code is executed when is not needed. In the next subsections we are going to describe the main changes to the Ardupilot code using the ReactiveCpp library. All these objects are going to call their update method only when there is a change on their parents node. From now on we are going to create signals for each object and we will refer almost always to these.

4.3.2 Setup

In order to initialize the signals, the only way to do that is on the setup code. It is called once at boot and it is used to initialize all the variables used by ArduCopter. In Listing 4.10 are shown the signals that we are going to use. We are not going to use for now the gps_state and barometer_state (we are going to explain after this decision in subsection 4.3.5).

Listing 4.10 Modified setup code. *It is shown where the new code (signal initialization) is added*

```
void setup()  {

alert::stream ins_state;
alert::stream gps_state;
alert::stream barometer_state;
alert::stream ahrs_state;
alert::stream inertial_nav_state;

void (*ins_alert)(void) = &read_AHRS;
ins_state.bind(ins_alert);

void (*ahrs_alert)(void) = &read_inertia;
void (*attitude_control_alert)(void) = &check_attitude;
ahrs_state.bind(ahrs_alert);
ahrs_state.bind(attitude_control_alert);
ins_state.stick(pushAHRS);

void (*inertial_nav_alert)(void) = &calc_position;
inertial_nav_state.bind(inertial_nav_alert);
ahrs_state.stick(pushInertialNav);

}
```

Referring to Listing 4.10, when the ins_state is triggered, some instructions are going to be executed and at the end the ahrs_state is called. On the ahrs_state call, other methods related to ahrs are going to be called and at the end, just like with the ins_state, the signals depending on ahrs object are going to be called, like inertial_nav state. The same thing is going to happen with this signal and at the end it is not going to call other signals (in our case the propagation ends here).

In Listing 4.10, from line 8 until end, we can see the creation of dependency between our three signals. In parallel, in Figure 4.8 are shown the signals graphically.

- In line 9 is created a function pointer related of read_AHRS() method. Then in line 10 it is added to the bind_list of ins_state signal. This characterize the behavior of this signal. Every time ins_state is triggered, the read_AHRS() method is going to be called.

- In lines 12 to 16 we are going to initialize and choose the behavior of ahrs_state signal. In lines 12 and 13 are created two function pointers to read_inertia() and check_attitude() methods. In lines 14 and 15 these two function pointers are going to be added to the bind_list of ahrs_state signal defining in this way the behavior of ahrs_state signal. In line 16 we are going to create a direct dependency between ahrs_state and ins_state adding on the stick_list of ins_state the push method of the ahrs_state. The ins_state is going to first call all methods depending on it, then at the end it is going to trigger the other signals depending on it (ahrs_state.push(value) in this case).

- In lines 18-20 we are going to add the calc_position() method to the bind_list of inertial_nav_state signal, and then we are going to add to the stick_list of ahrs_state the push method of this signal creating another dependency between these two signal.

4.3.3 Main Loop

In the setup subsection, we see how signals are depending on each other, starting from the ins_state. In Figure 4.8, ins_state signal is triggered after an «alert». The other signals depending by other signals are triggered when the parent signal terminates the execution of all the functions on bind_list and starts the execution of the functions on the stick_list (a list of method that contain the push() method of child nodes). Returning to the «alert» signal, it is referred to the push() method of the ins_state signal. In Listing 4.11 we have added a new method (check_sensors()) on the main loop. We simply call the push(..) method on ins_state signal passing as input the last time the code has read new values about gyroscopes, accelerometers and magnetometers. This value always changes when there are new values read on these sensors. This «alert» define the start of the propagation values.

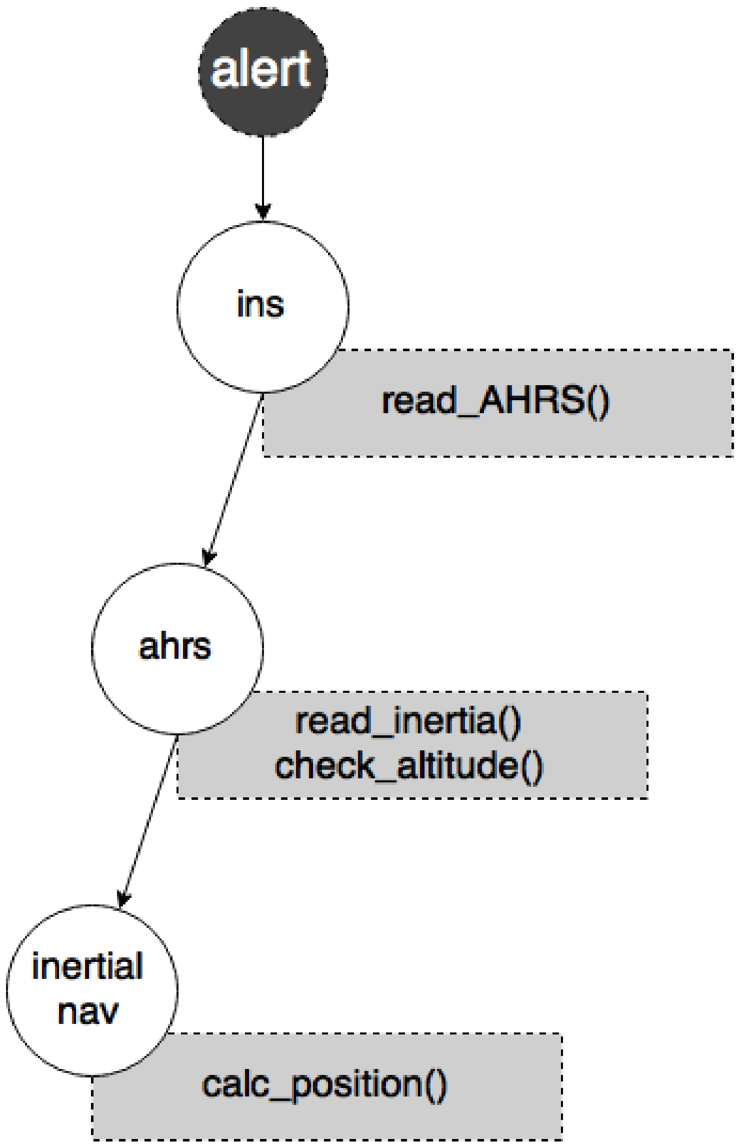


Figure 4.8: Dependency tree between signals. *Here are shown the signals we are going to use for a better propagation of sensor changes. Each signal is triggered when the object to which it was connected has made an update (for example when a read_AHRS() is called, the ahrs signal is triggered). The «alert» means that new values are available.*

Listing 4.11 Method related to sensor check

```
void loop() {
        check_sensors();
        ...
        ...
}
```

```
// check if sensors are updated
static void check_sensors(){
        ins_state.push(ins.get_last_sample_usec());
}
```

4.3.4 Fast Loop

In the fast loop (Listing 4.12) we have removed some lines of code. As we can see the read_AHRS(), read_inertia() and attitude_control.rate_controller_run() are removed from fast loop. These belong now to the signal behaviors explained before.

The check_attitude() correspond to attitude_control.rate_controller_run() (in line 13 of Listing 4.10). We have added the ins.update() instruction here because it should be called at any loop. In the original code of ArduCopter it is called at any loop by the read_AHRS().

Listing 4.12 Fast loop modified. *Here are shown the lines of code we have removed and one instruction added (ins.update()). The other instructions remained the same like Listing 3.2.*

```
static void fast_loop() {

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

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

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

        ins.update();

}
```

Compared to the original code, the fast loop code (the critical part of code of

ArduCopter) is lighter. The instructions that we have removed from fast_loop do not need to be called at any loop. They are going to be called only when new data are available, avoiding so the execution of unnecessary code. In the worst case (having always new values from the sensors), the Ardupilot will take the same time to process the sensors data. With a lighter fast loop, the time available for the scheduler is bigger. It enables the scheduler to run more tasks then the original code, having a better response of the system in general.

4.3.5 Scheduler

The scheduler is the part of code least affected by these changes. It is important to consider it because here, other check_sensors methods (related to other sensors) can be included. We have not considered the GPS and barometer sensors before because they have a different frequency (50hz) unlike the main loop (400hz). A solution to include these two sensors in the dependency tree is to create a method that checks for new values at the right frequency. In Listing 4.13 is shown a peace of code that can be combined with the above changes to introduce a dependency between GPS, barometer and ahrs objects. A check_sensors_fifty() method is called from the scheduler at 50hz frequency only for GPS and barometer readings. In the setup, the main changes are lines 9-13. The pointer to function to read_AHRS() is changed from ins_alert to _alert (it does not anymore depend only from ins). In line 12 and 13 is created the dependency between ahrs and the other two sensors. In 8 loops, the ahrs in the worst case can be updated 10 times (8 from the ins called at any loop, 1 time from gps and 1 time from barometer).

Listing 4.13 GPS and barometer signals

```
/*
1 = 400hz
8 = 50hz
*/

AP_Scheduler::Task scheduler_tasks[] PROGMEM = {
        { check_sensors_fifty,    8,      90 },
};
```

```
void setup()  {

alert::stream ins_state;
alert::stream gps_state;
alert::stream barometer_state;
alert::stream ahrs_state;
alert::stream inertial_nav_state;

//void (*ins_alert)(void) = &read_AHRS;
void (*_alert)(void) = &read_AHRS;
ins_state.bind(_alert);
gps_state.bind(_alert);
barometer_state.bind(_alert);

}
```

Chapter 5

Evaluation

In this chapter we describe the behavior of the system using reactive programming based library. We are going to compare it to the original ArduCopter project (see 2.2.2.2) showing some charts. The behavior should be very similar in both cases. The only way to compare it is to make use of generated log after each flight (simulated flights or real ones). From now on we will refer to the ardupilot project as original firmware or original version and to the modified project with a reactive library developed as in Chapter 4 as reactive firmware or reactive version (see the modified ardupilot with the ReactiveCpp on Section 4.3).

This chapter is divided into three parts. In Section 5.1 we are using only SITL (see 2.2.2.4) to simulate a flight and then extract information from log files. In Section 5.2 we are going to test the reactive firmware on the physical vehicle and control the real time of the main and fast loop to see how the performance is changed with the reactive programming paradigm. In the third part (Section 5.3) we are going to describe what can motivate a programmer to use the reactive programming paradigm in a control loop, modifying Ardupilot project.

5.1 SITL

In this section we are going to use two flight plans, the default one given by the ardupilot community (Figure 5.1) and a modified flight plan that is similar to the default one with a «Loiter» flight mode at the waypoint (WP) number 10. A waypoint corresponds to a sets of coordinates that identify a point in a physical space (in our case a simulated space). The «Loiter» flight mode takes 20 seconds, then the flight plan continues its path.

The two flight plans are composed by some waypoints. Starting from the «*Home*» waypoint (waypoint H in Figure 5.1), the quadcopter takes off until it reaches an altitude of 20m. It passes through the other waypoints with the «Auto» flight mode until at the end it turns back to «Home» waypoint and lands with the «Land» flight mode. The quadcopter reaches an altitude of 40m in the middle of the path (WP number 5) and then turns back to 20m. The MAVProxy console (see Figure 2.10) gives us some data in «real time» such as altitude, position, distance from finish, time passed, wind, vibrations etc. In the ArduCopter project we can not set the values of vibrations and winds (it is possible only for ArduPlane for now).

We are going to compare the Yaw and DesYaw (yaw value and desired yaw value of the vehicle during the flight) of the original version to the reactive one. We will do the same test with pitch, roll and altitude parameters. The pitch, roll and yaw (as shown in Figure 5.2) gives us the direction (yaw) and the moving direction (pitch and roll) of the vehicle.

- Roll - rotation around the front-to-back axis.
- Pitch - rotation around the side-to-side axis.
- Yaw - rotation around the vertical axis.

5.1.1 Yaw

In this subsection we are going to compare the desired yaw angle with the simulated yaw angle of the quadcopter. Comparing these two parameters we can extract the error (the difference between the two parameters). It is useful to understand how the vehicle reacts on external factors. With the simulator, it is not possible to set the wind or airspeed, anyway the error in yaw is generated from the simulated radio control input and the simulated value of the yaw angle. In Figure 5.3 and Figure 5.4 are shown the graphs about the two versions of firmware tested with the default flight plan.

As we can see the two graphs are very similar. In the x-axis we have the time and in the y-axis the difference between desired yaw and yaw (the error in yaw). We are going to analyze the data in detail to do a significant evaluation. From the log files, using Mission Planner (for more information see 2.2.2.3), it is possible to extract a text file with all the log data. In the «ATT» attribute (attitude information) of the log file we can find information about desired and simulated values of roll, pitch and yaw. In this case we have selected only the data where are stored information about yaw and desired yaw. We have calculated, from these data, the average value and the standard deviation related to the error in yaw. With the first value we can understand how much is the gap between the desired and simulated values and with the second value the distribution of these errors.

We have considered two scenarios that use only the first flight plan given from the community (the one without the «Loiter» flight mode). In the first scenario we have considered all the flight plan from the «Home» waypoint to the end. In the second scenario we have cut off the parts where the copter reaches a waypoint because if

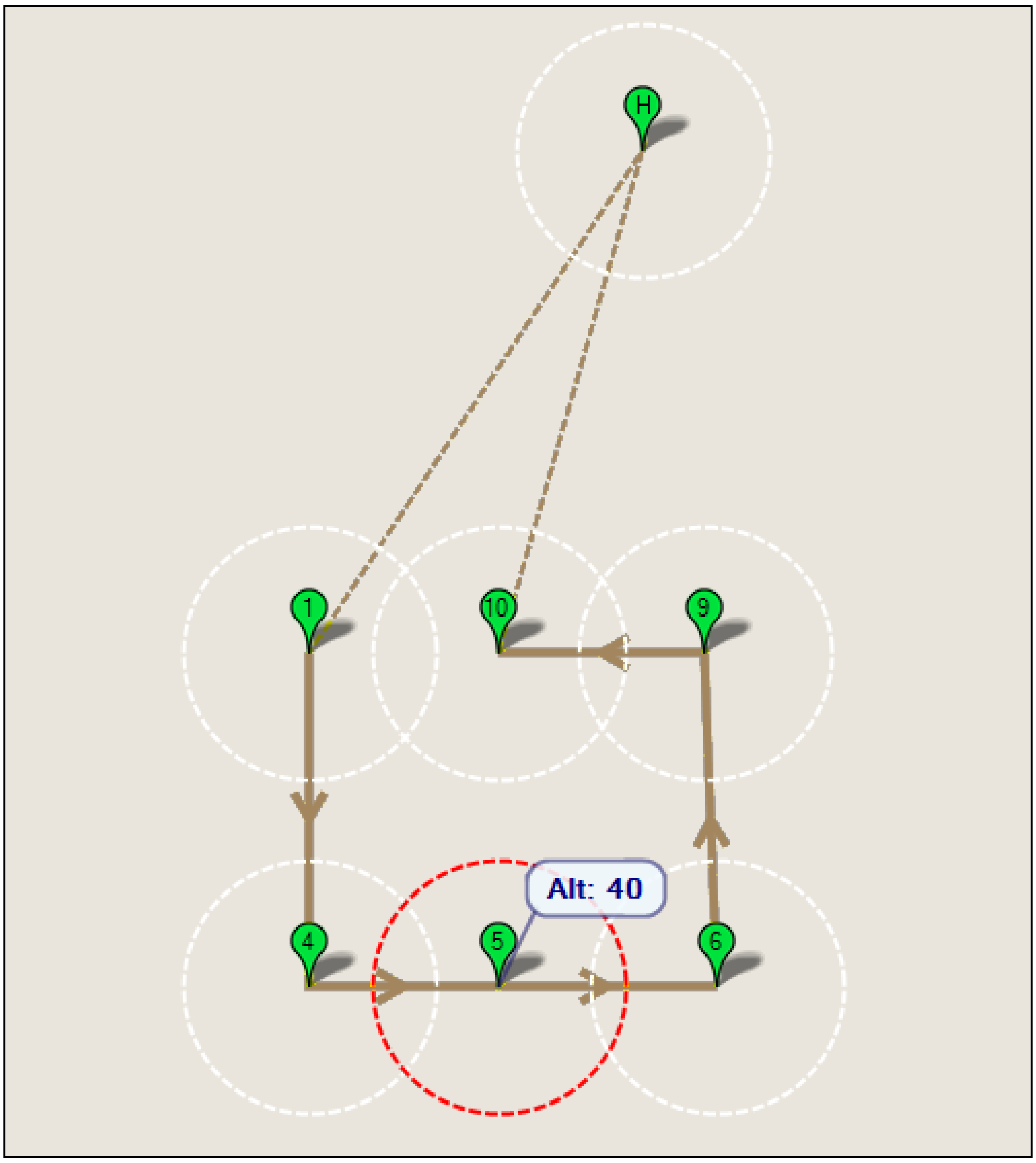


Figure 5.1: Flight plan. *It is composed by a set of waypoints (WP). In WP 5 the quadcopter reaches an altitude of 40m than return back at 20m.*

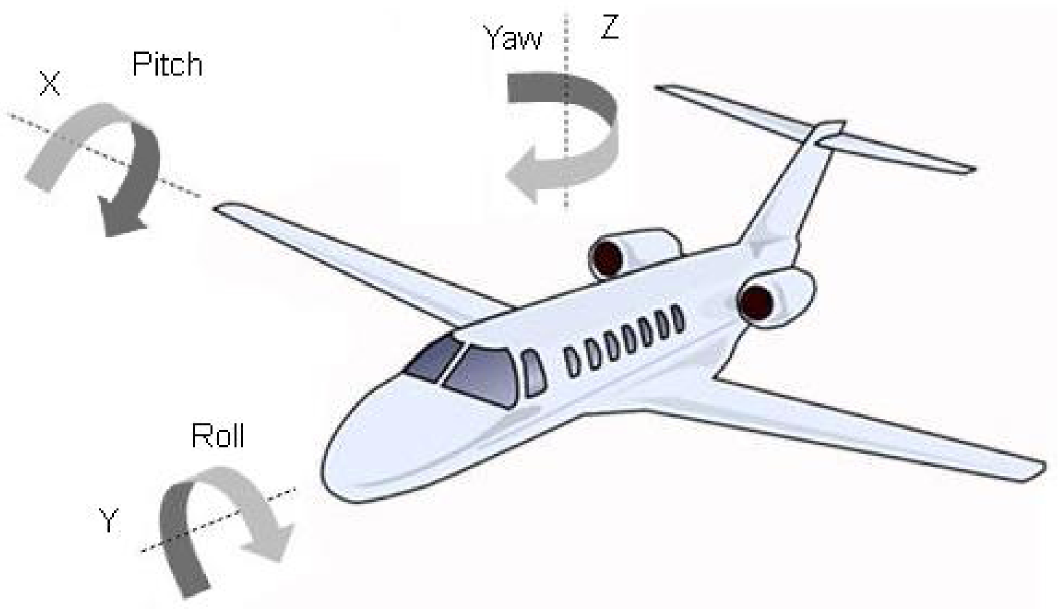


Figure 5.2: Pitch, Roll and Yaw. *A demonstrative image to show these three parameters related to the three axes.*

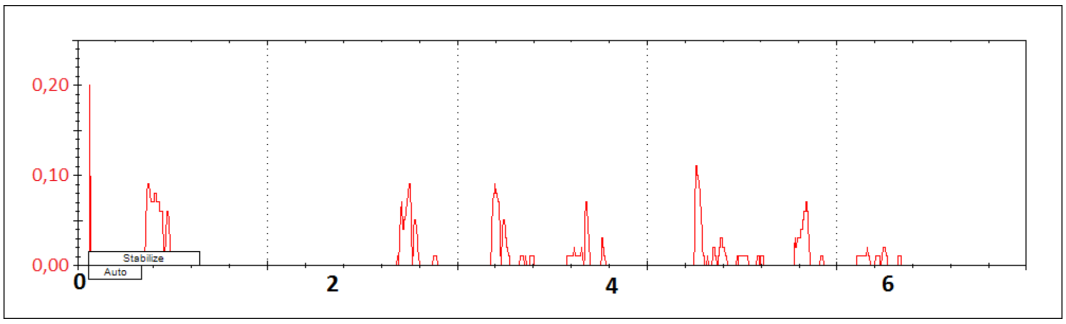


Figure 5.3: Error in yaw of the original Firmware. *In the x-axis we have the time in minutes and in the y-axis the error in degrees.*

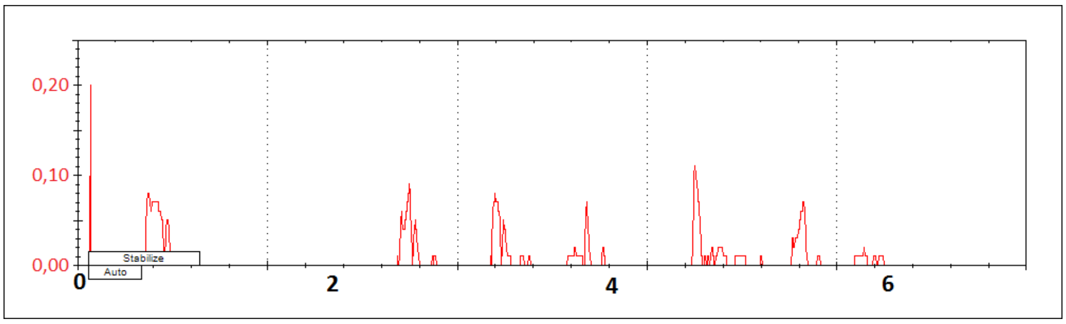


Figure 5.4: Error in yaw of the reactive Firmware. *In the x-axis we have the time in minutes and in the y-axis the error in degrees.*

it points to another waypoint the error variation can be very high (so we did not considered these 2 seconds needed to the vehicle to reach the right direction). In this way we can have more information about the «*vibrations*» of this parameter when it must be constant and see how the system reacts to external sensors (simulated from a linux system in our case).

	average	standard dev.
original	7,367199571	38,53454587
reactive	7,012069519	37,62386149

Table 5.1: Average and standard values of the error in yaw related to the first scenario. *Original flight plan considering all the path (from home to the end).*

In Table 5.1 (referred to the first flight plan, first scenario) we can see a slight difference between the two versions of firmware. The error in the reactive one is 5% lower then the error in the original one. We have obtained a better response from the reactive firmware related to the vehicle direction.

	average	standard dev
original	0,378516129	0,760354581
reactive	0,316451613	0,680496727

Table 5.2: Average and standard values of the error in yaw related to the second scenario. *Original flight plan considering all the path without the points where the vehicle reaches a waypoint.*

In Table 5.2 (referred to the first flight plan, second scenario) the average of the error in yaw is about 16% lower to reactive firmware compared to the original one.

On the last image of this subsection (Figure 5.5) we have a better view of the differences between the two firmwares. The light gray one is the reactive firmware and the dark gray one is the original firmware. In x-axis we have the time and in y-axis the error in degree. The reactive's yaw error stay almost always below the original's yaw error.

5.1.2 Pitch and Roll

In this subsection we have put together the error about pitch and roll because these two parameters give us the direction of the moving quadcopter. We have tested the two firmwares only with the modified flight plan because we are going to use the data when the vehicle is on «Loiter» flight mode (we are going to explain later why we need this flight mode). We can see that they are related. DesRoll and DesPitch are the pilot's desired roll and pitch angle in centi-degrees. Roll and Pitch are the vehicle's actual roll and pitch angle in centi-degrees. In Figure 5.6 we have the DesRoll, Roll, DesPitch and Pitch from the original firmware (it can not be

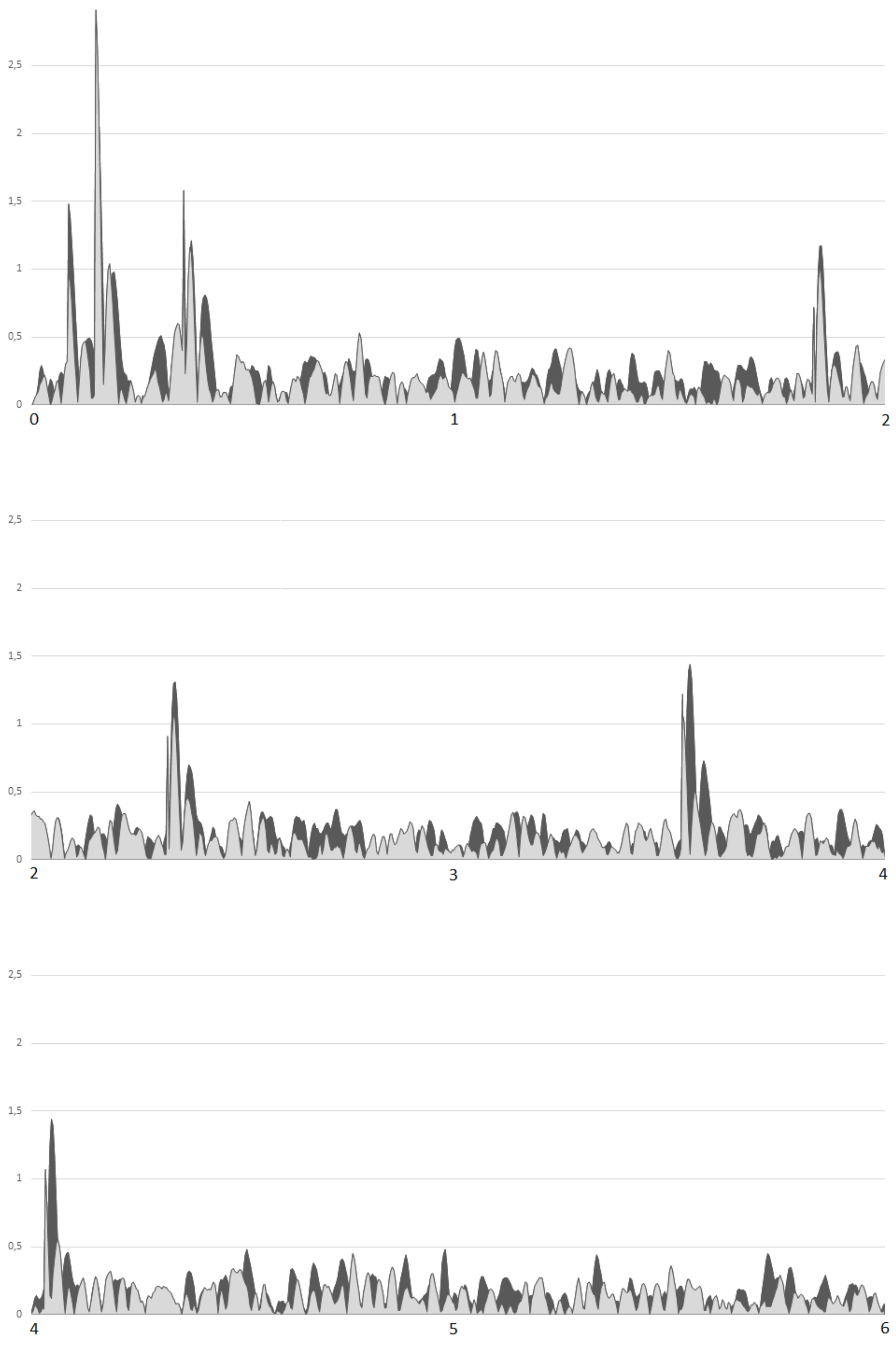


Figure 5.5: Error in yaw on original and reactive firmware. *The difference between the two errors on original firmware (dark gray) and reactive firmware (light gray). Chart divided in three parts to have a better view of the difference between the two firmwares. In the x-axis we have the time in minutes and in the y-axis the error in degrees.*

	ErrRoll	ErrPitch
original	0,3347	0,4005
reactive	0,2889	0,3601

Table 5.3: Average error in pitch and roll values. *The full path, from home to the end, is considered.*

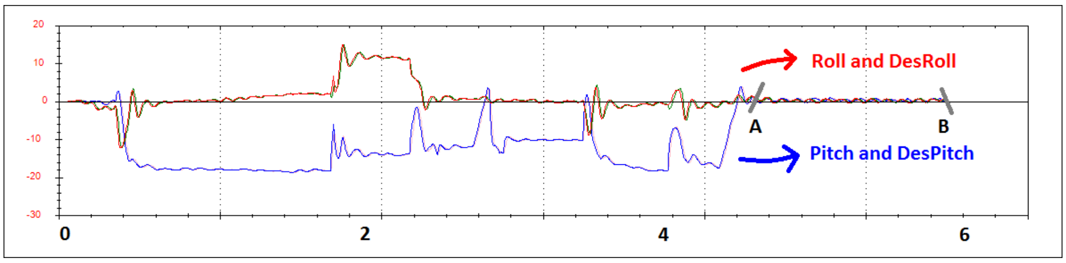


Figure 5.6: Error in roll and error in pitch of the original firmware. *In the x-axis we have the time in minutes and in the y-axis the error in degrees.*

distinguish DesRoll from Roll and DesPitch from Pitch). In Figure 5.7 we have the four parameters of the reactive firmware. The two graphs are very similar.

We are going to do two types of calculations. First of all we are going to calculate the average of the error in the two firmwares like we did with error in yaw. Then we are going to study the behavior of the system when it is in «Loiter» flight mode (path A-B of Figure 5.6 and Figure 5.7). In Figure 5.8 (path A-B zoomed) it is shown that the Roll and DesRoll charts presents a sort of amplitude (the behavior of Pitch and DesPitch is the same). In the reactive version the «amplitude» variation is smaller with the reactive firmware. In this case is significant to calculate the standard deviation to understand how these parameters change. The same study can be done in other points of the graph where the copter does not change very often the movement direction. Getting the data from the log file, we calculate the average errors of all the flight plan for the first case (Table 5.3). The difference between the error in roll of the two firmwares is about 13-14% lower in the reactive one. The reactive firmware has also a lower error in pitch of about 11%. So at first we have what we expected, a similar behavior using the reactive library. Calculating the error average in roll and pitch parameters we have obtained a better response to the inputs by the autopilot.

The second case of study for roll and pitch parameters is the study of the path A-B. Visually the «amplitude» of the graph (Figure 5.8) in reactive firmware is smaller than the original firmware. This path is referred to the «Loiter» flight mode. It is easiest to study this behavior in this flight mode because the roll and pitch must be constant, so it is easiest to study the vibrations of those parameters. Extracting some data from the log files we have calculated the standard deviation to see how those values change. In Table 5.4 we can see that in the original firmware, these

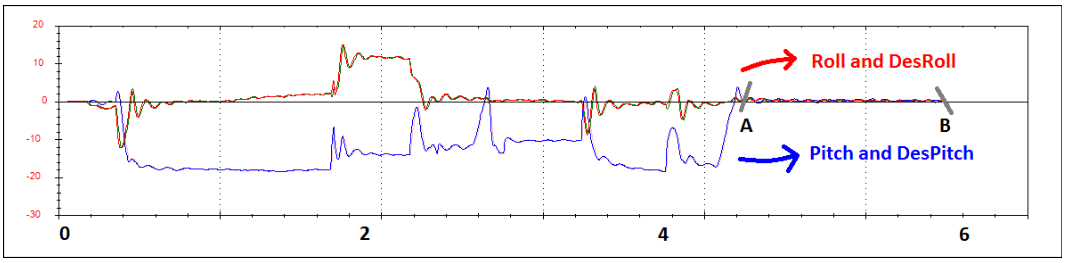


Figure 5.7: Error in roll and error in pitch of the reactive firmware. *In the x-axis we have the time in minutes and in the y-axis the error in degrees.*

	DesRoll	Roll	DesPitch	Pitch
original	0,343	0,3297	0,3388	0,3129
reactive	0,2272	0,2218	0,2144	0,2106

Table 5.4: Standard deviation related to pitch and roll parameters. *The modified path, from A to B, is considered*

four parameters have a bigger standard deviation. In the reactive one the standard deviation is about 35% lower. In an ideal autopilot system the standard deviation in this path must be 0.

In Figure 5.9 is shown graphically the difference between the error in pitch on the original version and the error in pitch on the reactive version related to the full flight plan. The reactive one as we can see stays almost always below the original version. Same thing we can deduce graphically for the error in roll from Figure 5.10.

5.1.3 Error altitude

In this subsection we are going to analyze the error in altitude. Comparing the real altitude of the vehicle with the desired altitude we can see in the two cases (original firmware and reactive firmware) how the error changes. In this subsection we are referring only to the default flight plan (the one without the «Loiter» flight plan). The three parameters of the two subsections above were extracted from the «ATT» message of the log files. Now, for the altitude we need for «CTUN» message (throttle and altitude information). In this message we can find BarAlt (altitude estimated from barometer), SAlt (altitude estimated from sonar), DSAlt (desired sonar altitude), DAlt (desired altitude) and Alt (altitude). Here we are considering only the DAlt and Alt parameters to calculate the error (the gap between the real value and the desired value). The test is the same done before. Looking at the graph (Figure 5.11) we can not see any difference between the two flights, so we are going to extract data to calculate the average of the error in altitude. Considering all the path from A to F where the altitude changes four times, we do not see the improvement. Anyway the average error is shown on Table 5.5 . The improvement is very small, only 0.9%.

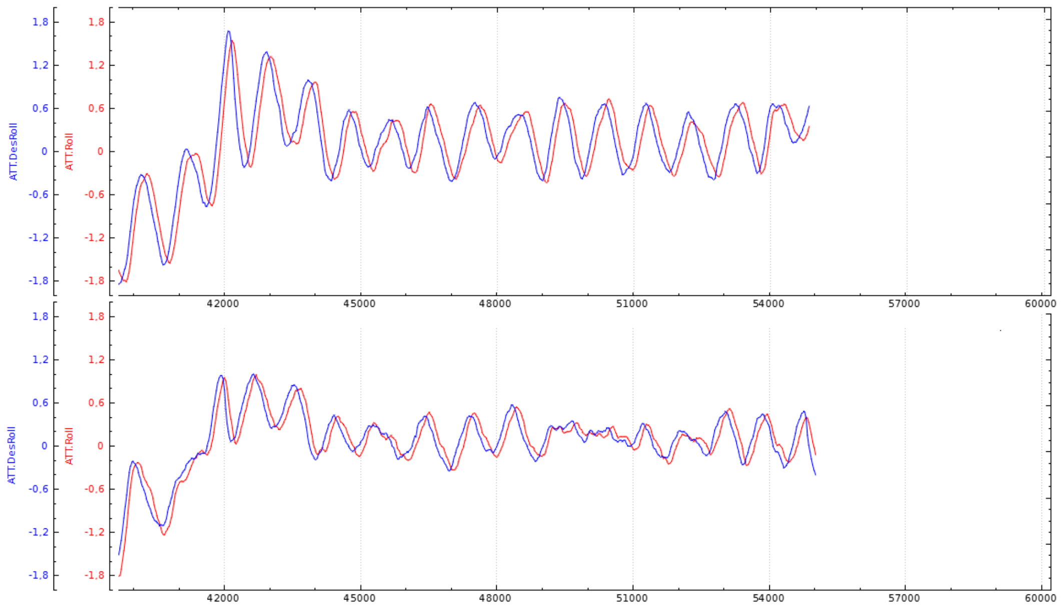


Figure 5.8: Amplitude of the Roll and DesRoll parameters. *The trend of the graph looks like a wave with a certain amplitude. This amplitude is bigger in the original firmware. In an ideal system it must be constant. In the x-axis we have the time in milliseconds and in the y-axis the error in degrees.*

	average error [m]
original	0,324
reactive	0,321

Table 5.5: Average error of the altitude of the full flight plan. *The full flight plan from A to F in Figure 5.11*

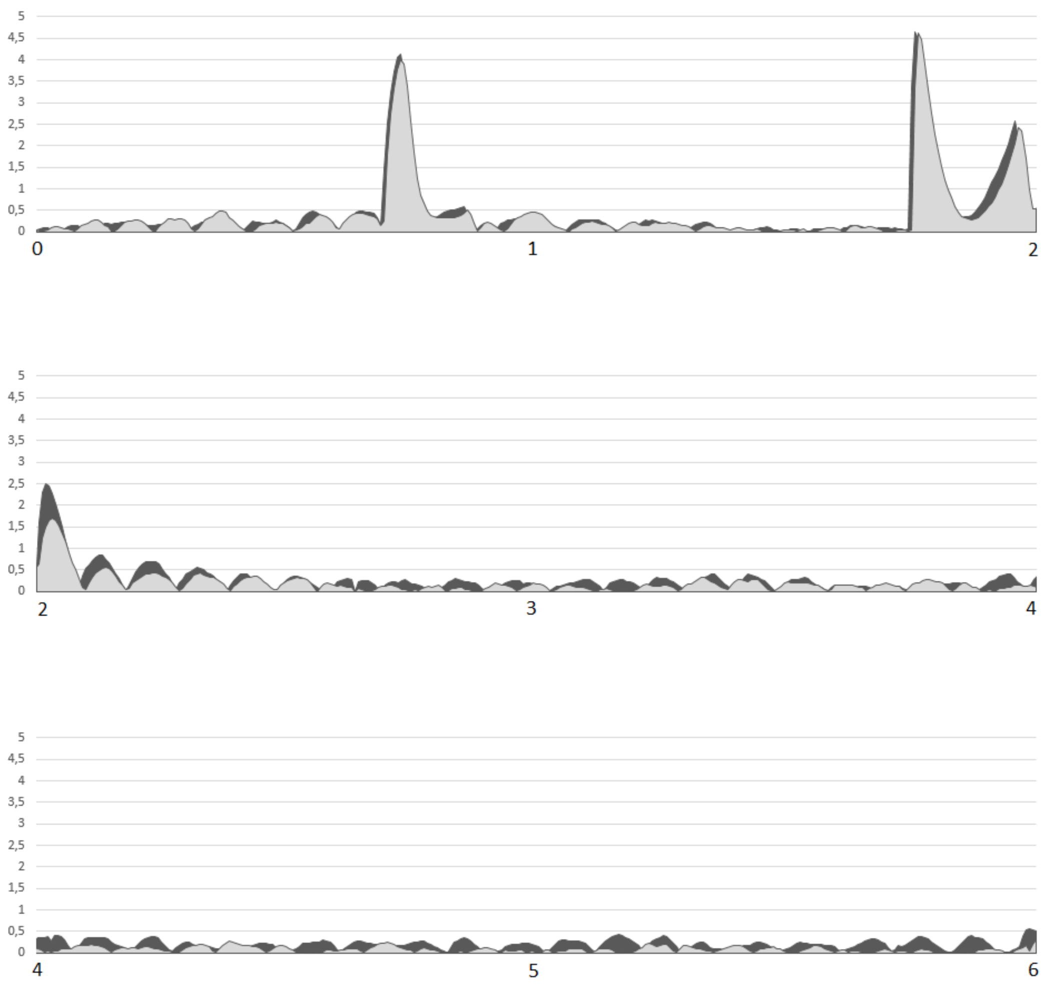


Figure 5.9: Error in pitch on original and reactive firmware. *The difference between the two errors on original firmware (dark grey) and reactive firmware (light grey). Chart divided in three parts to have a better view of the difference between the two firmwares. In the x-axis we have the time in minutes and in the y-axis the error in degrees.*

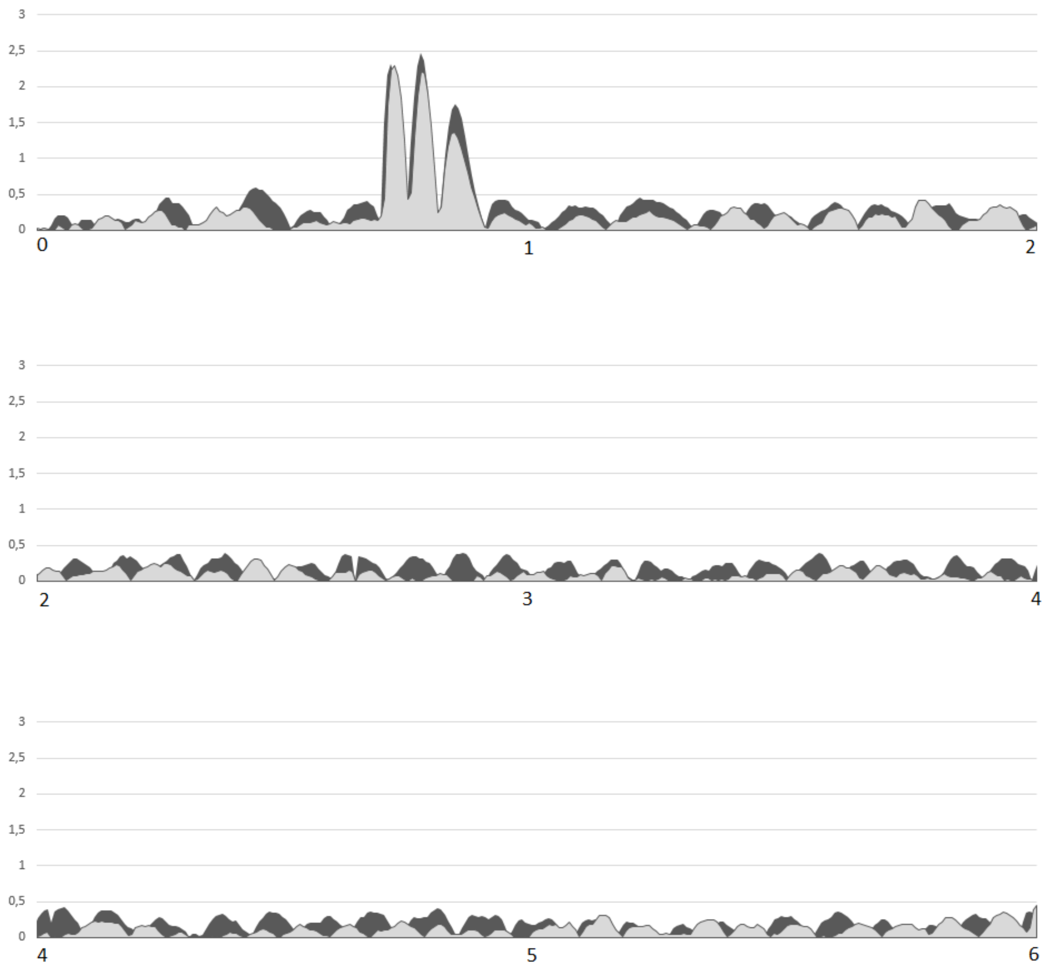


Figure 5.10: Error in roll on original and reactive firmware. *The difference between the two errors on original firmware (dark grey) and reactive firmware (light grey). Chart divided in three parts to have a better view of the difference between the two firmwares. In the x-axis we have the time in seconds and in the y-axis the error in degrees.*

	average error [m]
original	0,061
reactive	0,049

Table 5.6: Average error of the altitude of the modified flight plan. *The modified flight plan correspond to the path B-C and D-E*

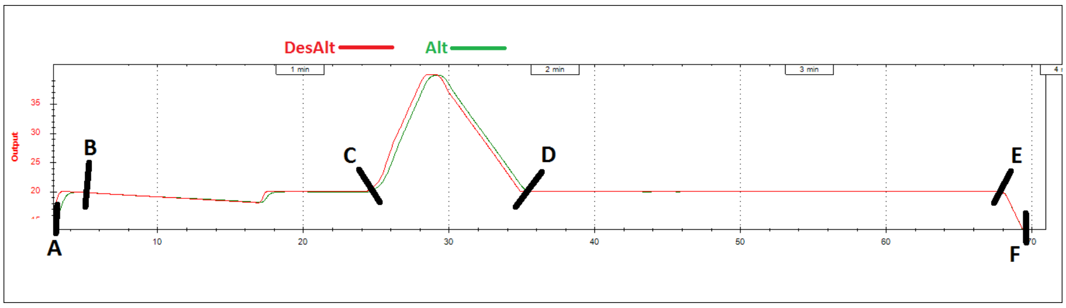


Figure 5.11: Altitude graph

Lets consider now the two paths where the vehicle must stay stable at 20 meters (path B-C and D-E of Figure 5.11). Here we can see an improvement introduced with the reactive version. The error in an ideal system must be always 0. In this case, looking the Table 5.6 , we can see that the copter is more stable with the reactive firmware. The difference of error values is about 19%.

5.2 Code execution

In this section we are going to extract data from the Pixhawk board. It is not needed anymore to simulate a flight with SITL, data are extracted from a real one. The flight in this case (using a radio control to send inputs to the quadcopter) consists in a simple flight of 20 seconds. It is impossible to execute the same set of commands in the two cases (with the original and the reactive firmware), but it is not necessary. The goal of this test is to calculate the average time the main and fast loop take to be executed.

The time of fast loop is expected to be lower in the reactive version because we have cut off functions that are now managed from the check_sensors() function (see Section 4.3) on the main loop. The idea is to store data using the DataFlash library given from ardupilot project. In Listing 5.1 we have added some lines of code on the ArduCopter.pde.

In each loop, the numbers of clocks are memorized on count_main_loop variable and then stored by DataFlash with the Log_Write_Data(ID, value) function on the log file. We have extracted those values from the two log files using the two versions of firmware. In the first case (original version) the average number of clock cycles for the main loop was 420990 and in the second case (reactive version) the average

Listing 5.1 DataFlash logs

```
/* clock counter for main and fast loop */
uint32_t count_main_loop = 0;
uint32_t count_fast_loop = 0;

/* register address */
volatile unsigned int *DWT_CYCCNT = (unsigned int *)0xE0001004;

void loop(){
        count_main_loop = *DWT_CYCCNT;
        Log_Write_Data(ID_main_loop, count_main_loop);
        ...
        ...
}

void fast_loop(){
        count_fast_loop = *DWT_CYCCNT;
        ...
        ...
        Log_Write_Data(ID_fast_loop, *DWT_CYCCNT - count_fast_loop);
}
```

	clock count	duration [ms]
original	420990	2,505
reactive	420057	2,500

Table 5.7: Main loop duration

was 420057. Each clock have a duration of 5,95ns (Pixhawk processor at 168 MHz). So, we can calculate the main loop time multiplying the number of clocks with the duration of a single one (see Table 5.7)

In this test the reactive version is 5 microseconds better than the original one but we think that doing more tests and stressing more the quadcopter, those two values are going to be always very similar with some difference in order of microseconds. The same test is done for the fast_loop function (Table 5.8). In the original version the average number of clock cycles was 62719 and in the reactive one was 19567. So the result of fast loop duration is about 0,37ms on the original version and about 0,11ms on the reactive one.

In other words, the code execution of the main loop has remained almost unchanged and the code execution of the critical part of the main loop (the fast loop) is decreased widely.

	clock count	duration [ms]
original	62719	0,3732
reactive	19567	0,116

Table 5.8: Fast loop duration

5.3 Summary

As we saw before, the performance is significantly increased on the reactive version. The use of the data flow (from sensors to the motors input) has been optimized updating object values only when new data is available. It is verified especially when the vehicle should maintain at least one of the four parameter (pitch, roll, yaw or altitude) constant. In this case, it is very simple to do an evaluation on the difference between the real and the desired value. Therefore, the performance can be an important reason to use a reactive programming paradigm.

In this thesis we modify only a little part of Arducopter project having the performance explained before. Modifying all the project (or creating it from scratch) with the reactive programming it can be a very significant gain in performance. By creating from zero the tasks called from the scheduler (witch consists in sensor reading signals) and divide them on groups by their frequency we can have a very optimized control system. These modifications can be done in the other sub-projects of the Ardupilot, such as ArduPlane or ArduRover.

In the hobbyist field, the precision may not be significant. For a pilot, that uses a quadcopter only to play, does not matter if the vehicle has an error in yaw lower then 10% compared to the original firmware. Same thing can not be said if the vehicle is for professional or military use. If the vehicle is for professional aerial filming, zooming in, the vibration affects the quality of the video. Another example are the quadcopters used from the Italian army for the protection of military and civilian convoys [6]. These quadcopters must collect, process and transmit with maximum reliability and precision images and geo-referenced data on the dangers found around the convoy. These examples explain how important can be the precision in some scenarios.

Another reason to choose to use a reactive programming paradigm in Ardupilot is the code writing. Lets analyze the code making some simple examples. The original firmware has its typical structure with:

- setup() where all objects are initialized
- fast_loop() where the critical functions are called
- scheduler that runs tasks about sensor readings and functions with their respective frequency call

- main loop() that calls the fast loop and then calls the scheduler giving the remaining time available

In Listing 5.2 we can see a simple code that describes the ArduCopter.pde. In Listing 5.3 is shown the other version using the reactive library. Some of the critical instructions of fast_loop() are deleted because their work is done by the propagation of sensors values. For example, adding a new task check_sensors() to the scheduler that runs at any loop or to the main loop(), all the time a sensor changes, will change all the variables that depend from that sensor. So the fast_loop() will be lighter. This enables the scheduler to execute more tasks related to the control system, like ordinary checks for the failsafe mode or tasks related to other additional sensors.

Let's analyze those two styles of writing code on Ardupilot project. This project, as we said before, is very large and supported by a large community. Writing a code using a reactive programming paradigm help others understand better what is the use of a single sensor value. In the original version it is more difficult to see where the GPS values are used on the project. On the reactive one is sufficient to see on the setup() the signal referred to this sensor. The signal (the GPS signal in this case) gives all the behavior of this sensor (all the functions or objects depending on this sensor) allowing developers to simply trace down the dependency between objects. If it is necessary to change things related to sensor values, it is possible to change the behavior of this sensor inside the setup() phase without searching where this sensor is used on the entire code. Therefore, using reactive programming paradigm on Ardupilot project makes the code more easy to understand.

Listing 5.2 Original ardupilot example

```
const AP_Scheduler::scheduler_tasks[] PROGMEM = {
        { rc_loop,                4,     130 },
        { throttle_loop,          8,      75 },
        { update_GPS,             8,     200 },
        { update_batt_compass,   40,     120 },
        { read_aux_switches,     40,      50 },
        { arm_motors_check,      40,      50 },
        ...
        ...
};

void setup(){
        // variables initialisation
}

void loop() {
        // block of istructions

        // Execute the fast loop
        fast_loop();

        // run all the tasks on the available time
        scheduler.run(time_available);
}

void fast_loop(){
        // critical instructions that should be always
        // executed
        f1();
        f2();
        f3();
        ...example
        ...
}

/* all the functions called from the scheduler */
```

Listing 5.3 Reactive ardupilot example

```
const AP_Scheduler::scheduler_tasks[] PROGMEM = {
        { check_sensors,          1,     1 },
        { rc_loop,                4,   130 },
        { throttle_loop,          8,    75 },
        { update_GPS,             8,   200 },
        { update_batt_compass,   40,   120 },
        { read_aux_switches,     40,    50 },
        { arm_motors_check,      40,    50 },
        ...
        ...
};

void setup(){
        // variables initialisation

        alert::stream gps_state;

        void (*_alert1)(void) = &read_AHRS;
        void (*_alert2)(void) = &do_something;

        gps_state.bind(_alert1);
        gps_state.bind(_alert2);

}

void loop() {
        // block of istructions

        // Execute the fast loop
        fast_loop();

        // run all the tasks on the available time
        scheduler.run(time_available);
}

void fast_loop(){
        // critical instructions that should be always
        // executed
        f1();
        //f2();
        //f3();
        ...
        ...
}

// all the functions called from the scheduler
```

Chapter 6

Conclusions & Future Work

As illustrated in this thesis, the work is positioned in the context of UAVs and reactive programming paradigm. The goal was the re-engineering of Ardupilot using the reactive programming paradigm.

In Chapter 2 is shown the state of the art divided in two main topics. The first one is related to the reactive programming. A brief description is given, also there are listed the main properties and some tools that use this paradigm (in C++). The second topic is related to multicopters. In this thesis we have used only a quadcopter, so in Chapter 2 are listed the main things that characterize a full multicopter suite for developers (hardware, firmware, software and other tools used for simulated flights).

In Chapter 3 are shown how the Ardupilot code is structured (main loop, fast loop, scheduler), how the code can be executed and some of the flight modes used from the simulator during the evaluation phase.

Chapter 4 presents the obstacles found during the implementation phase. In the first part (Section 4.1) the problems are shown (listing some of the limits of the existing tools) and one solution is presented and also is implemented (the ReactiveCpp library). Then, this solution is used in an example of control loop to show how can be used. Finally, in the last section of this chapter, the Ardupilot project is modified using the ReactiveCpp library.

In Chapter 5 are shown some tests used to evaluate our work. We have used two flights for each test (one with the reactive firmware and one with the custom one). From the flight logs, data are extracted and compared. For the altitude, pitch, roll and yaw we have compared the error (the difference between the real value and the desired value) for each parameter and then we have shown them with some charts and tables that allowed the distinction between the two firmwares. We have demonstrated that our custom firmware is more precise at positioning the vehicle decreasing the error (the difference between the desired value and the real value of a parameter) up to 35%. Then a significant test was done on the code execution verifying that the time execution of the main loop remains almost equal and the time of the fast loop (the critical part of the code) is decreased drastically. This enables the scheduler to have more available time to execute tasks.

The goal of this work was to customize a control loop of a drone with the reactive programming. From the state of the art, studying the existing tools we came to the conclusion that these tools can not be used on Ardupilot. As a further contribution we developed a brand-new open-source library (the ReactiveCpp), avoiding all the hardware and software limitations.

However, the ReactiveCpp has some limitations that in any case does not affect this work. The main limits of this library are:

- glitch avoidance is not satisfied
- "uncomfortable" code on the initialization phase

For the first one, a solution is to implement the library using threads for the concurrency access. When a variable depends from two parent variables, when one parent changes, it must wait for the other parent before the propagation takes place. In this moment only the Pixhawk board support the multithreading.

The second "problem" can be avoided with the C++11 standard that is going to be supported by Ardupilot shortly. In this case, instead of creating a function pointer to pass to the bind list, with lambda expression, functions can be passed in one single step like in Listing 6.1.

Listing 6.1 Using C++11 standard

```
alert::function<void(int)> _alert([](int value){
        read_AHRS();
        do_something()
        print_values());

gps_state.bind(_alert);
```

Another challenge can be the re-implementation of the entire ArduCopter project. The part about the libraries and HAL (Hardware Abstraction Layer) can still remain unchanged, the part about the sensor management and the command decision can be restructured from zero using reactive programming.

Bibliography

[1] 3DRobotics. 3drobotics. `http://3drobotics.com/`, 2015.

[2] Stuart M Adams and Carol J Friedland. A survey of unmanned aerial vehicle (uav) usage for imagery collection in disaster research and management. In *9th International Workshop on Remote Sensing for Disaster Response*, 2011.

[3] Ardupilot. Ardupilot APM. `http://ardupilot.com/`, 2015.

[4] Ardupilot. Copter introduction. `http://copter.ardupilot.com/wiki/introduction/`, 2015.

[5] Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter. A survey on reactive programming. *ACM Computing Surveys (CSUR)*, 45(4):52, 2013.

[6] Cobra. Italian army - COBRA. `http://www.quadricottero.com/2014/09/mini-droni-vincolati-per-la-protezione.html`, 2015.

[7] Gregory H Cooper and Shriram Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *Programming Languages and Systems*, pages 294–308. Springer, 2006.

[8] Gregory Harold Cooper. *Integrating dataflow evaluation into a practical higher-order call-by-value language*. PhD thesis, Brown University, 2008.

[9] Frank Dabek, Nickolai Zeldovich, Frans Kaashoek, David Mazières, and Robert Morris. Event-driven programming for robust software. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, pages 186–189. ACM, 2002.

[10] DIY Drones. Diydrones. `http://diydrones.com/`, 2015.

[11] Conal Elliott and Paul Hudak. Functional reactive animation. *ACM SIGPLAN Notices*, 32(8):263–273, 1997.

[12] HobbyKing. Hkpilot32. `http://www.hobbyking.com/hobbyking/store/__55561__HKPilot32_Autonomous_Vehicle_32Bit_Control_Set_w_Power_Module.html`, 2015.

[13] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. Arrows, robots, and functional reactive programming. In *Advanced Functional Programming*, pages 159–187. Springer, 2003.

[14] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. *Aspect-oriented programming*. Springer, 1997.

[15] Ingo Maier, Tiark Rompf, and Martin Odersky. Deprecating the observer pattern. Technical report, 2010.

[16] Leo A Meyerovich, Arjun Guha, Jacob Baskin, Gregory H Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. Flapjax: a programming language for AJAX applications. In *ACM SIGPLAN Notices*, volume 44, pages 1–20. ACM, 2009.

[17] Microsoft. Rxcpp library. `https://github.com/Reactive-Extensions/RxCpp`, 2015.

[18] Nuttx. Real-time operating system. `http://www.nuttx.org/`, 2015.

[19] PaparazziUAV. Paparazzi project. `https://wiki.paparazziuav.org/wiki/Main_Page`, 2015.

[20] James Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism.* " O'Reilly Media, Inc.", 2007.

[21] Schlangster. cpp.react library. `http://schlangster.github.io/cpp.react/`, 2015.

[22] Sodium. Sodiumfrp library. `https://github.com/SodiumFRP/sodium`, 2015.

[23] Michael Sperber. *Computer-assisted lighting design and control.* PhD thesis, Universität Tübingen, 2001.

[24] Michael Sperber. Developing a stage lighting system from scratch. In *ACM SIGPLAN Notices*, volume 36, pages 122–133. ACM, 2001.

[25] Guy L Steele Jr. The definition and implementation of a computer programming language based on constraints. 1980.

[26] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

[27] New York Times. Amazon delivers some pie in the sky. `http://www.nytimes.com/2013/12/03/technology/amazon-delivers-some-pie-in-the-sky.html`, 2013.

[28] Wired. Wired magazine. `http://www.wired.com/category/magazine/1`, 2015.