Design and integration of an Unmanned Ground Vehicle into the USAL architecture

Ricard Morales Pedro

SUPERVISED BY

Pablo Royo Chic
Raúl Cuadrado Santolaria

Universitat Politècnica de Catalunya
Master in Aerospace Science & Technology
December 2013
This Page Intentionally Left Blank
Design and integration of an Unmanned Ground Vehicle into the USAL architecture

BY
Ricard Morales Pedro

DIPLOMA THESIS FOR DEGREE
Master in Aerospace Science and Technology

AT
Universitat Politècnica de Catalunya

SUPERVISED BY:
Pablo Royo Chic
Raúl Cuadrado Santolaria

Department of Computer Architecture
ABSTRACT

The ICARUS research group activities at Technical University of Catalonia (UPC) are aimed at the automation and development of on-board avionics and ground systems, supporting both manned and Unmanned Aircraft Systems (UAS). Accordingly, they have developed a modular and reconfigurable architecture called USAL, which provides flexibility and reduces development costs of UAS missions. At this moment, they are planning to go from their simulated environments (ISIS+) to real UAS scenarios, being this the starting point of this Master Thesis.

Within the abovementioned context, the composition of this document is targeted to provide a reliable and safe test bed platform for the ICARUS USAL architecture, in such a way that they could perform extensive testing of all their systems in a regular basis. Because of its strong similarities with Unmanned Aircrafts (UAs), the approach of developing an Unmanned Ground Vehicle (UGV) has been proposed, being an intermediate step between the simulated scenario and an eventual UA.

To be started, the alternatives for the test bed platform have been analysed, taking into account factors like safety, time, costs, and simplicity. By using low-cost components we have been able to set up a simple UGV platform, thus fulfilling the initial requirements of a UA simplification.

Once we had the UGV, we performed its integration into the USAL architecture, managing the autopilot through the ICARUS ground stations. To do so, extensive testing has been carried out with an intermediate solution, ensuring robustness of the final development.

As the final step for the test bed platform, the creation of a completely autonomous vehicle was also necessary. Therefore, we designed a real, flexible, and exchangeable “Air Segment” for our UGV, providing reusability and portability to any type of vehicle.

Additionally, apart from the development of the UGV, we addressed the implementation of a reliable communications system, which will operate between the ground station and the vehicle. In this sense, we have developed a new protocol for the USAL architecture, endowing it with enhanced Quality of Service (QoS). As well as in all stages of this thesis, real tests have helped us to ensure the correct behaviour of our solution, analysing in this case the different radio devices.

And finally, as the primary objective was to build a test bed platform, we performed as much tests as we could of the USAL architecture. By doing so we not only proved our solution fulfilled the USAL testing requirements, but also allowed us to extract different diagnostics and conclusions of the ICARUS USAL architecture.
# Table of Contents

INTRODUCTION ........................................................................................................................................ 1

Chapter 1 DEFINING A USAL TEST BED PLATFORM ............................................................................ 3
  1.1. Motivation ......................................................................................................................................... 3
  1.2. The framework: USAL ....................................................................................................................... 3
  1.3. The starting point and expected work ................................................................................................. 6
  1.4. The UAS Test bed platform ............................................................................................................... 10

Chapter 2 THE UGV PLATFORM ............................................................................................................ 12
  2.1. Unmanned Ground Vehicles ........................................................................................................... 12
    2.1.1. The autopilot decision ................................................................................................................ 13
  2.2. The APM:Rover Autopilot ............................................................................................................... 13
    2.2.1. The origins: Arduino .................................................................................................................... 13
    2.2.2. The hardware: APM 2.5 Board .................................................................................................. 14
    2.2.3. The Firmware ............................................................................................................................. 15
    2.2.4. The implications of APM:Rover ................................................................................................. 16
  2.3. The Vehicle ..................................................................................................................................... 16
  2.4. AP Communications Protocol: MAVLink ......................................................................................... 17
    2.4.1. MAVLink Features ...................................................................................................................... 18
    2.4.2. MAVLink Packets ....................................................................................................................... 18
    2.4.3. Developing with MAVLink ......................................................................................................... 19
  2.5. Testing the UGV ............................................................................................................................. 20
    2.5.1. The Ground Stations .................................................................................................................. 20
    2.5.2. MAVLink in APM:Rover: Analysis ............................................................................................ 22
    2.5.3. Assembling the APM:Rover Platform ....................................................................................... 23
    2.5.4. First tests results ....................................................................................................................... 24
    2.5.5. First tests conclusions .............................................................................................................. 25

Chapter 3 USAL INTEGRATION .............................................................................................................. 27
  3.1. The Virtual Autopilot System (VAS) ................................................................................................. 27
  3.2. The Communication Manager ......................................................................................................... 28
    3.2.1. The starting point: a standard MAVLink library ....................................................................... 29
    3.2.2. MAVLink Communication Manager ........................................................................................ 30
  3.3. The Autopilot Manager .................................................................................................................... 33
    3.3.1. Specificities of a Rover APL ...................................................................................................... 33
    3.3.2. Automatic Navigation in APM:Rover ....................................................................................... 35
    3.3.3. The APM:Rover Autopilot Manager ......................................................................................... 37
  3.4. Testing the APM:Rover VAS ............................................................................................................ 37

Chapter 4 ASSEMBLY OF THE AIR SEGMENT .................................................................................... 39
  4.1. Architecture of the Rover ................................................................................................................. 39
  4.2. The On-board computer .................................................................................................................. 40
    4.2.1. CISC or RISC ............................................................................................................................. 40
    4.2.2. Choosing a mobile on-board computer ..................................................................................... 42
    4.2.3. Setting up the PandaBoard ....................................................................................................... 44
  4.3. The Software: Porting .NET C# to Linux .......................................................................................... 45
    4.3.1. The Mono Project ....................................................................................................................... 45
    4.3.2. ARM Hard Float and Soft Float ................................................................................................. 45
    4.3.3. Adapting the VAS to Ubuntu ..................................................................................................... 46
  4.4. The Power Subsystem ...................................................................................................................... 48
4.5. Assembly of the platform ................................................................................. 50
  4.5.1. First assembly .......................................................................................... 50
  4.5.2. Box design ............................................................................................ 51

Chapter 5 THE COMMUNICATIONS SUBSYSTEM ........................................... 56

5.1. Serial Communications .................................................................................. 56
  5.1.1. Testing MAREA Serial configuration ...................................................... 57
  5.1.2. MAREA Serial Messages ........................................................................ 58

5.2. Wireless Communications ............................................................................. 60
  5.2.1. Available radio modules .......................................................................... 60
  5.2.2. Range Comparison .................................................................................. 64
  5.2.3. Testing MAREA Serial Wireless configuration ....................................... 67

5.3. MAREA Serial Transport Protocol ................................................................. 67
  5.3.1. Byte error detection ................................................................................ 68
  5.3.2. Enabling Safe Retransmission ................................................................. 69

5.4. Data Link Enhancements ............................................................................... 72
  5.4.1. Packet Fragmentation .............................................................................. 72
  5.4.2. Reducing Latency ................................................................................... 73
  5.4.3. Dual Serial Transport ............................................................................. 74

5.5. Software implementation ............................................................................... 74
  5.5.1. Testing the Serial Transport Protocol ..................................................... 76

Chapter 6 USAL TESTS AND CONCLUSIONS .............................................. 77

6.1. Tests environment ......................................................................................... 77
  6.1.1. Ground Station ....................................................................................... 77
  6.1.2. Flight Plans: Testing spots ..................................................................... 78
  6.1.3. Entire architecture .................................................................................. 80

6.2. Tests results and modifications ..................................................................... 81
  6.2.1. Middleware communications ................................................................. 81
  6.2.2. Virtual Autopilot Service ....................................................................... 82
  6.2.3. Flight Plan Manager (FPMa) ................................................................. 83
  6.2.4. Flight Plan Monitor (FPMo) .................................................................. 83
  6.2.5. Flight Monitor (FMo) ............................................................................ 84

6.3. Conclusions .................................................................................................. 85
  6.3.1. Defining the UAS test bed platform ...................................................... 85
  6.3.2. The APM:Rover .................................................................................... 85
  6.3.3. The USAL integration .......................................................................... 85
  6.3.4. The assembly of the Air Segment ......................................................... 86
  6.3.5. The communications subsystem ......................................................... 87
  6.3.6. The final tests ....................................................................................... 87

6.4. Future Work ................................................................................................ 88

BIBLIOGRAPHY AND REFERENCES ............................................................. 90
List of Figures

Figure 1.1 Marea “IService” interface ............................................................................. 4
Figure 1.2 Overview of the typical USAL services and its four categories ..................... 5
Figure 1.3 Minimum USAL services for a UAS environment .......................................... 6
Figure 1.4 Expected tasks to perform in the VAS service (outlined in red) .................... 7
Figure 1.5 Master Thesis global picture: expected tasks are red-coloured ..................... 9
Figure 2.1 UGV mandatory elements and their typical interrelations ............................. 12
Figure 2.2 Arduino Mega 2560 board (left) and APM 2.5 (right) ................................. 14
Figure 2.3 The Turnigy 110BS RC Car ........................................................................ 17
Figure 2.4 Futaba T7C R/C System: Transmitter and Receiver ................................... 17
Figure 2.5 MAVLink packet structure .......................................................................... 18
Figure 2.6 MAVLink XML message to C++ library example ........................................ 19
Figure 2.7 APM Mission Planner screenshots: Flight Planner and Firmware tabs ........... 21
Figure 2.8 QGroundControl screenshots: Flight Planner and Logger tabs .................... 21
Figure 2.9 MAVLink messages in a typical APM:Rover steady state ............................. 22
Figure 2.10 APM 2.5 Board connections for APM:Rover implementation ........................ 23
Figure 2.11 RC Car, APM 2.5, GPS and 3DR Telemetry Radio assembled ........................ 23
Figure 2.12 Navigation loop and waypoint management of APM:Rover ........................ 25
Figure 3.1 Typical VAS service structure ..................................................................... 28
Figure 3.2 Particular VAS service structure for our APM:Rover autopilot ...................... 28
Figure 3.3 Mavlink.dll file UML representation .......................................................... 29
Figure 3.4 Mavlink Manager UML representation ....................................................... 31
Figure 3.5 “IAutopilot” Interface ................................................................................ 33
Figure 3.6 VAS states and APM:Rover modes adaptation .......................................... 35
Figure 3.7 Waypoint queue management in automatic navigation ............................... 36
Figure 3.8 APM:Rover Autopilot Layer structure and relations ..................................... 37
Figure 3.9 Intermediate architecture to test APM:Rover APL ..................................... 38
Figure 4.1 Basic elements of our Rover “Air Segment” ................................................ 39
Figure 4.2 Energy consumption comparison (in Watts) of Intel Atom N450 and ARM Cortex A8 (processors with similar computing power) [22] .................. 41
Figure 4.3 Preinstalled Ubuntu images available for the Panda Board [26] ..................... 44
Figure 4.4 VAS Console used in Windows OS .............................................................. 47
Figure 4.5 Power subsystem main and internal connections (red and green respectively) ... 48
Figure 4.6 Lithium-Polymer batteries ......................................................................... 49
Figure 4.7 ESC (left) and 5V DC Regulator (right) ...................................................... 49
Figure 4.8 First Assembly of the Air Segment ............................................................. 50
Figure 4.9 CAD Illustrations for the Air Segment box..........................................................53
Figure 4.10 Connectors and visible LEDs detail of the final box...........................................53
Figure 4.11 Wooden first box prototypes ................................................................................54
Figure 4.12 Final box mounted in our Turnigy platform..........................................................55
Figure 5.1 UART Signal Example [39] ...................................................................................57
Figure 5.2 Available radio modules (3DR, XBee, XTend, and RN-XV)........................................60
Figure 5.3 433MHz Stock antenna tested with the spectrum analyser ....................................66
Figure 5.4 433MHz Designed antennas tested with the spectrum analyser ..............................66
Figure 5.5 Packet Framing of the MAREA Serial Protocol .....................................................69
Figure 5.6 MAREA Serial Retransmission Protocol ...............................................................70
Figure 5.7 MAREA Serial packet structure ..........................................................................72
Figure 5.8 Serial Packet Structure (Safe and Fast Messages)..................................................73
Figure 5.9 Serial Packet Structure (ACK and NACK messages)............................................73
Figure 5.10 Dual Serial Transport solution for high data rates ..............................................74
Figure 5.11 MAREA Serial Transport C# Classes.................................................................75
Figure 6.1 Flight Monitor (FMo) interface ..............................................................................77
Figure 6.2 Flight Plan Monitor (FPMo) interface ....................................................................78
Figure 6.3 Flight Plan basic structure ......................................................................................79
Figure 6.4 Flight Plans at Castelldefels (Barcelona) ...............................................................80
Figure 6.5 Test bed platform overview ..................................................................................80
Figure 6.6 Scan Update tab .....................................................................................................84
List of tables

Table 1.1 Examples of some USAL Data Messages ................................................................. 4
Table 2.1 MAVLink packet detailed structure ........................................................................ 19
Table 4.1 Comparison of CISC and RISC architectures [21] ............................................. 40
Table 4.2 Comparison of the RISC-based available boards ............................................... 43
Table 4.3 Final box specifications ....................................................................................... 54
Table 5.1 Available radio modules comparison ................................................................... 61
Table 5.2 Radio results summary ......................................................................................... 63
Table 5.3 Numerical comparison of the radio ranges ............................................................ 65
This Page Intentionally Left Blank
INTRODUCTION

Composed within the ICARUS research group at Technical University of Catalonia (UPC), this Master Thesis is the result of the work carried out along this year to develop a test bed platform for their Unmanned Aircraft System (UAS) architecture: USAL. Standing for UAS Service Abstraction Layer, it can be defined as a hardware and software abstraction layer providing system flexibility and reusability of the components, thus reducing development costs and time-to-market.

Being the starting point of this document the definition of the scenario, we are going to respond to one of the problems ICARUS was facing in their research activities: moving from a simulated environment to the real world presents some difficulties, which they need to be somehow solved. To do so, we are going to present the design of an Unmanned Ground Vehicle (UGV), which will allow us to gather the required information and to easily test all the different UAS systems.

To be started, in the first chapter the entire context of this work will be introduced, together with the full picture of its expected areas of development. Then, the arguments for deciding to go for the UGV (or rover) approach, and the election of a suitable autopilot, will both be presented.

In the case of the second chapter, it will be focused to all the concepts related to the chosen autopilot: its origins, features, typical ground stations, etc. In this sense, we will see more in detail the characteristics of its communication protocol, and we will perform some testing to know the real capabilities it can provide us.

After the autopilot, it will be the turn for its integration within USAL, being here the link between the autopilot and USAL (the Virtual Autopilot System or VAS) the cornerstone of our work. We will explain things like how the integration will be performed, how it will be structured, which adaptations for the rover we will have to make, which adjustments, etc. Finally, we will test everything with the real car, ensuring this part is adequately completed.

Once we have the VAS service working properly, we will add the entire USAL architecture. To do so, we will design the “Air Segment”, that is the components required on the car to perform autonomous navigation (like the on-board computer, the autopilot, the GPS, etc.). The issues we will confront and the implemented solutions will be all explained, ending with the assembly of the final “Air Segment” solution.

As an additional point, and because it is a core component of any UAS, the fifth chapter is dedicated to the communications subsystem. We are going to start with the initial features of the USAL communications and with the tests of the available radio link devices, to then introduce and develop the Serial Transport Protocol, which will enable a reliable data link between the “Air Segment” and the “Ground Station”.

And finally, as the primary objective is the development of the test bed platform, what we will do is to perform as much tests as possible of the USAL architecture. In this
way, we will prove our solution will be feasible, thus finding the USAL limits, know its limitations, and proposing the consequent improvements.
Chapter 1

DEFINING A USAL TEST BED PLATFORM

To start with this Master Thesis, this first chapter will introduce the global picture of all the work performed to develop this USAL test bed platform. Beginning with its motivation, we will see the framework and the starting point of this endeavour, to eventually decide the preferred pathway with the selection of the vehicle.

1.1. Motivation

This master thesis has been composed within the ICARUS group at Universitat Politècnica de Catalunya, whose activities are aimed at the automation and development of on-board avionics and ground systems, supporting both manned and Unmanned Aircraft Systems (UAS).

As a headline, they have designed a modular and reconfigurable architecture for UAS missions called USAL, which allow them to reduce “time to market” when creating a new UAS or mission while providing reusability of the components (see [1] and [2]). In this context, they have been extensively developing and testing this architecture in simulated environments, by using an aircraft simulator program as their aerial platform (the simulator environment is named ICARUS Simulation Integrated Scenario or ISIS+ [3]).

However, they want to start moving this architecture to a real scenario, and that is why they have yet started testing some components in real helicopters. But these real tests are not affordable in a regular basis, being complex and pretty expensive in terms of time and costs (details are given in section 1.5). Hence, the ICARUS group needs a real (but safe) platform to perform intensive testing of their systems; this is where this project started, intending to fulfil this “must have” or Top Level Requirement (TLR):

“To build a vehicle capable of being used as a test bed platform and to integrate it into the ICARUS software architecture”

1.2. The framework: USAL

USAL is the acronym of UAS Service Abstraction Layer, and is the framework of this entire thesis: the software architecture the ICARUS group has been working on for the last years.

Typically, UAS make use of the advantages of small/medium aircrafts (mainly costs, endurance and flexibility) for very much different kind of purposes: remote sensing and monitoring, aerial video and photography, search and rescue support,
communications relay, etc. But there are plenty of approaches and solutions engineers can take, what created the need for some kind of flexible and reusable architecture. This was the origin of USAL, a service-oriented framework which tries to harmonize the hardware and message data types of UAS architectures [2] (see next table 1.1).

### Table 1.1 Examples of some USAL Data Messages

<table>
<thead>
<tr>
<th>Message</th>
<th>Description</th>
<th>Values</th>
<th>Data Type</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>Position</td>
<td>WGS84 Geo-position</td>
<td>Latitude Longitude Altitude</td>
<td>Double</td>
<td>Radians Radians Meters</td>
</tr>
<tr>
<td>Angles</td>
<td>Aircraft Angles</td>
<td>Roll Pitch Yaw</td>
<td>Float</td>
<td>Radians Radians Radians</td>
</tr>
<tr>
<td>IdWp</td>
<td>Waypoint Identifier</td>
<td>IdRef IdLeg IdStage</td>
<td>Uint Uint</td>
<td>N/A N/A N/A</td>
</tr>
<tr>
<td>Waypoint</td>
<td>USAL Waypoint</td>
<td>IdWp Latitude Longitude Altitude Speed FlyOver</td>
<td>Object Double Double Float Float Bool</td>
<td>N/A Radians Radians Radians Meters Metery/sec N/A</td>
</tr>
</tbody>
</table>

To define what is a service-oriented framework, we need to explain that in software development services are defined as applications following a set of defined procedures (in USAL, for instance, every service must follow an “IService” interface, which is available in the following figure 1.1). This is important because USAL architecture is primarily based in a Middleware called MAREA (Middleware Architecture for Remote Embedded Applications [4]), which is the responsible for connecting publishers (services sending information) with subscribers (consumers of information).

### Figure 1.1 Marea “IService” interface
Continuing with the definition of USAL, it structures the different services along four categories: Mission, Payload, Awareness, and Flight Category. The most common services are proposed in [2] and can be seen in the following figure 1.2, although this classification encompasses all the possible applications that could eventually be proposed.

![Figure 1.2](image)

Figure 1.2 Overview of the typical USAL services and its four categories

Overall, using this approach based on services makes USAL to be a modular architecture: this means that when someone wants to set up a UAS mission he can select the services he wants to run. However, there are services that will be present in any mission: these are the Virtual Autopilot System (VAS), the Flight Plan Manager (FPMa), the Flight Monitor (FMo) and the Flight Plan Monitor (FPMo). Their mission is summarised next:

- The VAS is the responsible for the interaction with the autopilot hardware: it adapts or converts the data types, implements the USAL flight states, and handles the correct management of the waypoints for any autopilot. It is specific to every autopilot.

- The FPMa is in charge of translating Flight Plans into queues of Waypoints (which are then sent to the VAS), providing much richer flight plan capabilities on top of the autopilot (unlimited waypoints, parameterized scans, holds, reactions, emergencies, etc.).
• The FMo is the ground station for the Pilot-In-Command, managing the change of flight states and the external devices (like a Joystick).

• And finally, the FPMo is the ground station handling the flight plans (allowing us to upload and change them, skip elements, etc.).

Regarding the other categories, and as they provide primarily additional functionalities and robustness to the system, they are not strictly necessary for a “basic” use of the USAL architecture. In fact, with the services classified as “Flight Category” we already have the two mandatory components for any UAS (figure 1.3): the “Ground Station” (or “Ground Segment”) and the “Air Segment” (or “On-board Components”).

![Figure 1.3 Minimum USAL services for a UAS environment](image)

Being said that, and as our goal in this thesis is to build a test bed platform for all the USAL services, we will focus in getting this “Lite” version of USAL working properly, which will be a solid base from which eventually test the rest of the services.

### 1.3. The starting point and expected work

By the moment this project started, the USAL architecture was not completely developed. However, and considering that there exists a constant improvement and testing work done by the group itself, the FMo, FPMo and FPMa were already working solutions.

In the case of the VAS, and as some parts are specific to the autopilot used, it was fully developed for the X-Plane Simulator software. In this sense, changing to a real autopilot (no matter which one) will mean to implement the adaptation of this service, interfacing with it and providing the USAL functionalities required. Prior to that, we
should decide the vehicle we are going to use, thus selecting a suitable autopilot. All these tasks can be observed in figure 1.4, where the expected features we will need to develop are painted in red for better visualisation.

![Figure 1.4 Expected tasks to perform in the VAS service (outlined in red)](image)

Once the work with the VAS and the autopilot will be completed, it will be the time to place the whole “Air Segment” on the vehicle (computer, communications, etc.). This will present some obstacles that we will need to overcome, and they can be separated into two main tasks:

- On one side, the selection of the on-board computer will not be straightforward: there are some alternatives with its pros and cons. Therefore we will need to compare them to try to find the most suitable to our purposes.

- On the other side, we will need to place all the components in a small and compact form factor, in order to ensure that our solution will not be dependent on the platform or vehicle used. By doing this, it will allow ICARUS to reuse it and to place it in any other vehicle they may eventually want to test, providing flexibility and resulting in a long-term useful solution.

Finally, after all these concepts have been developed we will be facing another problem: the communication between Air and Ground Segments was still not tested in real scenarios (they have always been used in secure networks like Ethernet). This will present some new issues we will need to clear up, and which are summed up in the next paragraph.
Using Ethernet communications imply that they have not addressed yet topics like the loss of messages or the required data bandwidth, what is inherent to any real or weak link. Additionally, the range of Ethernet (Wi-Fi) devices is quite reduced, going typically to no more than 50-100 meters.

Consequently, there is the need of changing to serial long-range radio devices, comparing and choosing a suitable hardware, and of implementing a protocol to manage the retransmission and fragmentation of important packets. This will bring a reliable data-link to the USAL architecture, being possible to use it with any aerial vehicle.

And in the end, as the first objective of this Thesis is to develop a test bed platform, what we will do is to test the performance of the USAL architecture. By doing so, we will prove the feasibility of our solution, thus provide ICARUS with the first analysis and conclusions of their essential systems.

Finally, and to summarise the different tasks we have introduced, the expected work of this Master Thesis can summarised with the following points (and red-coloured in figure 1.5):

- Select an adequate vehicle to be used as our test bed (plane, rover, multi-copter, boat, etc.).
- Find and choose a suitable autopilot for the selected vehicle, explaining the decision and exposing the advantages and requirements.
- Manage the autopilot by creating a communication library, providing flexibility and exchangeability with further developments.
- Link the USAL architecture with the selected autopilot, implementing the required functionalities to exploit its capabilities.
- Select an on-board computer to place it in the vehicle, what will allow us to run some services there like in a real “Air Segment” (thus following the complete USAL architecture).
- Select the rest of “Air Segment” components and pack them in a small form factor, leading to a compact solution which could be used in any UAS.
- Develop a safe serial transmission link for USAL communications, allowing reliable data exchange between the ground station and the UAS vehicle. To do so, we will need to analyse the former USAL solution, designing a new protocol and adding some extra functionalities.
- Also related with the data link, we will need to select suitable hardware for our solution, carrying out extensive testing of the different alternatives.
- And eventually, the tests of the entire platform will be performed, providing different results and first conclusions for the USAL architecture.
Figure 1.5 Master Thesis global picture: expected tasks are red-coloured
1.4. The UAS Test bed platform

Following the path presented by the previous section, the first step will be to select the vehicle we are going to work with. To do so, we first need to analyse the main problems ICARUS was meeting in their tests:

- Regarding to time, and apart from the one invested in documentation (inherent to any research activity), UAS real tests require much work to be done prior to day of flight. For instance, time is spent in the assembly of the aircraft, in the test of every subsystem, in the travel to the testing spot, etc.

- Although tests are always performed with the aim of being successful, they come implicitly with the probability of getting errors. They can appear and be recovered by the robustness of the system, or sometimes they may lead to a complete failure. This failure gives no major diseases in simulated environments, but in the case you are flying at hundred feet above ground the consequences will at all times be catastrophic: breaking the instruments or the on-board computer(s), losing parts of the aircraft or even the whole vehicle, etc.

- And finally, as current legislation requires an area of segregated airspace to perform the test flights, it is required to apply for government permissions and to an adequate insurance coverage for the UAS operation.

Hence, we need to go for a vehicle which solves all these difficulties: on one side, it must fly at a very low altitude (or to not fly at all) to not break the instruments when errors arise; on the other side, it must also be safe for people and easy to set up to not invest much time in planning and permissions.

Among the range of possible vehicles (fixed-wing aircrafts, helicopters, multi-copters, dirigible airship, rover and ships), the rover approach provides us the following advantages:

- Firstly, it allows different navigation configurations: it can be tuned to perform straight point-to-point navigation with sharp turns (like helicopters and multi-copters) or to drive along smooth curves (like a plane). Besides, it can stay in a static position like it was in a hover/loitering manoeuver.

- Secondly, driving on the ground means it will never fall down and break any component, being the safest alternative in this sense (flying at very low altitudes with a, for instance, a multi-copter will be much more risky).

- Thirdly, because of the large availability of Radio-Controlled (RC) cars in the market, we will be able to develop a very low-cost rover platform in an easy and straightforward way, thus saving time and costs.
- And finally, to perform a test with a rover you may only need to go to a plain surface and turn it on; there is no need to have regulatory permissions or to have a particular far-away testing spot.

Consequently, we decided to develop a rover as our UAS test bed platform, trying to be as low-cost as possible. However, one may claim that rovers are not UAs and therefore not be suitable for our purposes, but we must state that they are called as Unmanned Ground Vehicle (UGV) because of their strong similarities with UAS. Some of them are summarised as follows:

- Regarding the nature of the components, a rover usually carries on the same basic elements of an UAV: power module, on-board computer, autopilot, payload, etc. As it happens with UAS, some of these elements are always optional and some are mandatory, but the architecture is almost the same.

- If we talk about navigation procedures, rovers are basically a simplified version of a plane, multi-copter, etc. The first one drives over a two-dimensional space, acting only over two factors: steering and throttle. On the contrary, flying vehicles move inside a three-dimensional space, resulting in much more complex navigation algorithms (the variables can be up to 8, depending on the number of control surfaces or rotors the vehicle has). Anyway, it is the autopilot the one in charge of managing the navigation, thus it does not affect the architecture of the system.

So concluding, and as UGVs are actually simplified UAVs, using this approach will be the safest, simplest and most inexpensive alternative to build our test bed.
Chapter 2

THE UGV PLATFORM

As it has been explained in the previous chapter, we have chosen to develop an Unmanned Ground Vehicle (UGV) as the test bed platform. To provide a sound basis for the future work, we will explain in this chapter the essential components of these vehicles, together with the implications they have in the design of the entire platform. Additionally, in order to ensure that a rover will suit our final requirements, we will introduce the conclusions extracted from the tests of the chosen UGV.

2.1. Unmanned Ground Vehicles

Being the simplified version of an Unmanned Aircraft (UA), UGVs are always composed by the following components: the vehicle with its motors and servomotors, an autopilot with its required sensors, and the communications and power subsystems. Because additional systems like sensors or external devices (called the “payload” as in UAV or space systems) are optional, we will not take them into account when developing our test bed platform.

In the next figure (2.1) this basic UGV structure is illustrated, along with the typical interrelations between them. There, the connectors in red indicate which are managed by the autopilot, and the ones painted in black are just the power supply connections to the different components.

![Figure 2.1 UGV mandatory elements and their typical interrelations](image)

Although we will explain all these components in detail in posterior sections, one can deduce from the figure above that the autopilot is the core component of any UGV. Being said that, and because it is the responsible of all the different subsystems, its election will pose some implications in the design of the whole platform, so it is necessary to start with the selection of an adequate autopilot.
2.1.1. The autopilot decision

Following with the abovementioned statement, we must first find an autopilot capable of performing the ground driving manoeuvres of a UGV. In this sense, we can address this condition in different ways:

- The first option is to use a traditional airplane autopilot with Taxi capabilities (ground movements around the Airport Platform Area). This means that we could take profit of this aeronautic stage to drive along any surface, with the expected performance being quite awkward though.

- The second is to use the air navigation of a standard UAV autopilot on the ground, adapting their outputs and expecting that the navigation procedures could behave correctly enough to drive the rover.

- And the third option is to use an autopilot dedicated to ground vehicles, which will give us better navigation patterns, and maybe different configurations.

Being obvious that the ideal choice was the third one, we made some research to try to find a suitable autopilot. However, we realised that there is only one commercial solution for manning small ground vehicles (there are some projects to drive real cars carried out by cars’ manufacturers like Toyota or BMW [5], but this was not our target): the Arduino-based “APM:Rover” (formerly known as ArduRover) which is the subject of the next section.

2.2. The APM:Rover Autopilot

The APM:Rover (see details in [6]) is one of the legs of the ArduPilot Project, which can be defined as an open-source series of autopilot platforms supported by the DIYDrones web [7] community members. It was created in 2007 by Mr. Chris Anderson (ex-editor of the U.S. technology magazine Wired [8]), and has evolved into a complete architecture for flying (and driving) different unmanned vehicles: planes, multi-copters (from helicopters to octo-copters) and rovers.

As the project is all open-source, you could for instance take the available schematics and build your own components, improve them, or even sell them. Nevertheless, 3D Robotics [9] is the company behind DIYDrones, and it is the major seller of every gadget related to the ArduPilot project: boards, sensors, airframes, etc.

2.2.1. The origins: Arduino

ArduPilot autopilots are based in Arduino [10] boards: an open-source electronics prototyping platform that has become very popular in the recent years due to its inexpensive cost and its ease of use.
Arduino is actually a microcontroller, a circuit board that has a microchip in it that can be programmed to do many different things. For instance, it may allow you to read information from sensors (photo-resistors, motion sensors, GPS, etc.), to control devices, or to display and store data.

The programming of the Arduino boards is done using the Arduino programming language. This language is based in another open-source programming framework for microcontrollers called Wiring, which is easy to use and has some similarities with C. Moreover, it provides the user with free software to code and flash the board, which is named the Arduino Development Environment software.

2.2.2. The hardware: APM 2.5 Board

Since the first prototype commercialised in 2009, the ArduPilot project has evolved and has designed a series of electronic boards. The most recent is the ArduPilot Mega 2.5, which is based in an “Arduino Mega 2560” board (both can be seen in the next figure 2.2).

![Figure 2.2 Arduino Mega 2560 board (left) and APM 2.5 (right)](image)

The characteristics of the APM 2.5 board can be summarised with the following points:

- It runs on a 16MHz Atmel ATMega2560 processor, which has a built-in hardware failsafe function (it can recover from a power supply failure, or return to the ground station on a radio loss, etc.). Additionally, it has 16Mbytes of flash-type memory, endowing the APM with the possibility of allocating on-board logging data.

- It includes a 3-axis gyroscope, a 3-axis accelerometer and a 3-axis magnetometer. It has also a barometric pressure sensor for altitude measurements.

- You can connect any other sensor in the 8 Input ports or any actuator in the 8 Output ports. Additionally, 8 extra ports can be configured as Inputs or Outputs.

- And finally, you can find some typical connectors to attach external devices:
o A Micro-USB, which allows you to attach directly the board to a computer. This allows you not only to get telemetry and interact with the autopilot, but also to upload the firmware you may need. However, and as it is a wired connection, it may not be possible in all scenarios.

o A I2C (“Inter-Integrated Circuits”) serial port, typically used to connect additional sensors like an external barometer.

o A Power Module port, to allow powering the board through an external power supply.

o A port for a GPS Module where, although 3DRobotics recommends the UBlox LEA-6H GPS [11] (precision of 2.5m), you can place any module you would like.

o And the last one is to plug in a custom serial radio device running at 57600bps, which can be used to communicate directly to a ground station (to receive telemetry, upload flight plans, etc.).

With these features, the APM 2.5 board is able to turn any RC vehicle into a fully autonomous unmanned vehicle, and it depends on which firmware you choose that it can fly fixed-wing aircrafts (“APM:Plane”), multi-copters and helicopters (“APM:Copter”), or driving ground rovers and boats (“APM:Rover”).

2.2.3. The Firmware

The firmware is the essence of any autopilot: it is the software uploaded to the board’s microprocessor and the responsible for all the actions it will perform. It is intended to handle a variety of procedures and functions during a mission, like autonomous stabilization, GPS navigation, telemetry broadcasting, mission managing, camera control, etc.

In the case of the APM Board, depending on which firmware has been uploaded it can fly planes, multi-copters, and rovers (the official names for these three branches of ArduPilot are, respectively, “APM:Plane” [12], “APM:Copter” [13], and “APM:Rover”). Although the differences between a plane, a multi-copter, and a rover raw in the singularities of their movements (and thus the navigation algorithms), the main structure of their corresponding firmware source code is almost the same in every case.

The main and most important branch of ArduPilot is the one for fixed-wing aircrafts (“APM:Plane”), thus having always the most stable and latest firmware version (actually, both APM:Copter and APM:Rover are evolutions of APM:Plane). This implies that, although both APM:Copter and APM:Rover work runs in parallel, they may have to change parts of their code after every major improvement is made to the APM:Plane firmware. Anyway, being the APM an Arduino-based board and ArduPilot an open-source project mean that the firmware can be modified by any user to suit
any vehicle we could ever imagine, uploading the resulting firmware to the board through the Arduino IDE software.

2.2.4. The implications of APM:Rover

As it was said before, the choice of the autopilot poses implications on the design of the other UGV subsystems. Because of using the APM:Rover we need to follow the defined ArduPilot architecture, affecting the vehicle, the power supply, the sensors, and the communications subsystems.

Firstly, regarding the vehicle we will need to define which rover platform are we going to use. Here, the only requirement APM:Rover states is that it must be a small electric car, which can run with electric motors and servo-motors like any typical RC vehicle.

In relation to the power supply, the AMP board is able to take the energy from a standard connected Micro-USB port or from any Electronic Speed Controller (ESC), to then power the rest of the connected components (servo-motors, sensors, GPS, and communications radio). As these ESC are the devices in charge of controlling the speed of any RC vehicle motor, by translating electric signals into different levels of current intensity, the ESC of the throttle motor should supply the APM board and its connected sensors and devices.

In the case of the sensors, unless we want to use better external ones we can use the on-board sensors of the APM board. The GPS, however, must be attached directly to the board.

And finally, the communications of ArduPilot are based in the MAVLink protocol, what stands for Micro Air Vehicle Communication Protocol and is an open-source lightweight protocol used for small UAS (like ArduPilot, pxIMU, SLUGS, FLEXIPilot, MatrixPilot, SenseSoar, SmartAP or AutoQuad6 [14]). Its messages, characteristics, etc., are introduced in the next section 2.4.

2.3. The Vehicle

Once we know what type of vehicle we can use to develop our test bed platform (an electric RC car), we selected a RC car model available at the ICARUS laboratory: the Turnigy 100BS (see next figure 2.3).
This car model is a 1/16 scale 4WD Electric powered car, and it was chosen among the large range of available cars in the market for its low price and high performances (high speed, 4 wheel-drive system, reasonable shocks, etc.). Additionally, you can screw and adjust almost any part of the vehicle, and there is full availability of spare parts to replace them in case of one is broken accidentally (or just in case you want to improve its mechanical components).

To drive this car, you can use any Radio Control System that has at least two RC channels (one for the throttle and one for the steering). However, the ArduPilot system requires some more in order to manage the different autopilot states and other related functionalities (for instance, although ArduRover requires just four RC channels, ArduPlane requires at least 5 of them). Being said that, we are going to use a Futaba T7C Radio-Control System (see next figure 2.4), a seven channel digital-proportional (programmable) RC system.

When we talk about the communications protocol, the ArduPilot community has opted for using the MAVLink Communications Protocol, which is actually used in a variety of UAS autopilots and environments.
2.4.1. MAVLink Features

A summary of the MAVLink features is presented next:

- It is developed around a set of predefined messages. Nevertheless, there is space for every user to define its own messages (message ID from 150 to 250) in order to fulfill any personal requirements.

- The packet length can go from 8 (minimum without payload) to 263 (255 payload) bytes, adding only 8 header bytes per packet.

- It can handle up to 255 aircrafts or vehicles (System ID 1-255).

- The checksum is the same as used in ITU X.25 (International Telecommunication Union) and SAE (Society of Automotive Engineers) AS-4 standards (CRC-16-CCITT). They are reported in the document SAE AS5669A, which refers to the JAUS Software Defined Protocols (a future architecture standard for UAS, whose adapted free version can be found in [15]).

- It is open-source with a GNU-LGPL license, so compiling an application with it is considered "using the library" and not a derived work. Therefore, MAVLink can be used with no limitations in any closed-source (private) application without the need of publishing the final source code.

To work with MAVLink, one uses a message definition set created in a XML file (the most common messages can be found in section 2.5.2). This XML is the one people can modify to include more different messages, making the protocol really flexible and versatile; ArduPilot, for instance, uses the full set of messages and adds some others, which are targeted primarily to control its added sensors and functionalities (like, for example, the sonars or a mounted camera).

2.4.2. MAVLink Packets

The structure of the MAVLink packets can be described with the following figure (2.5) and table (2.1), where you can see that the typical elements of any communications protocol are present (length, sequence number, sender and receiver ID, type of message, and verification bytes):

![MAVLink packet structure](image)
2.4.3. Developing with MAVLink

In order to work with the XML message definitions, the file is usually converted into an object-oriented library, being C/C++, C# and Python the most typical programming languages for its implementation (see figure 2.6). Hence, MAVLink supports the typical C programming language data types: char, unsigned/signed integers 8/16/32/64 bits long, float, double, and arrays of any of them (no “boolean”).

The generation of these libraries is partially supported by the open-source community of developers behind MAVLink. Therefore, although the generators can be hard to get them working fine, they can be found in the MAVLink online software repository [16].

Regarding the sequencing of the messages, MAVLink can be seen as quite simple and straightforward: for example there is no automatic resending of lost messages, if one is lost it simply waits for the next transmission. Hence, if one wants to have full acknowledgement that a message has been received, it has to be done externally to

![Figure 2.6 MAVLink XML message to C++ library example](image-url)

### Table 2.1 MAVLink packet detailed structure

<table>
<thead>
<tr>
<th>Byte Position</th>
<th>ID</th>
<th>Content</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>STX</td>
<td>Packet start sign</td>
<td>v1.0: 0xFE (v0.9: 0x55)</td>
<td>Indicates the start of a new packet.</td>
</tr>
<tr>
<td>1</td>
<td>LEN</td>
<td>Payload length</td>
<td>0 – 255</td>
<td>Indicates length of the following payload.</td>
</tr>
<tr>
<td>2</td>
<td>SEQ</td>
<td>Packet sequence</td>
<td>0 – 255</td>
<td>Each component counts up his send sequence. Allows to detect packet loss.</td>
</tr>
<tr>
<td>3</td>
<td>SYS</td>
<td>System ID</td>
<td>1 – 255</td>
<td>ID of the SENDING system. Allows differentiating different MAVs on the same network.</td>
</tr>
<tr>
<td>4</td>
<td>COM</td>
<td>Component ID</td>
<td>0 – 255</td>
<td>ID of the SENDING component. Allows differentiating different components of the same system, e.g. the IMU and the autopilot.</td>
</tr>
<tr>
<td>5</td>
<td>MSG</td>
<td>Message ID</td>
<td>0 – 255</td>
<td>ID of the message - the id defines what the payload “means” and how it should be correctly decoded.</td>
</tr>
<tr>
<td>6 to (n+6)</td>
<td>PAYLOAD</td>
<td>Data</td>
<td>(0 - 255) bytes</td>
<td>Data of the message, what depends on the message id.</td>
</tr>
<tr>
<td>(n+7) to (n+8)</td>
<td>CKA+CKB</td>
<td>Checksum</td>
<td>Error correction following standard ITU X.25/SAE AS-4 excluding packet start sign, so bytes 1...to(n+6).</td>
<td></td>
</tr>
</tbody>
</table>
the protocol (looking at the message sequence number or whatever). Nevertheless, it does define some procedures for the waypoint list managing and the GPS navigation (both can be found in [14]).

2.5. Testing the UGV

In order to test the capabilities and offered functionalities of both the RC car and the APM:Rover autopilot working together, we thought about placing all the basic elements required to get them work. They (apart from the previously explained electric car, the RC Transmitter and Receiver, the APM board and its sensors) are the ground station and the Communication Subsystem.

Regarding the Communication Subsystem, it will be composed by a couple of radio devices allowing serial port communications. Actually, we are going to use the devices that the ArduPilot project proposes: the 3DR Radios [17], which work at 433MHz and act as a transparent device (a complete analysis of these radios will be found in chapter 5).

In the case of the ground station (the responsible in any UAS environment of managing the UAV), the only requirement we will need to look for is that they must be based (and use) on the MAVLink Protocol. Although you can find different alternatives in the internet fulfilling this requirement, in the end we decided to use the most standard (and open-source) stations “APM Mission Planner” [18] from the ArduPilot community, and the “QGroundControl” [14] from the MAVLink Protocol developers.

2.5.1. The Ground Stations

As it has just been said, we decided to use the “APM Mission Planner” and the “QGroundControl” as our ground stations for testing the APM:Rover-based UGV. Apart from allowing us to compare the functionalities between them, the reason behind using two different stations is that they will give us two main separate results: while the first will allow us to deeply and easily configure all the parameters of the APM:Rover firmware (apart from managing waypoints and flight plans), the second will be used to test and analyse the standard MAVLink messages sent and received.

Regarding the “APM Mission Planner”, it is because of the importance of having a proper control of the aircraft (or any unmanned vehicle) while using the autopilot that the ArduPilot community has been working in designing their own ground station. This program allows you to operate in real-time with your aircraft, your multi-copter or your rover, and to configure everything in the APM board. The following are some of the major functionalities this software offers:

- It allows visualizing all the telemetry of the vehicle in real-time, showing at every moment the followed path and the flight plan (or “ground-desired path”).

- It includes a series of buttons to perform every action the APM can perform, like Return-to-Launch, change speed, or do a “loitering” manoeuvre.
You can edit, create, and upload, complete flight plans with a proper visual interface (see example in figure 2.7).

It has a tab to configure every parameter of the APM board, from the flight states to the RC values.

And finally, it gives the user an easy way to upload the firmware to the APM Board and for any of the ArduPilot supported vehicles (illustrated in the next figure 2.7).

In relation to the QGroundControl, it has been developed by the creators of the MAVLink Protocol to support all the MAVLink-based UAV autopilots. To do so, they have implemented the management of all the MAVLink Common Message Set, allowing us to perform any generic autopilot action: upload flight plans, change parameters of the autopilot, receive telemetry, etc. Additionally, they provide a series of logging functionalities to show the final user the messages sent over the communications channel.

This station will allow us to see at every moment how to perform the different procedures, besides to better analyse the behaviour of the autopilot. A screenshot of the flight plan tab and the logging functionalities is presented in the next figure (2.8).
2.5.2. MAVLink in APM:Rover: Analysis

In order to know which are the most important messages implemented in the APM:Rover firmware, and to see how it really uses the MAVLink Protocol, we have performed the analysis of the messages exchanged between the ground station and the APM:Rover in different situations. To do so, we have used the “QGroundControl” station, connecting the autopilot directly to our computer (by an USB cable).

Because the complete set of MAVLink messages, along with its composition and different values, can be found in [19], in this section we are going to present only a brief summary of this analysis.

- Firstly, we present the messages used in a steady state, where only telemetry and the state of the system are being sent. Although the telemetry information could be modified by asking the autopilot not to send a group of variables, we wanted to record the typical scenario where the APM Mission Planner has initialized the autopilot. Here (figure 2.9) you can see a screenshot of the messages logged in this scenario:

```
<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYST-STATUS</td>
<td>(2.0 Hz, #1)</td>
<td></td>
</tr>
<tr>
<td>GPS_RAW_INT</td>
<td>(1.0 Hz, #24)</td>
<td></td>
</tr>
<tr>
<td>RAW IMU</td>
<td>(1.0 Hz, #27)</td>
<td></td>
</tr>
<tr>
<td>SCALED_PRESSURE</td>
<td>(1.0 Hz, #29)</td>
<td></td>
</tr>
<tr>
<td>ATTITUDE</td>
<td>(8.5 Hz, #30)</td>
<td></td>
</tr>
<tr>
<td>GLOBAL_POSITION_INT</td>
<td>(3.0 Hz, #33)</td>
<td></td>
</tr>
<tr>
<td>RC_CHANNELS_SCALED</td>
<td>(25.4 Hz, #34)</td>
<td></td>
</tr>
<tr>
<td>RC_CHANNELS_RAW</td>
<td>(2.0 Hz, #35)</td>
<td></td>
</tr>
<tr>
<td>SERVO_OUTPUT_RAW</td>
<td>(2.0 Hz, #36)</td>
<td></td>
</tr>
<tr>
<td>MISSION_CURRENT</td>
<td>(1.9 Hz, #42)</td>
<td></td>
</tr>
<tr>
<td>VFR_HUD</td>
<td>(3.5 Hz, #74)</td>
<td></td>
</tr>
<tr>
<td>HEARTBEAT</td>
<td>(1.0 Hz, #0)</td>
<td></td>
</tr>
</tbody>
</table>
```

**Figure 2.9** MAVLink messages in a typical APM:Rover steady state

- Regarding the flight plan modification, whether uploading new flight plans or setting the next waypoint to go (changing index count), the QGroundControl software and APM:Rover talk to each other by means of the following messages: “MISSION_ITEM”, “MISSION_REQUEST”, “MISSION_SET_CURRENT”, “MISSION_REQUEST_LIST”, “MISSION_COUNT”, “MISSION_ACK”, and “MISSION_CLEAR_ALL”. The procedures to perform this flight plan management can be found in [14].

- The QGroundControl offers also the possibility of getting and setting the parameters of the APM:Rover. These are handled by the “PARAM” messages: “PARAM_REQUEST_READ”, “PARAM_REQUEST_LIST”, “PARAM_VALUE”, and “PARAM_SET”.

- On special events (like when changing the flight modes, uploading flight plans, or rebooting the board), the autopilot sends text-type messages to the ground station. These are received with the “STATUSTEXT” message.
Finally, MAVLink stands for the possibility of sending specific commands to an autopilot (from navigation commands like returning-to-home, to adjusting on-board system parameters, through changing flight modes and states). The complete list of available commands can be found in the “MAV_CMD” enumeration, but the message used is the “COMMAND_LONG”.

2.5.3. Assembling the APM:Rover Platform

Once we defined the components we are going to use, the assembly of the RC Car and the Autopilot board was made. To do so, we followed the steps described in the APM:Rover tutorial [6], which can be summarised with the next picture (2.10):

![Figure 2.10 APM 2.5 Board connections for APM:Rover implementation](image)

Finally, as one can see in the following photographs (figure 2.11), we managed to place all the components together inside the original car plastic cover, what provided the car with much more toughness and strength against collisions, while protecting the electronic components from dust and stones.

![Figure 2.11 RC Car, APM 2.5, GPS and 3DR Telemetry Radio assembled](image)
2.5.4. First tests results

Despite the problems we had with a broken shaft in the firsts tests (actually the car is made of plastic and can resist a moderate amount of stress), we performed some practical tests on ground to test not only the performance and capabilities of the APM:Rover firmware, but also how it will behave in different navigation situations (like when changing the flight plan in real-time, for instance).

The results we got from the tests were various:

- We confirmed that the messages used by APM:Rover follow the MAVLink standard, allowing our USAL Virtual Autopilot System (VAS) to talk directly to the autopilot (replacing the ground station).

- The movement of the rover with the automatic navigation mode is fine, although it needs quite amount of testing to tune the diverse navigation parameters (PIDs, navigation periods, etc.).

- The electromagnetic interference produced by the electric motor affects the on-board magnetometer, making the rover sometimes to weave in straight lines. This behaviour is slightly improved when enabling APM:Rover auto-calibration, which correlates its values with the GPS data to adjust its offsets.

- Uploading a complete flight plan can be done at any moment (“on-the-fly”) and in any “flight” mode, as well as the modification of any configuration parameter.

- The change of the MISSION_CURRENT value (the waypoint the car is navigating to) can be done correctly, making the rover change its trajectory to go to the new waypoint selected.

- The behaviour of the rover when changing the flight plan while it is going to a waypoint can be explained with the following figure (2.12) and example:
  
  o Take the scenario where the rover is driving to the waypoint number four. At this moment, APM:Rover has memorised the fourth waypoint in its software and the MISSION_CURRENT value is set to number 4.

  o Then we upload the new flight plan, what does not affect the path of the rover instantly. In fact, it goes to the previously memorised fourth waypoint, without taking into account the new flight plan.

  o When it finally reaches the waypoint number 4, then it looks into the new stored flight plan and takes the fifth waypoint it finds there, driving accordingly to get it. The MISSION_CURRENT value is set to number 5, and APM:Rover continues working fine.
In automatic mode, if we clear the whole flight plan the rover continues to the stored waypoint as explained before. Then, when it reaches the waypoint, it stops and changes its state to “Hold”. Similarly, when it completes the flight plan it stops and changes to “Hold” mode, meaning it does not restart the flight plan.

Regarding the waypoint managing, the ground control stations only have implemented the upload of the complete flight plan (no MISSION_PARTIAL_LIST messages), so we could not test at this moment the change of only one or a series of waypoints.

And finally, regarding the “flight” modes of the rover, it has implemented the following ones: “Manual” to be driven by the RC Radio, “Auto” to perform automatic navigation, “Hold” when it stops after a flight plan or as a fail-safe reaction (lost link, or whatever), and “Learning” mode (which is a functionality of the firmware to create flight plans while driving the rover, not necessary for our objectives).

2.5.5. First tests conclusions

From these first tests, we extracted some conclusions that were the guidelines for our future work:
• Breaking the shaft in the first tests made us take into account the fragility of the structure in posterior tests. Therefore, we decided to avoid difficult terrain like sand unless we have stronger parts.

• The configuration of navigation parameters is critical and unique for every RC car, so we should find the good ones and store them before running any further tests. Additionally, we must keep in mind when developing our USAL service that it could be required to change them depending on the scenario (to do smooth curves like a plane, or to make sharp turns similarly as a helicopter does).

• Although the MISSION_PARTIAL_LIST has not been implemented in any of the two tested Ground Stations, the APM:Rover firmware already does (it allows only to modify elements in the flight plan, not to add new ones). Consequently, and although it will require some testing, we could use it in our software if needed.

• The capability of uploading the flight plan, or the parameters, at any moment without affecting the actual path of the rover gives us freedom to decide when to perform these actions in the future software.

• Although uploading a new flight plan does not make the rover change its memorised waypoint to go to, changing the MISSION_CURRENT does. This means that we can force the rover to go to a desired waypoint without sending a GOTO command, making it better for USAL flight plan and mission managing.

• And finally, combining the recently mentioned capabilities, it will be possible to perform an infinite queue of waypoints for the USAL architecture. By having a set of waypoints (five, for instance) periodically refreshed, in combination with a smart use of the MISSION_CURRENT value, should allow us to do so. However, we need to implement a “Waypoint Reached” event to make us be aware that a flight plan update needs to be done.

Besides all the technical and operational characteristics that were tested, we also faced another characteristic to take into account: the firmware upgrading. To illustrate this fact we can explain that in the couple of weeks the tests lasted, we actually noticed some significant evolution in the available firmware in the internet (like modified navigation algorithms and different software bug correction). This constant upgrading is not a problem itself, but has some implications in our future work: we cannot focus in modifying the firmware to meet our requirements, because it will be much different by the moment we end our modifications.

Concluding, after testing the autopilot capabilities we proved that the APM:Rover autopilot is a good choice suitable for our test bed solution. Additionally, it can be the first step for eventually using the other firmware versions (APM:Plane or APM:Copter) without major changes, allowing ICARUS Group to test USAL architecture in many different vehicles.
Chapter 3

USAL INTEGRATION

As it name states, this chapter will cover all the information regarding the integration of APM:Rover into the ICARUS USAL architecture, focusing in the implementation made in the Virtual Autopilot System (VAS) service.

3.1. The Virtual Autopilot System (VAS)

As it was explained in the first chapter, the VAS is the name of the USAL service responsible of handling all the operations and communications with the autopilot (it basically acts as an interface between USAL and the AP). Essentially, and summarising, the functionalities it provides can be divided into four separated areas:

- **Flight State Management**: implements and manages the different flight states from a mission point-of-view (for Start-up, Taxi procedures, Take-off, Navigation, Landing, Safe states, Reactions, and Manual operation).

- **Navigation Information**: working closely with the Flight Plan Manager (FPMa) service, it is in charge of controlling inbound and outbound navigation commands (waypoints, angles, etc.).

- **Flight Telemetry**: information like angles, velocity, position, etc., provided to other services can be grouped in this area.

- **Status/Alarm Information**: being raised when any part of the device fails, alarms can be used to alert the ground control station and other services.

Regarding its implementation (and according to figure 3.1), the VAS service uses a two-layer level design: one interacting with USAL (called the Virtual Autopilot Layer or VAL), and another interacting with the autopilot (called the Autopilot Layer or APL). The first layer (VAL) is fixed and never changes, while the second is the one needed to be modified to meet the peculiarities of any specific Autopilot.
In our particular case, and because of the existing possibility of adapting the software to different MAVLink vehicles (planes and multi-copters), we have detached the communication with the autopilot from the implementation of the USAL functionalities. The result is the division of the former Autopilot Layer (APL) into two different libraries: the Communication Manager (a MAVLink Manager) and the Autopilot Manager (in this case the APM:Rover Manager). This can be seen in the following figure (3.2).

3.2. The Communication Manager

Focusing on integrating the APM:Rover into the USAL architecture, what we need to develop is a software library able to talk directly to the APM board. It will offer a set of functionalities like telemetry, APM parameter managing, etc., while handling the
communication with the autopilot using the MAVLink protocol. This will give us full control of the communications, allowing us not only to work with the most updated APM:Rover firmware version, but also with any other MAVLink-based autopilot.

Regarding the programming language, and because all USAL is coded in C#, we chose it to provide stability and to avoid further portability problems. Although the APM Mission Planner is programmed in this language (QGroundControl is developed in C++), we decided not to try to use its source code to ensure we get a clean and standard MAVLink library (actually APM Mission Planner code is quite chaotic).

### 3.2.1. The starting point: a standard MAVLink library

It was said in the previous chapter that there exists MAVLink library generators for C, C# and Python programming languages (for C# the only working generator is allocated in the MAVLink “Github” repository [16]). What they basically do is transforming the XML message definitions into a DLL file, allowing us to work with MAVLink message structures and to encode/decode them into arrays of bytes (which can be sent through the serial port, for instance). The resulting MAVLink DLL schematic is the next figure (3.3).

![Figure 3.3 Mavlink.dll file UML representation](image)

To be more specific, this DLL contains:

- All the values contained in the MAVLink Type Enumerations, plus the message structures (as “MAVLinkMessage” objects), contained in the XML file.

- The definition of a “MessagePacket” object (used to build the messages into a MAVLink packet) and which will be interpreted by the function “Send” of the DLL library. This “Send” function will give us the array of bytes needed to be sent over the serial or our defined communication port.
• A procedure to decode an array of received bytes called “ParseBytes”, which is the responsible of raising the corresponding (and explained below) events.

• Two events to be redirected to our message handling functions: one for every packet received, and one for packets received with bad CRC correction code (which usually alerts you that a message the autopilot uses is not defined in the MAVLink-XML file, and so does not in the DLL).

Concluding, using this “mavlink.dll” file provides us the structures and object types for handling the MAVLink protocol, without being linked to any particular autopilot. This implies that the user is the responsible for building all the packets to send, filling all the required fields composing them, and to decode the received ones to point them to his desired functions.

3.2.2. MAVLink Communication Manager

In order to implement the USAL requirements, we planned to program this library independently, in such a way that could be used in the future in other projects or by other developers. In this sense, it will not be oriented only for APM:Rover, it will be able to work with any other MAVLink-based autopilot (like APM:Plane, targeting its eventual integration into USAL)

With this idea, the designed library was coded with the following functionalities (the UML representation is figure 3.4):

• Regarding the connection of the software with the Autopilot:
  
  o Being serial ports (sequential bit-by-bit physical interfaces) the main method for managing autopilots, we kept this preference in our implementation. As a result, our library provides a way to easily handle this connection through a serial port (wired or wireless), with no further action from the programmer than setting the name of the port and the desired baud rate (“Connect” and “Disconnect” methods).

• With respect to the telemetry information:
  
  o There is a method (“RequestTelemetry”) to activate/deactivate the telemetry messages the Autopilot produces, which allow us to free the processing duties of the AP (if telemetry is not needed). In addition, we can set the desired frequency for these messages, getting exactly what we plan to obtain.

  o The library generates “events” when new telemetry values are received from the autopilot. These events follow the USAL architecture, being speeds, angles and position the given information.
• Concerning the configuration of the autopilot:
  
  o It takes care of the parameters of the autopilot, being in charge of storing them and sending/requesting them when needed (everything in a comfortable way for the final user with the “GetParametersList” and “SetParameter” functions).

• About the mission and waypoint managing:
  
  o This library contains a copy of the on-board autopilot flight plan (to not overload the MAVLink communication link) and the procedures to
properly upload it. With this feature, the final user will be able to request the flight plan as many times as it wants, but the autopilot will only be noticed when the flight plan has been changed (methods “GetWaypointsList”, “UploadFlightPlan”, and “ClearFlightPlan”).

- The “PartialList” procedure is also implemented in our library, allowing us to modify one or more waypoints in the list (not to add additional waypoints). Additionally, we programmed another function to get a Partial List (one or more items) of the on-board flight plan (“UploadFlightPlan” and “GetWaypointsPartialList”).

- As it was said in the previous chapter, we needed to be aware when a waypoint has been reached (although there is a “Waypoint Reached” message in the MAVLink protocol, the APM firmware does not send it). The autopilot only sends “text messages” when changing the destination waypoints, which we managed to interpret to raise a proper event (“Waypoint_Reached”).

- **With regard to the management of the autopilot state:**
  - Apart from a method to change the Autopilot Mode (Manual, Hold, Auto, etc.), there are two public variables (“AutopilotMode” and “AutopilotState”) to allow accessing at any moment both the AP mode and the Autopilot State (Booting, Calibrating, Critical, Active, etc.). Additionally, and in relation to the Autopilot Mode, we have implemented an “event” to alert the software that a change has taken place (by the RC radio or the APM firmware, for instance).

  - There are also other “events” or “alerts” related to specific MAVLink messages (which could be useful for a MAVLink-based autopilot). They are the “HeartBeatMessageReceived” (used to ensure the autopilot is responding), and the “StatusTextMessageReceived” (sent by the autopilot when it needs to give us some extra information not reflected by other messages, like when starting up or whatever).

- **And finally, there is a method which will take profit of the MAVLink RC_CHANNELS_OVERRIDE message (“OverrideRCChannels”). This message sets the values of the output servos connected to the Autopilot, and can be used when having manual control of the vehicle through the computer or with a joystick.**

As a result, we have developed a C# library which provides an abstraction layer to future programmers (which will not need to know all the MAVLink messages and procedures) and which will be the base for our Virtual Autopilot MAREA Service.
3.3. The Autopilot Manager

Although the architecture of a ground and aerial vehicle is almost the same, the integration into an UAS-oriented platform has some peculiarities: for example, there are navigation commands that cannot be executed on ground, states like landing or take-off that cannot be performed, or even payloads that cannot be carried on. However, in any case the rover ends to be a simplified plane (with less actuators, systems, etc.) making the APM:Rover integration the first step to produce an ArduPilot-based UAV.

3.3.1. Specificities of a Rover APL

In order to properly develop our desired Autopilot Layer, we must recall following two basic guidelines for USAL implementation: we must keep the “IAutopilot” interface (the methods we need to implement and which are presented in figure 3.5) to communicate between the APL and the VAL, besides using the VAS states types and management procedures defined by USAL.

![IAutopilot Interface](image)

Figure 3.5 “IAutopilot” Interface

Although there are 14 procedures available to code in the “IAutopilot” interface, not all methods will be needed for a proper use of a rover (for instance, methods like “SendALT” to send new altitude, or “SendRWY” to send the runway parameters are pointless for terrestrial vehicles). Consequently, the next are the essential which have been coded:
• “ListenAp”: Method to make the autopilot layer to start working with the real autopilot.

• “ChangeState”: This processes the required operations to change the AP state, not only regarding the autopilot but also its USAL implications (transitions, new states, etc.).

• “ClearAllWp”: This is received before uploading a new flight plan to the VAS queue of waypoints.

• “ManualNavigation”: Used to send joystick parameters appropriately converted to the AP.

In a similar way than before, there are several autopilot states defined by USAL that will not be used in our implementation. To summarise, we will present the adopted solution (figure 3.6) and the states used to fulfil the VAS requirements:

• “CONFIGURE CHECK”: This state is activated when connecting and initialising the ArduPilot board (it corresponds to its “CALIBRATING” state). Here, the parameters are loaded and the autopilot is configured, including the calibration of its own sensors (barometer, accelerometer and magnetometer).

• “PARKING”: It corresponds to the “HOLD” mode of APM:Rover, and it takes place when the AP triggers a “failsafe” procedure (for example when no RC radio is detected, or when the list of waypoints has been navigated and there are no more actions to perform). As in the car the “HOLD” mode produces the vehicle to be stopped, the PARKING state appeared to be the best adaptation possible.

• “DIRECTED”: This VAS state is typically used when the UAV is under stabilized flight, being possible to send parameters like desired altitude, speed and bearing. As this state is not implemented in the APM:Rover, in our case we have used it to name the case when no action is transmitted to the rover from the ground station, being the RC radio the one controlling the vehicle. Summarising, it corresponds to the typical “MANUAL” ArduPilot mode.

• “MANUAL NAVIGATION”: It corresponds to the use of a Joystick to control the movement of the vehicle (in a UAV the movement is defined by a 3-axis joystick plus throttle and other surface actuators, while in the rover there are only two axis used corresponding to steering and throttle). The mode in the ArduPilot board is “MANUAL” as in the previous state, but here the ground station is sending to the car the desired values for the servos/actuators (overriding the RC transmitter values).

• “WAYPOINT_NAVIGATION”: Being the foremost objective of an autopilot, the rover will be in this state when performing full autonomous navigation (“AUTO” in ArduPilot). Another USAL service (the FPMa) is responsible of sending to the VAS the flight plan list, who takes care of constantly feeding the APM:Rover autopilot.
• “MANUAL”: Finally, this VAS state is intended to be as a USAL “failsafe” state, meaning that full control of the vehicle is given to the primary device (in this case, the RC radio transmitter) without possibility of going back to a navigation state.

3.3.2. Automatic Navigation in APM:Rover

As it has been explained previously in chapter 2, the APM:Rover navigation relies on the use of an uploaded flight plan, which it tracks and drives as a list of waypoints. In USAL, as there is the possibility of changing the flight plan in real-time (adapting it to the requirements of the mission), we cannot upload the entire flight plan at start and wait to be completed. Besides, the whole flight plan can have almost infinite waypoints, which will not fit inside the autopilot memory. Consequently, we need to implement a continuous navigation loop, in such a way that we can constantly feed the autopilot with a set of desired waypoints.

To this purpose, we have implemented a continuous loop managing a circular queue of waypoints. This queue takes profit of the previously added functionalities developed in our MAVLink Manager library: they are the “Waypoint_Reached” event (which will trigger the action), the “Upload_Partial_List” procedure (which allows us to change one single element in the flight plan), and the “Set_Mission_Current” action (which makes the autopilot restart the queue). By combining them, we will manage a queue of 5 waypoints which will be constantly updated in a reliable and efficient way.

Apart from at start of navigation, the abovementioned queue is updated every time a waypoint is reached. In this sense, depending on the ID of the passed waypoint the
procedure to execute will be different, whose basic characteristics are the ones summarised here (the procedure is illustrated in figure 3.7):

- In order to perform a continuous loop, we will need to tell the autopilot to restart the flight plan at some time. In this sense, because it is not possible to end it (remember the firmware changes its state to “Hold” after finishing it), this restarting has to be done while the autopilot is still navigating to a waypoint. Hence this has been performed while we are driving to the last point, changing the “Mission_Current” on-board value to “1”.

- Because we want not to lose waypoints while performing the abovementioned change, the last and the first ones need to be exactly the same. This will allow the APM:Rover to make the transition between the two without affecting the navigation path.

- Regarding the upload of waypoints, every time a waypoint has been reached a new one will be uploaded (in our case, because if we have reached WP_2 we are then driving to WP_3, we can upload a new WP_2 without interfering with the navigation).

- And finally, the queue has been built using 4 different waypoints (five is the total size of the queue), but this is an arbitrary choice.

In the end, by using this navigation loop we will make the autopilot never ending its queue of waypoints, and therefore we will have the possibility of driving almost infinite points (provided USAL flight plan contains them).
3.3.3. The APM:Rover Autopilot Manager

To develop this APM:Rover Autopilot Layer, we have structured it under the following series of objects (a visualisation can be found in the next figure 3.7):

- “ArduRover”: It implements the “IAutopilot” interface and controls the behaviour of the APL, dispatching procedures to other classes and implementing the change of states (triggered by external USAL services).

- “AP_Navigation”: As its name states it copes with all the navigation-related procedures. It is in charge of requesting and publishing AP mode changes, uploading flight plans, sending joystick values and managing the automatic navigation loop explained above.

- “AP_Reader”: It takes control, and performs the required actions, of all the events triggered by the Autopilot Manager. They are essentially telemetry values, status text messages, and AP mode changes (produced by the RC radio transmitter, or by the APM:Rover firmware itself).

- “AP_Parameters”: Although at this moment this object is only used to upload a list of desired parameters to the Autopilot (at start-up), it may be used in the future to handle the real-time modification of parameters (for instance, by a dedicated configuration service).

3.4. Testing the APM:Rover VAS

After concluding our work in the VAS Service, we want to test its capabilities and its correct (or not) performances. To do so, and because the APM:Rover does not have a functionality to be simulated (in a Hardware-in-the-loop environment), we need to test it in a real driving platform. The first approach of assembling the whole USAL architecture (with a computer on the rover running the VAS and FPMa, another
computer as a ground station, and the connection between the two through an Ethernet Wi-Fi connection), posed some initial problems:

- Firstly, the inclusion of the complete architecture might arise other MAREA internal errors (or from other service), which may not allow us to detect our own implementation errors.

- Secondly, to date MAREA and the USAL services have only been working in Windows operating systems, needing a computer with a x86 processor architecture like Intel or AMD. This means that we need to place, for instance, a nano-ITX board like the EPIA N-700 [20] (a 12x12cm board available in the ICARUS Group), which has a relatively high power consumption (around 13-15W), and which needs an external voltage regulator to provide 12V stable power. These two points make it not the most suitable solution for such a small car like ours (at least at this point of development).

- Finally, if a modification in the already assembled hardware can be avoided, it will save us valuable time for upcoming designs.

For these reasons we decided to go for an intermediate scheme, which takes profit of the radio port in the APM board. Actually, what we have done is to use the previous architecture of the ArduPilot environment: a ground station (running all “Air” and “Ground” services) connected to the APM board through the telemetry radios (using them as a “transparent” serial device as they were connected by a cable). This architecture is summarised in the next figure.

![Figure 3.9 Intermediate architecture to test APM:Rover APL](image)

As it was its main objective, and although we had some initial problems derived from the radios, this solution worked fine and allowed us to extensively debug the first versions of the VAS service. Besides, it made us glimpse one problem that will be presented in detail in next chapters: the loss of bytes when using the 3DR radios.
Chapter 4

ASSEMBLY OF THE AIR SEGMENT

After developing the Autopilot Layer (APL), and therefore the required software is ready to be used in a real platform, the next step is to follow the USAL architecture: to place the two main “Air Segment” services (the Virtual Autopilot Service or VAS, and the Flight Plan Manager or FPMa) on-board the car. To do so, we need to analyse the elements we need and the implications they bring to the whole system, assuming that there will never be an obvious choice for none of the components.

4.1. Architecture of the Rover

In the next figure (4.1) all the necessary elements for our USAL Rover are presented, along with the required interconnections between them. There, the boxes painted in blue are the systems that we already have, and the orange the ones we need to select and to integrate:

![Figure 4.1 Basic elements of our Rover “Air Segment”](image)

To be more specific, the blue elements (the ones we already have) are:

- The autopilot and sensors are the APM board and its integrated sensors, the APM:Rover firmware, and the GPS external module.

- The RC communications subsystem connected to the autopilot is the RC radio receiver (the transmitter is in the Ground Station), used to take direct control of the car and the autopilot like, for instance, in contingency situations.
The motors and servos are the ones of the RC car (in this case the steering servo-motor and the throttle motor), which are connected to the autopilot.

The elements we need to select and integrate (the ones painted in orange) are the minimum and essential to develop our USAL platform, being common in any unmanned vehicle. The on-board computer and the power subsystem are explained in the next sections, while the communications system will be detailed in the following chapter (number 5).

4.2. The On-board computer

Apart from the autopilot, the main element in any unmanned system is its on-board computer, which is typically in charge of managing the payload and the communications with the Ground Station. In our case, as we need to execute the USAL services in the rover, we definitely need to place a computer in it. However, its selection is not an easy choice, being necessary to analyse some parameters and trade-offs like size, energy consumption and processing power.

4.2.1. CISC or RISC

First of all, we must explain that there are two big families of CPU architectures: CISC and RISC processors. CISC computers (Complex Instruction Set Computing) use few instructions but more complexes, resulting in relatively slow performance per instruction but providing more code flexibility. On the other hand, RISC chips (Reduced Instruction Set Computer) have simpler instructions with faster processing time, but require more operations to accomplish a task. A small summary table is presented below (4.1), but more information can be found in [21].

<table>
<thead>
<tr>
<th></th>
<th>CISC</th>
<th>RISC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Emphasis on hardware</td>
<td>Emphasis on software</td>
<td></td>
</tr>
<tr>
<td>Includes multi-clock complex instructions</td>
<td>Single-clock, reduced instruction only</td>
<td></td>
</tr>
<tr>
<td>Memory-to-memory: &quot;LOAD&quot; and &quot;STORE&quot; incorporated in instructions</td>
<td>Register to register: &quot;LOAD&quot; and &quot;STORE&quot; are independent instructions</td>
<td></td>
</tr>
<tr>
<td>Small code sizes, high cycles per second</td>
<td>Low cycles per second, large code sizes</td>
<td></td>
</tr>
<tr>
<td>Transistors used for storing complex instructions</td>
<td>Spends more transistors on memory registers</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1 Comparison of CISC and RISC architectures [21]

Broadly speaking, we can associate CISC processors with personal computers and its two major manufacturers: Intel and AMD. Basically, although they require more
energy to perform simple operations, they provide more ease in software
development and thus can run more complex programs with less coding effort.

In a similar way, and although the creator of this architecture was ARM (Advanced
RISC Machine) Holdings, we can link RISC-based computers with mobile devices
and with names like Qualcomm, Texas Instruments or Samsung (who are actually its
main producers). These devices take profit of its lower energy consumption (see
figure 4.2) thanks to its simpler operations, while they are nowadays less capable in
terms of global computation power (compared to CISC).

![Figure 4.2 Energy consumption comparison (in Watts) of Intel Atom N450 and ARM Cortex
A8 (processors with similar computing power) [22]](image)

The answer to why this topic affects our project has been introduced in the previous
chapter: USAL has been programmed in C# programming language, aimed to be run
under Windows operating systems (OS) and Intel x86 processors (thus using CISC
architecture). Because of that, porting the software to a RISC mobile platform will
present some difficulties whose worthiness we need to analyse.

Although taking a small Intel computer might have been the easiest and fastest
solution, RISC boards presented some advantages to our platform:

- CISC processors require more power to operate. In the car, the battery we will
  have will be limited, so any saving in this direction would be valuable.

- Less consumption power means less heat dissipation requirements. This
  allows RISC processors to not need an additional fan to dissipate energy,
  reducing the complexity of the boards.
• ARM boards are smaller. The field of RISC-based board sizes is evolving really fast (thanks to the evolution of the smartphone market) and this will be an advantage when mounting the board on the vehicle.

• Air Segment software is not high-demanding in terms of processing power. This means that VAS and FPM can run in background and without visual forms, whose consequence is that we do not need to go for some of the advantages of Windows OS.

• And finally, ARM boards are getting cheaper day after day. Although once the board is bought this may not have much special interest, it would be interesting for next projects within USAL framework.

Hence, taking all these points into account we saw that although it was not the faster solution, it would be a good investment to work in porting VAS and FPMa services to an ARM-based board. In this sense, it will not only serve us to integrate ArduRover into USAL, but also any other vehicle (aerial or terrestrial) that ICARUS Group may want to develop in further projects.

4.2.2. Choosing a mobile on-board computer

In the existing market (mainly in internet stores selling electronic devices, like [23] or [24]) one can find multiple options of small-sized all-in-one boards. This kind of boards are characterised for having all the necessary components a computer needs to work integrated in just one board. They are typically built around a System on a Chip (or SoC), housing a Central Processing Unit, a Graphical Processing Unit and some RAM memory.

Additionally to the SoC, it depends on the manufacturer the extra chips and connectors he may want to place to enhance its capabilities. Typically, all this boards have USB or Ethernet controllers, voltage protections, coders for audio, video outputs and a serial interface, what makes them compatible with most typical mid-user standard applications (media players, development boards, etc).

Regarding our purposes, we need a board which has the following requirements:

• Two or more USB ports.

• A serial or an Ethernet port (or even a standard Wi-Fi module) to take control of the board without a screen.

• The highest processing power possible in a relatively small-size packed board.

By the day we started this project (in January 2013) there were some different alternatives available in the market, being the following the most remarkable: the Raspberry Pi, the BeagleBoard-xM, the BeagleBone, the PandaBoard, the ODROID (in its –X or –U versions) and the CubieBoard. In the next table (4.2) you will find their most characteristic specifications.
Regarding the software capabilities, all these boards can theoretically run any free OS (mostly known as distributions) which supports ARM processing units (some examples are Ubuntu, Android, or Angström, all based in Linux). However, every core has its peculiarities and not all distributions are suitable for every board, meaning that we must take into account what OS we want to execute. As we will explain later, because we need to use a program called “Mono” to execute USAL, we need a board capable of running Ubuntu or Debian distributions.

Concluding, because of having good computation capabilities, a lot of connectivity possibilities through different interfaces, and a wide community of developers supporting it, the final chosen option was the PandaBoard. As presented in the table, this board has two CPU cores running at 1GHz with 1GB of RAM, 2 USB ports (plus two extra through expansion connectors), an Ethernet port, HDMI video out, wireless connectivity and a DB-9 serial port interface.

Basically, the PandaBoard was targeted to software and hardware developers wanting to test their devices in ARM-based chips, and has wide support from the internet community of developers [25]. For instance, and as they share the same SoC, this board was used as a development platform for creating Samsung Galaxy SII smartphone applications, what let us compare the capabilities of such hardware. In addition to its hardware specifications, the PandaBoard has support for Android, Angström and Ubuntu (not for Debian), which will allow us to run our USAL services.

![Table 4.2 Comparison of the RISC-based available boards](image-url)
4.2.3. Setting up the PandaBoard

Prior to develop software for the PandaBoard there are some installation procedures required. The first step is to choose the distribution you want to work with, and here the PandaBoard community [25] provides you with some pre-installed Ubuntu images to be copied directly into your SD card (the PandaBoard uses an SD card as a storage drive). Although we had tested other distributions available along the internet, the best results were obtained with the “official” ones listed in the PandaBoard webpage (see figure 4.3).

![Figure 4.3 Preinstalled Ubuntu images available for the Panda Board](image_url)

As the numbers in the Ubuntu versions relate to its release date, we can see that they are not the latest Ubuntu versions (the last is from April 2012). However, they are expected to be largely tested and with proven good performance. “Desktop” versions come with a complete set of GUI (Graphical User Interface) applications installed (desktop interface with forms, buttons, web browsers, etc), while “Server” versions install only the required files needed for the OS plus some optional server plugins (Apache, SSH, etc.).

For our project where no visual forms are required, we went for a “Server” version because they are “lighter” (or faster) distributions. Instead of going for the most recent version, we went for the “Ubuntu 11.10 Oneiric Ocelot Server”, mainly because of the Hard Float/Soft Float issue we are about to explain in the next section. A complete description of the procedure to format the SD card and to install Ubuntu in it can be found in [27].

After formatting the SD card there was the need of installing and configuring the distribution. This could be done by connecting the power plug to the board at one side, and a serial cable connected to the PandaBoard’s DB-9 serial port and to an external computer at the other side. We could then access the PandaBoard via this external PC, by opening a serial port terminal (with software like “Putty” [28]) with an 115200bps port baud rate.

At this point, Ubuntu is already installed and we can access the board through the serial port. Additionally, if we want to access the board via SSH protocol [29] (which
will allow us to send files and execute software in the board easily), we can do it by connecting it to an Ethernet LAN network or by configuring the on-board Wireless network adapter ("sudo apt-get install wireless-tools" command).

It is important to remind that having updated Linux distributions (and programs or packages) is really important to prevent execution errors, so that was the first thing we did by means of the next commands: "sudo apt-get update" and "sudo apt-get upgrade". With the OS finally ready, it was time for the next step: install the possibility of using C# developed software by using "Mono".

4.3. The Software: Porting .NET C# to Linux

4.3.1. The Mono Project

MAREA – and thus all USAL architecture – is coded using C# programming language, an object-oriented programming language which allows developers to build applications running on the .NET Framework [30]. The first version of its specification was published in 2002 (now there is the fifth version C# 5.0) and was developed, as well as the framework, by Microsoft [31]. It became a standardised language in 2003, when it was recognised by the ISO (International Organisation for Standardization [32]), being the complete specifications found in [33].

The strategy of Microsoft was to make C# and the .NET Framework only available for Microsoft Windows OS, to promote its use over the competence by providing easier and faster tools to code software. The principal C# compiler (Microsoft Visual C#) is not a free software, meaning that if you want to develop and sell software with this tool you need to purchase a full license of it (the typical IDE software, or Integrated Development Environment, is Microsoft Visual Studio). However, nowadays these two limitations can be overcome by using new free C# compilers, like the “Mono” project [34] or the “DotGNU” project [35].

“Mono” and “DotGNU” are both open-source compilers implementing the Common Language Runtime and the .NET Framework libraries, which are required to run C# coded programs in OS different than Microsoft Windows. For example, the “Mono” project provides libraries to execute C# software in Mac OS, iOS, Ubuntu, Debian, OpenSUSE or Android. Comparing “Mono” and “DotGNU”, the first has the most developed set of libraries and higher compatibility, consequently being our choice for our application.

Installing “Mono” in Ubuntu is as easy as typing “sudo apt-get install mono-complete” in the command line, waiting then for the procedure to complete. After doing so, we were able to run executable programs developed for the .NET Framework in Ubuntu.

4.3.2. ARM Hard Float and Soft Float

Continuing with the explanation of the RISC processors of the previous section, we need to explain the existence of a pre-processor called ARM Vector Floating Point (VFP) [36]. This piece of hardware is typically integrated as an extension into newer
ARM SoC chips, and its main advantage is that it increases performance in all floating-point operations (float and double variable types). To mention, ARMv5 architectures were the first to include VFP in 1995, and since then posterior architectures had always brought this Floating Point Unit.

The “Hard Float” or “Soft Float” nomenclature stands for whether an OS is ready or not to take profit of the VFP architecture. Consequently, “Soft Float” systems execute the floating-point operations by converting float values to integer values, performing the operation, and then reconverting again the number to float. This result in “Hard Float” systems being faster for intensive floating values applications, getting improvements ranging from 4% to 40% in terms of performance [37].

Getting these increased performances may lead to think that nowadays all software is based in “Hard Float” compilations, but this is not 100% true. Actually, because since some years ago CISC processors (Intel and AMD) have been almost the unique option in the market (by price and performance), software developers have not paid much attention to program using the advantages of this Floating Point Unit. However, this has been changing recently, and nowadays the trend is to move progressively into this type of architecture.

The answer behind why do we need to know these facts is that, after extensive research of why the software was not working in our PandaBoard, we found that the “Mono” Project is only compatible with “Soft Float” compilations. This made us to eventually work always with “armel” (ARM architecture emulator) distribution versions, and thus to discard the new and higher performance “armhf” (ARM Hard Float) compilations. In the case of Ubuntu, the latest “armel” available distribution is the 11.10 version, being the 12.04 and 13.04 both “armhf” (and this is why we installed Ubuntu 11.10 in the PandaBoard).

4.3.3. Adapting the VAS to Ubuntu

The portability of software coded from a particular OS to another can be sometimes hard to be performed, basically due to the differences in the frameworks used and in the abstraction layers they have. In our case this was not an exception, and the adaptations we had to made can be separated into two aspects: the visual forms and the serial port management.

On one side, and referring to the visual forms, all USAL services rely on the use of the C# Windows Forms [30] proposed by Microsoft Visual Studio. These forms provide an easy way to create Graphical User Interfaces (GUI), but they can only be executed in Microsoft Windows OS. In this sense, the “Mono” project offers an open-source GUI alternative set of widgets called Gtk+ [38], which is available for all platforms and distributions. However, changing from Windows Forms to Gtk+ is not a straightforward task, requiring a major restructuring of the applications.

Additionally, and because we opted for a “Server” version of Ubuntu without Desktop or graphical interface (pursuing better performance), we decided to adapt VAS service to not have any visual widget or screen. This was made by suppressing all code relative to the “VASCConsole” (visualisation in figure 4.4), not interfering with the
normal operation of the VAS. Regarding the FPMa, no further action was required as it is a service without GUI.

On the other side, the serial port management required some more changes to the code, which are related to the reception of data and to its connection.

- With respect to the reception of data, Windows OS offers the control of the data received by means of raising an event called “DataReceivedEvent” (as it states, it is raised when new data bytes are available in the buffer, providing the software is using it an easy way to receive packets). However, as in Ubuntu this event does not exist, it is necessary to code an alternative solution: there must be an independent Thread [30] in charge of decoding every single byte received (one for each serial port we may want to use).

- About the second change, we need to explain that connecting to a port in Windows has no major effects than a single “Open” call procedure, and then you can receive data straightforward. But when using Ubuntu, connecting to the serial port makes the connected device to be rebooted, like if it was just connected (because the OS do not constantly take care of the serial port, it just restarts it when someone attempts a connection). This had a main implication when connecting to the ArduPilot board: it provokes to restart the calibration procedures of all its sensors, like it was just powered on. For this reason, we had to reprogram the connection procedures in the Communication Manager of the VAS (giving the board time to be calibrated), and additionally getting all parameters and recorded flight plan at start-up (to get the “Home” or “Starting point” GPS position).

When all these changes in the code were made, and after performing some minor tunings in Ubuntu (like changing language local settings to “en-US” to read dotted-values as floats instead of integers), we got eventually both VAS and FPMa services running in Linux with the PandaBoard.
4.4. The Power Subsystem

The power subsystem is one of the core elements in any electronic system, and therefore in any unmanned system. In our case, and as it is strongly dependant on the design of the entire platform, it can be separated into two different groups: powering the elements requiring direct supply from the power source and powering the ones which will be powered by internal components (this can be observed in figure 4.5).

![POWER SUBSYSTEM diagram](image)

**Figure 4.5** Power subsystem main and internal connections (red and green respectively)

The internal connections can be easily summarised as following:

- The APM 2.5 board will be powered by the USB connector, which will be connected to the PandaBoard.

- The main Communications Subsystem (or radio module) will be powered also by a USB coming from the PandaBoard.

- The APM Board will be the responsible of powering the RC Radio receiver by means of their 3-pin (Signal-Positive-Negative) input connectors, as well as the steering servo of the car.

- Finally, the GPS sensor will be connected to one of the APM Board serial ports, which will provide it the required energy.

On the contrary, the explanations of the main connections require additional considerations, being the power source its cornerstone. In this sense, RC cars (and planes, copters, etc.) run nowadays basically on Lithium-Polymer batteries (also known as Li-Po), whose main characteristics are to be lightweight, to have high capacities with high discharge rates, and to be fairly inexpensive (examples in figure 4.6).
These batteries come in different sizes, voltages and capacities (depending on the number and dimensions of the capacitive cells used), but the typical ones used for RC vehicles have from 3 to 5 cells and capacities from 2200mAh to 5000mAh. The output voltage depends on the number of cells they have and on how much are they actually charged, being the resulting tension the sum of each individual cell (which range from 2.7V when they are discharged to 4.2 at their maximum capacity).

The use of these batteries means that our platform must be capable of handling different input voltages, either by discharge or by using batteries with different number of cells. In this sense, as we have two elements remaining to power we prefer to treat them separately (both can be seen in figure 4.7):

- On one side, a brushless motor (which is actually our Throttle motor) requires big amounts of energy to run, implying that they cannot be powered by a small board like our PandaBoard or even the APM. Instead, they are typically sourced by an Electronic Speed Controller (or ESC): a device which takes the energy directly from the battery and, depending on the signal received in its input pin, drives more or less energy to the motor. Hence, the battery will be connected to the ESC, and the ESC will power the motor/s.

- On the other side, as the PandaBoard needs 5V to operate we need to find a regulator providing a 5V stable output no matter the input tension it has. Regarding the intensity, the PandaBoard consumes 1-1.5A and the APM around 1A, so it must be able to provide at least 2.5 Amperes. The safer and most used alternatives for powering computers are quite big (almost like the APM board), so finally we decided to try a cheap and small RC 12V to 5V regulator we find in the internet to see if it could match our requirements. Surprisingly, after doing some tests we proved it was a feasible solution because of giving a stable 4.98V output (3A maximum) with inputs from 5.4V to 20V, being consequently our selected regulator.
4.5. Assembly of the platform

Once every component of the air segment has been selected, we can now proceed with the assembly of all the components. To do so, we must take into account topics like the shape and the characteristics of our RC car, the size of the components, the length of the wires, etc…

4.5.1. First assembly

Taking profit of the car’s mounting pivots of its plastic cover, the first approach was to place there a planar surface, allowing us to mount everything in an easy way and to test the correct behaviour of the complete system. After some soldering and some carpentry work, the result can be seen in the next photographs (figure 4.8):

![Figure 4.8 First Assembly of the Air Segment](image)
As one can see in the photographs, by doing this work we noticed some of the problems we will need to overcome in latter configurations (like trying to not use the built-in USB connections as they require much more space, or to try to place some components together due to the shortness of their wires, etc.). However, because of placing this surface offered us some advantages for our particular car model, we will keep this idea for further designs. These odds are:

- First is to have enough space between it and the chassis to place there the Li-Po battery, which as long as it is not bigger than 2cm width it can be interchangeable.

- Second is that assuming the battery is the heaviest part of the whole platform, placing it in a lower point lowers also the centre of gravity. This will help us in providing stability of the rover at turns and to partially avoid front uplifting when speeding up.

- And third is to have relatively large free space at the back to place different radios, enabling us to test them and to try their diverse combinations.

With this assembly we started then to test the entire platform as it will finally behave, facing the first software problems and the communications' limitations. As they are both explained in next chapters, here we are presenting only the mechanical problem it had: as the components are not protected, and although we are on the ground (safer than in air), any misbehaviour of the rover can make it turn around and break the boards. Therefore, we need to design some kind of enclosure to prevent this failure of the system.

### 4.5.2. Box design

Apart from the need of component protection, designing and building a new assembly for the platform has additional motivations:

- As the goal of this thesis is to develop a test bed platform for other vehicles, the final design has to be somehow interchangeable between them. Because of its size (28x14x4cm plus the USB connectors and the regulator), with our prior assembly this may not be feasible at all, thus requiring a different approach in this direction.

- Additionally, having the components and its cables distributed along this wide sheet of plastic makes the platform a bit unstable, tending to oscillate laterally. This movement, apart from being undesirable for any electronic device, makes the APM:Rover firmware not to perform the navigation accurately, as the magnetometer and the gyroscope provide it pretty confusing and unsteady values.

Consequently, we need to create an enclosure for all the electronics in the smallest form factor possible, whose requirements can be summarised in the following points:
• It has to contain all the elements (PandaBoard, DC Regulator, GPS, RC Receiver, APM:Board and radios) inside a box, being the antenna(s) and the ESC the only external components. The reason for the antenna is its size and the possibility to change it without effort, and the ESC is because it is dependent on the vehicle used (for instance, a multi-copter will have from 4 to 8 ESC).

• The SD card of the PandaBoard should be accessible without opening the case, being the objectives twofold: firstly it will allow us to make changes in the OS, or to change to another version, without the need of being in the laboratory (swapping the card); and secondly it may help ICARUS to avoid possible theft of the platform, as everything is stored in the SD card and therefore it cannot work without it.

• Regarding the ports and connections the box must provide, the access to the DB9 Serial Port has to be granted, as well as the connections for the antenna(s) and the power supply. Additionally, we need to present to the outer the pins for all the possible servo-motors and motors any vehicle could have, being a 8x3pin connector the minimum in this sense (likewise the APM:Board which has 8 fixed output ports to control from rovers to octo-copters).

• Because it is meant to be eventually flown, the whole construction should be as lighter as possible, trying to avoid stiff materials like iron or metal. In this sense, and as it is a test bed and there are components whose led indicators we may need to look at, it will be an advantage to have some transparent faces in this box. However, the entire structure has to be stiff enough to protect the components in the case a fall or a crash takes place, being somewhat of a trade-off between weight and toughness.

• Finally, and stressing what has been said previously, it has to be as smallest as possible if we want the platform to be used for driving future vehicles.

Taking all these characteristics into account, we started to design some alternatives by means of a Computer Aided Design software program (CAD). To do so, we modelled some of the components and made a two and three-dimensional representation of them, in such a way that we were able to infinitely change their position to find the best distribution possible (see examples in figure 4.9).
Finally, and after trying some different structures in thin plywood (or MDF), we ended up with the following solution built exclusively with transparent 0.2mm thick plastic and 1mm thickness aluminium frame (see figures 4.10, 4.11 and 4.12). Its main characteristics are summarised in the next table 4.3)

Figure 4.9 CAD Illustrations for the Air Segment box

Figure 4.10 Connectors and visible LEDs detail of the final box
Figure 4.11 Wooden first box prototypes

<table>
<thead>
<tr>
<th>Final Box Specifications</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Size</strong></td>
</tr>
<tr>
<td><strong>Weight</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Voltage Input</strong></td>
</tr>
<tr>
<td><strong>Power consumption</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>On-board computers</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Internal sensors</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>External connectors</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Wireless connectivity</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Table 4.3 Final box specifications
Hence, what we finally obtained is a transparent and really lightweight enclosure for all our components. Apart from protecting the electronics, assembling this box has resulted in a closed-product completely exchangeable with any other RC vehicle. Additionally, the on-board computer provides a great flexibility, being possible to load any kind of software (in our case, the USAL “Air Segment”), or to manage many different types of “payloads”.

Finally, and foremost, this box will allow ICARUS to reuse it in any further vehicle, thus being a perfect test bed platform for UAS.
Chapter 5

THE COMMUNICATIONS SUBSYSTEM

As it was pointed out previously, the main communications subsystem is one of the core components of any UAS architecture. Its importance raw in the fact of being the only link between the ground station and the unmanned vehicle, therefore its malfunctioning will probably lead to the loss of the system. Consequently, here reliability is not a trade-off, it actually is mandatory for a correct operation of the entire platform.

As a summary of the functionalities of the communications subsystem, it is meant to provide waypoints and mission management actions to the “air segment” (in our case the rover), while receiving its telemetry and payload information.

5.1. Serial Communications

Regarding the different types of wireless connections (wired is not an option for obvious reasons), the most used radio links in RC applications and UAS environments are point-to-point serial radios. They are characterised for using typical and easy to use UART interfaces (Universal Asynchronous Receiver-Transmitter) to connect to a computer, and for offering longer ranges than traditional networks.

To sum up, we may explain some essential features of UART-based devices:

- They require 3 pair of connection pins (6 in total) to be used: Ground and Positive voltage (0V and Vcc) to power the radio, Transmission and Reception (Tx and Rx) for incoming and outgoing data, and Clear-To-Send and Request-To-Send (CTS and RTS) for data flow control.

- Related to their transmission speed, it can be configured by the common UART “baudrate” property, being the number of bits transmitted per second. Its typical values are 9600, 19200, 38400, 57600, 115200 and 230400bps, but actually they are not the speed at which data is transmitted. Because of each data byte consists of a start bit (low voltage), 8 data bits and a stop bit (high voltage) [39], the final transmission rate of the UART protocol can be up to 80% of the baud rate (8 data bits out of ten sent bits). An example of this transmission method is given in figure 5.1.

- As it uses separated pins for Tx and Rx, UART is a full-duplex communication protocol, providing transmission and reception of information at the same time (between the computer and the radio module). However, we will see later that the transmission speed between radios will depend on the air data rate, not on this baud rate.
In any case, by using this standard serial transfer protocol we will be able to change between pairs of radios without the need of changing anything in the rest of the system, posing an important advantage for our test bed platform.

5.1.1. Testing MAREA Serial configuration

As the starting point for developing the data link between the two segments we had the MAREA “gateway” functionality for serial ports, which was actually the responsible of routing MAREA messages outside an Ethernet LAN network.

As part of the “Publish” and “Subscription” MAREA philosophy explained in the first chapter, the gateway connects publishers and subscribers on both sides of a serial port. This was performed in a direct way without using any particular protocol, neither for acknowledging the correct reception of packets nor for an efficient way of transmitting messages.

To test the functionality of the gateway, we placed two different computers and connected them via a standard RS-232 serial cable (or using a virtual serial port). In this scenario (where the channel is 100% reliable), there was no loss of communications, making the architecture to work as expected with no major modifications. Additionally, after some bug corrections we also tested the same setup with the PandaBoard (using an USB-to-RS232 adapter) obtaining similar results.

After proving the feasibility of using the “gateway”, we also performed tests with different baud rates (from 230200bps to 19200bps). By doing so, we noticed some slowness in the communications when lowering the baud rate to speeds below 115600bps, making us to be suspicious that low bandwidths may present a problem to the good operation of the system. In the case of using the PandaBoard this effect was even stressed, essentially due to its lower computational capacity.

To do all these tests, the only change that has to be done is in the configuration MAREA file “network.xml”, which alerts MAREA that we will use the “serial” type of connection with a determined baud rate:
Example:  
\[
\begin{align*}
\text{\texttt{serial address=\"COM1\" baudRate=\"115200\"/}} & \text{ in the Ground Station (Win)} \\
\text{\texttt{serial address=\"/dev/ttyUSB0\" baudRate=\"115200\"/}} & \text{ in the Rover (Ubuntu)}
\end{align*}
\]

5.1.2. MAREA Serial Messages

Because we wanted to know what was exactly happening, we analysed the messages sent between the two stations.

Firstly, it is important to explain that the “publishing” and share of information within MAREA is based on the definition of four communication primitives: variables, events, remote invocations and file transmissions [4]. As a summary, we can state:

- **“Variables”** are values typically sent at regular intervals, where the loss of one or more messages has no major effect in the performance of the system.

- **“Events”** are also values or pieces of information (like “variables”) with the characteristic that reception by all subscribers is guaranteed. Some examples can be the uploading of a flight plan, or the raising of a system alarm.

- **“Remote invocations”** (also called “functions”) are used basically for what its name stand for (basically to activate/deactivate actuators, or to request a calculation from other service).

- And last, the **“File Transmission”** primitive is used to send large sets of file-structured information, like XML configurations or mission-oriented data.

But even though the whole communication within MAREA is structured by combining these four different primitives, it uses an additional framing method to exchange information (adding sender information, receiver address, information about the packet, etc.). This framing is based in a generic C# object serialization method [30] which, although it is not the most efficient practice, allows MAREA to send entire classes to other services. To summarise, the following are the different objects operating across the middleware when USAL is being used:

- **“Discover”**: It is a message used for all the services to try to find subscribers and/or publishers to/for the variables and events they publish or want to be subscribed to. Its typical average size is of 450 bytes.

- **“SlowData”**: With an average size of 470 bytes, it usually carries the blocks of information like telemetry variables, flight plans, etc.

- **“Publish” / “Unpublish”**: These two messages provide MAREA the information of the variables and events they are offering, whether they are starting to publish them or not doing it again. Their medium size is of 490 bytes.
• “Subscribe” / “Unsubscribe”: They are sent to subscribe to remote events or variables, and are typically around 590 bytes.

• “SubscribeACK”: Is the final step of the subscription process, to make the publisher be aware that it has a subscriber. Its medium size is of 500 bytes.

• “CallFunction” / “ReturnFunction”: As they name state, they are the procedures to call a “Remote Invocation” or “Function”. Their size is variable depending on the information they carry, but as an illustration here are the values for uploading a flight plan of 12'6Kbytes (two “CallFunction” of 862 and 12606bytes, and two “ReturnFunction” of 325bytes each).

Consequently, what we could see is that this serialization framing method of the MAREA messages is causing the packets to be quite big compared to the information they carry, what actually will be a major problem in the system architecture. To see this problem in detail, take for example the case of a telemetry position packet:

• It is framed as “SlowData”, and its size is of 437bytes.

• The desired information it carries is only 3 “double” (3*64bits) and two “float” (2*8bits) types of objects.

• Then, having 32 bytes of information and 405 header’s bytes, it gives a massive result of a 1265% of over-heading (what will introduce a lot of problems in our serial channel).

So in the end, we could extract some conclusions from the analysis of the serial MAREA messages:

• First, although the serialization method used is really flexible (it allows easy integration or change of any new object), it puts in a lot of additional data to the packets. This, in fact, makes the messages to be really inefficient, as they are too big compared to the information they carry.

• Second, this size of the packets presented another issue, which is the slowness of the PandaBoard and its not-so-good serial channel management (because they require more computation power to be processed).

• Third, MAREA mechanisms for subscribing variables and events have one possible flaw: for every variable/event a service publishes to the middleware, it sends periodically a “Discover” message to find suitable “subscribers” or “publishers”. In scenarios where not all data published is being subscribed (like in our “Lite” version of USAL), this results in additional packets sent to the channel, thus occupying part of the available bandwidth.

• And last, some telemetry information of the VAS is sent at a frequency of 20Hz, what combined with the large size of the packets produces the saturation of the channel.
Anyway, although we saw that it has some limitations it was proved that the solution of a serial communication was feasible, being then time for getting rid of the connection cable.

5.2. Wireless Communications

As in the case of the choice of the PandaBoard, the election of a pair of radio devices requires some study to be done. The starting point is to define the requirements they have to fulfil, being the following our basic guidelines:

- Relatively long range capabilities: if we may want to use it for a UAV we will need to reach high separation distances.

- A data transfer rate similar, or higher, to that of a serial cable (115kbps): we have seen that MAREA serial communications is not optimised, so in order to get an acceptable communication link we will need as much throughput as possible.

- Reduced power consumption: as we will be running a small car with a small battery, power savings will be always welcome.

- Serial communication with the computer, what will ease its integration and allow interchangeability.

- Moderate costs: having a small car (with an inexpensive autopilot) and placing a pricey radio would not have much sense at all. Besides, as a test bed platform it might be prone to be broken at some point (hopefully not), and we might not want to break a 1000$ radio if we could have gone for a much cheaper one.

5.2.1. Available radio modules

Having stated these requirements, the different radios we had available at ICARUS were the following ones:

![Available radio modules](image)

*Figure 5.2 Available radio modules (3DR, XBee, XTend, and RN-XV)*
• **The 3DR radios**: Designed by the same company than the APM:Board, these open-source couple of “transparent” radios cost less than a 100$ in their webpage [17] or half that price over the internet. Basically, they are supposed to provide a range of about a mile with a transmission power of 100mW, always working in the 433MHz frequency band.

• **XBee modules**: Theoretically, these modules provide long-range capabilities with a power output of 50mW [40]. They use the DigiMesh (proprietary) network protocol at 900MHz, allowing the retransmission of corrupted messages at a price of 90$ both [41].

• **XTend Long range Digi radios**: Being the most powerful (and expensive) pair of radios we had, they can transmit at a power up to 1W with an announced range of 40 miles (65Km). They work in the 900MHz frequency band, and their cost is around 370$ [42].

• **RN-XV Wi-Fi module**: And the last module we had was a 2.4GHz Wireless Ethernet 35$ device, which has a low power output of 16mW but a transfer data rate up to 484kbps [43].

<table>
<thead>
<tr>
<th></th>
<th>3DR Radio</th>
<th>Xbee-Pro</th>
<th>Xtend LR</th>
<th>RN-XV Wi-Fi</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power Output</td>
<td>100mW</td>
<td>50mW</td>
<td>1W</td>
<td>16mW</td>
</tr>
<tr>
<td>Throughput</td>
<td>115Kbps</td>
<td>150Kbps</td>
<td>115Kbps</td>
<td>484Kbps</td>
</tr>
<tr>
<td>Frequency Band</td>
<td>433MHz</td>
<td>900MHz</td>
<td>900Mhz</td>
<td>2.4GHz</td>
</tr>
<tr>
<td>Cost</td>
<td>50-100$</td>
<td>90$</td>
<td>370$</td>
<td>35$</td>
</tr>
</tbody>
</table>

**Table 5.1 Available radio modules comparison**

Being the most important points the four detailed in the table above, we should discard any radio not fulfilling our previous requirements. As we can see, from the four the only radio that cannot suit our purposes is the Wi-Fi module, as its low power output will not be enough for a good outdoor data link. From the other three, we needed to perform further analysis to know which will be the best option possible.

To do such analysis, and because we need as much throughput as possible to deal with the MAREA serial packets, what we did is to develop a software program which will give us the real performance of every couple of radios:

• The basics have been to take a standard MAREA average packet of 420 bytes and to send it as much times as possible, in such a way that the port works at maximum speed.
• During a certain period of time, on the receiver we were checking the CRC values of the received packets, counting the good and the wrong messages received.

• After the time has been elapsed, we work out the resulting throughput taking into account only the correct ones, as it will be in a real scenario.

• To know also if the radios are full-duplex or half-duplex, we have performed these tests in the two configurations: only one sends and the other receives, and the two sending and receiving at the same time.

• As you will see in the table of results, we tested all the modifiable parameters of each radio, trying to find their limits and their best behaviour possible. This a brief definition of all of them:
  - The “baud rate” is the speed of the serial connection between the radio module and the computer.
  - The “air speed” is the data transmission rate between the two radios (the wireless channel).
  - The “ECC” stands for Error Correction Code, and is a functionality of the 3DR radios which include headers to try to correct few transmission errors at the receiver.
  - The “Op Resend” is also a functionality of the 3DR radios, which makes the radio resend packets when its buffer is almost empty.

Some of the best values we obtained with this software are summarised in the following table (5.1):
The conclusions we extracted from these tests were various:

- First, in relation to the “air speed” all radios are half-duplex (they cannot transmit and receive at the same time), what means they actually share the same radio channel for transmitting and receiving information. Although they are bidirectional (two-way communication), they are all half-duplex.

- Second, and this time related to the “baud rate”, both the 3DR and the XBee radios are full-duplex (can transmit and receive at the same time) radios. This is because in the serial connection between the module and the computer, there are two separated pins for transmitting and receiving information. Nevertheless, although the XTend radio endows these two pins, it acts as a half-duplex one (it cannot manage one side saturating the channel).
Third, we saw that at its best configurations they can transmit almost the same amount of data: around 80,000-90,000bps accounting for bidirectional communications. This corresponds to an “air speed” of 115200bps, whose 80% value (maximum available transmission bandwidth) will be 92,000bps. This means that it may be a virtual trade-off limit for these kind of serial long range radios, as the higher the air data rate the lower the sensitivity of the receivers (thus lower the range).

And last, we could see that the communications protocol they implement affect the general behaviour of the radios. For example, the 3DR radios is prone to get wrong packets (presumably due to its low-cost components), while the XTend has always a 100% of correct packets.

Finally, as the three radios have pretty similar throughputs in their best configurations (except for the XTend in unidirectional communications), we need a couple of extra arguments to decide which one will be our chosen option:

On one side, we will play the card of the cost: although the XTend radio has ten times higher power output, for our rover scenario it will surely not be necessary. Hence, in our case we will not use the XTend radio and go with the cheaper ones, but it could be chosen in the future when, for instance, placing the platform in an aircraft.

On the other side, we will look at the operating frequency: although our platform is not focusing to be working every day, using the free and available 433MHz frequency will be an asset when using them here in Europe.

Hence, and although they are the ones having more errors, the 3DR radios appeared to be the most balanced option for our purposes: their power output may give us medium range, they are inexpensive compared to the XTend, and they provide a full-duplex data throughput of 50,000Kbps. Furthermore, they operate in the 433MHz frequency, which is available here in Barcelona.

5.2.2. Range Comparison

Regarding the expected ranges with the three different radio modules, we can focus in what they say in their datasheets, or we can take its power output value as a reference.

If we assume the values the manufacturers say are correct, we may expect ranges of about 6 miles in the case of the XBee, around a mile with the 3DR, and up to 40 miles with the XTend modules. However, these values are usually overestimated and are taken considering really high gain antennas for the both radios. As this will not be the case, we will compare the different expected link budgets to eventually perform a series of real tests.
To be recalled, we need to know that we will have communication link when (5.1 is in dB units):

\[ \text{Rx Sensitivity} \leq P_{Tx} + \text{GainsAntennas} - \text{Losses}_{\text{FreeSpace}} \]  \hspace{1cm} (5.1)

As the losses in Free Space depend on the distance and on the frequency (5.2), if we isolate the expected range we end up with the next formulas (5.3 and 5.4):

\[ \text{Losses}_{\text{FreeSpace}} = 20\log\left(\frac{4\pi}{c} \text{dist} \cdot \text{freq}\right) = 20\log(d) + 20\log(fMHz) - 27.55 \]  \hspace{1cm} (5.2)

\[ 20\log(d) \leq 27.55 + P_{Tx} + \text{GainsAnt} - 20\log(fMHz) - \text{RxSens} \]  \hspace{1cm} (5.3)

\[ d \leq 10^\left(\frac{27.55 + P_{Tx} + \text{GainsAnt} - 20\log(fMHz) - \text{RxSens}}{20}\right) \]  \hspace{1cm} (5.4)

Because we do not know the gains of the antennas, neither the exact sensitivity of our devices (they say all have -100dBm at 115200Kbps air data rate), what we will do is to assume they share the same constant values. In this way, what we will compare is only the effect of the power output and of the frequency used (formula 5.5 and table 5.3).

\[ d \leq 10^\left(\frac{\left(P_{Tx} - 20\log(fMHz)\right)}{20}\right) \cdot 10^\left(\frac{27.55 + \text{GainsAnt} - \text{RxSens}}{20}\right) = 10^\left(\frac{P_{Tx} - 20\log(fMHz)}{20}\right) \cdot K_{\text{constant}} \]  \hspace{1cm} (5.5)

<table>
<thead>
<tr>
<th>Power Output (mW)</th>
<th>Frequency (MHz)</th>
<th>Max Distance (\cdot Constant)</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>16.96970004</td>
<td>900</td>
</tr>
<tr>
<td>100</td>
<td>20</td>
<td>433</td>
</tr>
<tr>
<td>1000</td>
<td>30</td>
<td>900</td>
</tr>
</tbody>
</table>

**Table 5.3** Numerical comparison of the radio ranges

As we can see, the XBee modules are the ones with lower range, while the difference between the XTend and the 3DR is theoretically only the 52%. It has to be pointed out, though, that in these equations we are assuming to have the same intrinsic noise levels in all radios, what is not actually true (the 3DR has higher noise levels than the XTend modules). Anyway, as the 900MHz band in Spain is used by
GSM phone cells it will probably be much noisier than 433MHz, so at the end we can presume similar noise levels on both radios (being our calculations pretty plausible).

Finally, it was noticeable that when we tested the different radios in a real outdoor scenario, the ranges we were actually having were really bad (for the 3DR radio it was less than 30 meters). To know the cause of this problem, what we did is to test the antennas in the Radio-Frequency (RF) laboratory. With a spectrum analyser device we were able to see how adequate for our frequencies these antennas were. In the next figure (5.3) the result with the stock 3DR antennas is presented, where the lower the value at our desired frequency (for the 3DR is 433MHz) means better performance.

![Figure 5.3 433MHz Stock antenna tested with the spectrum analyser](image)

Because you can see they were not selective at all (being a source of noise for the radio module), what we did is to build 3 pairs of antennas at 433MHz and 2 more at 900MHz. To specify, the antennas created are the most simple and easy to build [44]: dipoles, monopoles and coiled antennas, which greatly improved our final range. In order to precisely tune them, we took profit of the spectrum analyser, whose results can be observed in figure 5.4.

![Figure 5.4 433MHz Designed antennas tested with the spectrum analyser](image)
Although these antennas were not actually directive antennas (as they are omnidirectional they radiate in all directions), they allowed us to cover all the 350 meters our testing spots have. The best results were obtained with the dipole antennas (we kept one for the ground station), but in the car we placed a coiled antenna because of the size of a 433MHz dipole (it is about 30cm long, too much for our little car). So, in the end, we proved that the stock antennas were not giving good performance at the radiating frequencies, and consequently we built some new ones providing much better capabilities.

5.2.3. Testing MAREA Serial Wireless configuration

Being consequent with our previous choice, we connected the 3DR radios and tried to run MAREA in the same conditions than in section 5.2 (with the radios instead of the serial cable). Under these circumstances, there was communication between the air and ground segment, but MAREA did not handle well the interconnection between the stations. Basically, the problems we were facing were two:

- One was related to the subscription of variables and events, which were not all successful. Following a completely random distribution, some values were not subscribed by the services, leading to the malfunctioning of the entire architecture.

- The second was not being able to perform some actions when involving two or more services, for instance when trying to upload a flight plan or changing the autopilot state. It looked like the receiving services were not aware of the transmitters sending the actions, or like the two sides were ignoring themselves after a while.

However, because the only difference between this scenario and the serial cable case was the byte errors introduced by the 3DR, the reason behind these issues was easy to come up with: we were losing messages and MAREA was not prepared to manage this problem. Once we proved that fact, it was clear that we needed to develop some kind of protocol to handle this type of real (non-ideal) defective communications.

5.3. MAREA Serial Transport Protocol

As it has been explained previously, and because the communications within MAREA services had always been done through Ethernet (TCP and UDP protocols) or cable connections, it had never been necessary to implement a communication protocol to recover from faulty messages. However, we have seen that sending messages via serial radio devices implies the possibility of losing some bytes (due to interferences, synchronisation errors, or even chip deficiencies), confirming with the tests that USAL architecture was not ready to deal with them.
To solve this problem, we will focus our work into two differentiated areas:

- The first will be in the detection of defective messages, either if they are because of corrupted information or because of receiving incomplete packets.
- And the second will be in the task of ensuring the reliability of the communication, retransmitting the data that has not been received properly.

### 5.3.1. Byte error detection

In any telecommunications system, when sending information from a transmitter to a receiver the most typical method to detect faulty packets is by using the Cyclic Redundancy Check (CRC) method [45]. It can be summarised as being the addition of a series of bytes at the end of every packet, whose value corresponds to the result of a calculation which takes into account all the packet bytes.

Being said that, MAREA was actually using this CRC correction method to detect erroneous messages. However, to understand why MAREA was producing the errors mentioned in the previous section we need to explain how the receiving method in the serial port was being performed: the reader interprets the first bytes received as the total length of the packet, parsing this determined number of bytes into a MAREA packet.

By using this decoding method, it happened that if a byte (or bytes) is corrupted then the CRC will warn us to discard it with no further consequences; otherwise, if a single byte is lost during a transmission then the parsing gets altered, causing the subsequent messages to not be decoded properly, and therefore leading to the total loss of the communication link (this was what caused the two segments to be ignoring themselves).

Consequently, to prevent this lag (or drift) in the decoding procedure, we need to define in some way the start and the end of every message. This will allow the software to correctly split and parse the received bytes, no matter if the bytes have been lost or only been corrupted.

To do so, we decided to adopt the protocol used by another commercial autopilot (the UAVNavigation AP04 [46]), which uses the Data Transfer Protocol (DTP) defined in the IETF standard RFC171 [47] (and updated by the RFC264 [48]). This decision was motivated by the small amount of additional overhead it produces, providing efficiency and basic error detection.

This protocol uses the definition of two control characters to format and build the packets: DLE (Data Link Escape) and ETX (End of Text). These characters are both common ASCII values [49] (016 and 003 respectively), and are commonly used for the framing of packets in serial communications. In our case, the procedure for building these frames is the following (it can be observed in figure 5.5):

- Firstly, we take the MAREA packet and we calculate its CRC32 value (CRC method calculation with a 4 bytes result), placing it at the end of the packet.
• Secondly, we double the existing DLE values. This means that if there are bytes with a value of “16”, we place a DLE byte before them to alert they are information data and not a DLE character.

• Then, we place a DLE value at the start and the combination DLE+ETX at the end to delimit the frame. Then, the packet is now ready to be sent through the serial port.

[Diagram of packet framing]

Figure 5.5 Packet Framing of the MAREA Serial Protocol

Oppositely, the method to decode the packets was essentially the inverse process:

• We receive bytes from the serial port till the combination DLE+ETX is found.

• Then we remove the DLE characters, taking into account that DLE+DLE = one “16” value byte.

• Now we can take the last four bytes (CRC32 value) to check if the data has been corrupted, obtaining eventually the initial MAREA packet if the result is satisfactory (if not the message is discarded).

With this packet structure we were now capable of discarding any faulty messages within MAREA serial communications, being able to keep receiving correctly the subsequent packets and messages. Nevertheless, and although we tried to use the USAL architecture with only these modifications, it still presented the first problems of not managing the subscriptions in a proper way (because of not having implemented a retransmission protocol for lost packets).

5.3.2. Enabling Safe Retransmission

To provide a robust and reliable communication link between the air segment and the ground station, the first thing we need to do is to identify every message in order to know which messages will be received and which ones will be lost. In this sense, this
can be made by means of a sequence number, which was introduced in the header of every serial MAREA packet.

Two bytes were reserved for this sequence number, giving capacity to up to 65 thousand messages before restarting the sequence (thus preventing message sequence duplicity). After the introduction of this number, we then defined a basic transmission protocol to ensure a safe communication (it can be visualized in the following figure 5.6):

![Figure 5.6 MAREA Serial Retransmission Protocol](image-url)
• On the side of the sender, when a message is sent it will retain its information together with its exact “sent time”. This value, which will be monitored by an independent loop, will be taken to calculate the estimated “Resend Time”. If this time has been passed without receiving the corresponding ACK, this independent loop will queue again the same message to be retransmitted.

• On the side of the receiver, every time it decodes properly a “Safe” message it will send a small Acknowledgement message (ACK with the sequence number) to the sender. Contrarily, if the CRC checking fails, it will try to extract the sequence number from the first bytes of the packet, to then send a Negative Acknowledgement (NACK) message.

• Related to this Negative Acknowledgement, when the transmitter receives one of these messages it will look for the corresponding packet in the copy of its packets sent, to consequently resend it and refreshing its new “Sent Time”.

This was the basic structure for the protocol as our first approach. However, as we wanted to provide the best link quality possible to the communications system, we looked for a way of reducing the computation power needed for this solution.

The first thing we did was to ask ourselves if the different MAREA messages were actually all requiring an ACK. As the answer was negative, we decided then to distinguish the nature of each one to treat them adequately.

As it was explained in section 5.1, MAREA publishing and share of information is based in the definition of four primitives: Variables, Events, Remote Invocation and File Transmissions. From these four, Variables are the values that do not require a safe transmission, so we do not need to use the retransmission protocol for this type of information.

So finally, to implement the desired transport protocol with the differentiation of this mandatory and optional data, we defined the following four types of messages:

• **Fast Message**: Packets that do not require reception confirmation. Can be lost without affecting the performance of the system.

• **Safe Message**: Packets transporting important information that require a safe reception. If needed, they must be retransmitted.

• **ACK**: Acknowledgement message sent when correctly receiving a Safe message. It contains the sequence number of the message received.

• **NACK**: Negative Acknowledgement sent when a Safe Message have been received incorrectly. The suggested sequence number is extracted from the header of the packet.
In order to decode each packet as fast and efficient as possible, a single byte identifying the type of message was inserted in their headers. Arbitrary values of 171, 018, 035 and 036 were assigned respectively. Consequently, with this additional byte the header of every serial packet has now the following structure (figure 5.7):

<table>
<thead>
<tr>
<th>Mes. Type</th>
<th>Sequence Number</th>
<th>MAREA Message Data</th>
<th>CRC32</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1 byte)</td>
<td>(2 bytes)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 5.7 MAREA Serial packet structure

With this structure we eventually got the Serial Transport Protocol working properly, and although it already is a reliable environment for the exchange of messages, we will try to improve it by introducing some new features.

5.4. Data Link Enhancements

Once the Serial Transport Protocol has been defined, we can introduce some modifications to provide flexibility and adaptability to suit many different scenarios. To improve the data link, the adjustments can be focused in three different fields: by reducing the bandwidth used for retransmitting packets, by diminishing the latency of fast packets, and by allowing the utilisation of two pairs of radios.

5.4.1. Packet Fragmentation

About reducing the retransmission bandwidth, we need to analyse what can be done to improve the communication error rate. In this sense, because the probability of byte errors depends on the radio devices, then we are expecting to have always similar ratios of byte error rates (BER) if we do not change them.

Nevertheless, the probability of getting a byte error in a message increases with the size of the packet, so we can actually do something to try to reduce the bandwidth used for retransmissions: to small down the size of the messages sent by fragmenting them. This fragmentation process requires cutting down the messages at the sender and rebuilding them at the receiver, what has been programmed in the following way:

- On the side of the sender, if messages exceed a defined a maximum size of packet, they become chopped into smaller pieces. The procedure takes the bytes sequentially to send them as separated messages, sharing the same “Sequence number” but with the addition of a “Fragmentation Sequence” and a “Fragmentation Total” byte (to allow the receiver reconstruct them).

- On the other side, the combination of both “Fragmentation” values manages the reconstruction procedure, where messages are stored until all fragments have been received. In the case of FastMessages, not-completed fragmented messages are removed after a certain period of time.
To be highlighted here is the fact that the maximum allowable message size will be dependent on the selected fragmentation size. Being said that, and because the “Fragmentation Sequence” is a one-byte length sequence, the maximum allowable number of fragmented pieces for a single message will be 255 and therefore \( \text{MaxPacketSize} = 255 \times \text{MaxFragmentedSize} \).

And finally, after the addition of the fragmentation feature the framing of the serial packets have been designed with the following structure (before the addition of the DLE + DLE/ETX framing bytes, in figures 5.8 and 5.9):

<table>
<thead>
<tr>
<th>Mes. Type (1byte)</th>
<th>Sequence Number (2bytes)</th>
<th>Frag. Seq. (1byte)</th>
<th>Frag. Total (1byte)</th>
<th>MAREA Message Data</th>
<th>CRC32 (4bytes)</th>
</tr>
</thead>
</table>

**Figure 5.8 Serial Packet Structure (Safe and Fast Messages)**

<table>
<thead>
<tr>
<th>Mes. Type (1byte)</th>
<th>Sequence Number (2bytes)</th>
<th>Frag. Seq. (1byte)</th>
<th>Frag. Total (1byte)</th>
<th>CRC32 (4bytes)</th>
</tr>
</thead>
</table>

**Figure 5.9 Serial Packet Structure (ACK and NACK messages)**

As we can see, the permanent resulting heading added to every MAREA message is of 9 bytes, without taking into account the DLE/ETX combinations plus the DLE duplicated values. Consequently, the minimum total header will be of 12 bytes, which in a typical 420bytes MAREA packet results in a 2’85% of the total size.

### 5.4.2. Reducing Latency

Latency, which can be defined as the delay between the time a message is sent and it is received, appears when an application tries to send more bytes than the available bandwidth. In our case, it typically occurs when the full bandwidth of the radios is being occupied, either by “Safe” retransmissions (because of a weak radio link) or by high “Fast” transmission frequencies (like when using a Joystick or receiving high frequency telemetry).

To alleviate this effect, what we proposed is to limit the maximum number of “Fast” packets that can be queued in MAREA, discarding the older messages that cannot still be sent. In this way, we restrict the highest latency time, ensuring we always get the newer messages possible.

Because both the “Safe” and the “ACK/NACK” queues (explained in the previous section) are mandatory for a proper operation of MAREA services, we could not
apply the same strategy with them (but we have already prioritised them against the “Fast” messages).

5.4.3. Dual Serial Transport

The last improvement we developed for the Serial Transport Protocol came up when facing the lack of bandwidth to cope with all MAREA messages. In this sense, what we added is the possibility of using two pairs of radios instead of one (see figure 5.10), in such a way that the resulting data throughput will be doubled.

![Figure 5.10 Dual Serial Transport solution for high data rates](image)

To illustrate the advantages of this choice, we have seen in the tests that the 3DR radios had performed well in unidirectional tests, being able to manage higher data transfers (60000bps) with its maximum baud rate (115200bps). Hence, using two pairs of radios (each pair in one-way communication) could be a feasible solution for getting high data transmission rates.

In any case, we must take into account when using this feature to correctly configure the two pair of radios, selecting not the same frequencies in order to avoid possible interferences.

5.5. Software implementation

Taking into consideration all the explained protocols and procedures, we ended up with a flexible, robust and reliable data link for the USAL architecture: it is independent of the radio-link devices used, and combines retransmission of determined messages, packet fragmentation and queue limitations.

In order to implement this transport protocol, both the “dual” and “single” options have been programmed around the same classes, being easier to modify for future developers. In this sense, we have arranged the code using the following guidelines (the detail of the resulting structure can be seen in figure 5.11):

- The interface between MAREA and the developed Serial Transport Protocol is the class (or object) named “SerialTxRx”. This sends and receives the data from the “SerialTxRx”, which is the manager of the rest of the classes.
The class responsible for the transmission of packets class has three different queues of messages, which are feeding continuously the serial port: SafeMessagesQueue, FastMessagesQueue, and ACK_NACKQueue.

These queues are filled by the “PacketManager” class, which is in charge of applying our framing methods to the packets sent from MAREA.

By having these three differentiated queues, we are able to prioritise the ACK/NACK messages over the rest in the transmitter loop, being the “Safe Messages” the next in the list. This makes us to be sure that the retransmission of packets takes place only when a packet is lost, and not because the processing time has taken too long.

The retransmission of messages is managed by another independent loop in the code (in “SafeMessagesTx”), which is responsible for looking at the elapsed time in the list of sent “Safe Messages”. Additionally, it also handles the retransmission because of a received NACK message, refreshing the “sent time” as needed.

On the other side, in relation to the reception of data every message received (combination DLE+ETX is found) is sent to a “PacketReceiver” method to be executed in a “ThreadPool” [30], which decodes it and sends it to MAREA without stopping the serial receiver.

Figure 5.11 MAREA Serial Transport C# Classes

Regarding how to set it up, the user only has to modify the “network.xml” MAREA file to configure all the aspects in the Serial Transport Protocol:
• When using one single pair of radios, we need to define the name of the port ("address"), its speed ("baudRate"), the time to wait in milliseconds between message retransmissions ("TimeToResend"), the maximum packet size for the fragmentation feature ("MaxPacketSize"), and the maximum allowable size for the “Fast” messages queue ("MaxMessageQueue").

**Default values:**

```xml
<serial address="COM1" baudRate="57600" TimeToResend="1500" MaxPacketSize="300" MaxMessageQueue="20"/>
```

• Similarly, when using two couple of radio devices we need to define the abovementioned parameters, together with the names and the communication speeds of the two serial ports (without forgetting to specify the "serialduo" option).

**Default values:**

```xml
<serialduo addressTx="COM1" baudRateTx="115200" addressRx="COM2" baudRateRx="115200" TimeToResend="1500" MaxPacketSize="300" MaxMessageQueue="20"/>
```

### 5.5.1. Testing the Serial Transport Protocol

With this architecture, and after the complete integration of the software and the hardware, we tested in the laboratory the general behaviour of this protocol with the following satisfactory results:

• We proved that retransmission was made properly (with ACK/NACK packets working as expected), and that the “TimeToResend” value plays a key role for a proper operation of USAL.

• Fragmentation feature alleviated the channel when retransmitting, although its value is a trade-off between performance and data addition (for instance, packets with 150 data bytes will be growing a minimum 8% because of header bytes).

• NACK packets were actually more useful than expected, as they severely reduce the time for retransmissions.

So concluding, we developed and tested a complete solution for our initial problem: the data link between the air and ground segments of the USAL architecture. Furthermore, the adaptability and reliability of the solution makes it useful for future systems with no effort at all, what is in fact the cornerstone of any test bed platform like our rover.
Chapter 6

USAL TESTS AND CONCLUSIONS

Being the final objective of this thesis and the main purpose of any test bed platform, the last step of this work has been to perform as much tests as we could of the USAL software architecture.

6.1. Tests environment

To explain how these tests have been performed, we need to explain the characteristics of all the components involved: from the ground station to the testing spots. Thus, we will be defining the variables and constants of the tests environment (going from single unitary tests to complete architecture), in such a way that the results lead to the correction of an error or to the improvement of a functionality.

6.1.1. Ground Station

As it has been introduced in the first chapter, the basic services of the USAL architecture are both the Flight Plan Monitor (FPMo, illustrated in figure 6.2) and the Flight Monitor (FMo, in figure 6.1). The first one is in charge of the flight plan management, while the second allows you to change VAS states and the use of a joystick device. Because the flight plan management is the core component of any unmanned system, much more importance has been given lately to the development of the FPMo (it requires more reliability and functionalities than the FMo).

Figure 6.1 Flight Monitor (FMo) interface
Being said that, and taking into account that different services are usually “subscribed” to different events and variables, we decided to carry out the majority of the tests with the FPMo and using the FMo only when needing a Joystick device. By doing so, not only will we saving transmission bandwidth, but also will be limiting the number of actors involved in the tests (less components imply better error resolution). To be able to cope with the management of the states, the ICARUS group modified the FPMo and added this functionality by a new pull-down menu.

6.1.2. Flight Plans: Testing spots

To perform any test, action, or automatic navigation with this architecture, the main step is the elaboration of a flight plan: the definition of all the patterns and different waypoints that we may want to fly with an UAV. In this sense, the ICARUS solution is a very powerful and flexible tool, allowing a large number of alternatives and predefined patterns (more information in [50]). To be mentioned, and although in our case we are not actually flying anything, we will keep this nomenclature to match the final objectives of the test bed platform.

Being consequent with the decision taken in the first chapter (with the APM Mission Planner and the autopilot), we continued avoiding difficult terrain like sand or grass at all times. This fact, along with the lack of precision of the GPS used (accuracy of +-2.5 meters), forced us to look for spacious and asphalted places where we can peacefully do our desired tests.
Finally, we found that two parking spots which suited our requirements, being big enough and with very low transit affluence (necessary if we want to keep our platform alive). Hence we developed both flight plans containing the basic types of navigation patterns (all are explained in [50]), which are described as follows:

- “Track to a Fix”: Defines a straight line between two points, which is the basic structure of any path.

- “Hold”: Based on the airplanes’ holding patterns, they are formed by the definition of a coordinate (fix) and its two horizontal dimensions. In our case, and because the rover is able to do 90 degrees straight turns, we will see they can be set out as rectangles.

- “Scan”: Being a relatively new feature of the USAL architecture, ICARUS has developed algorithms to make scans of a determined area in an easy and proper way. By specifying some arguments like the area perimeter, the separation between tracks, or the entry and starting sides, they compute the best alternative and translate that into a series of waypoints. More information can be found in [51].

Taking profit of the iterative flight plan and of the flight alternatives’ split functionalities of the USAL software, we designed the flight plans as a main “mission” loop (it can be seen in the next figure 6.3) preceded and succeeded by a “departure” and “approach” procedures (to act always as an aircraft will). By doing this, we can provide almost infinite waypoints to the autopilot, while being able to change the flight plan and performing different updates and alternatives (our test flight plans can be observed in figure 6.4).

![Figure 6.3 Flight Plan basic structure](image-url)
6.1.3. Entire architecture

Finally, to sum up all the components present in the different tests we will present the complete overview of the system (figure 6.5), composed by:

- The Ground Station: because USAL does not require high computation capabilities, it can be run in any available laptop. There will be typically running the FPMO, using also the FMO when requiring the test of Joystick navigation.

- The Communications subsystem: conformed by the failsafe RC radio and the pair (or two) wireless radio modules, being typically used the 3DR radios as it has been explained in the previous chapter.

- And the Air Segment, which is formed by the rover, the battery and the constructed box with all its components. In the on-board computer, there will be running the Virtual Autopilot System (VAS) and the Flight Plan Manager (FPMa) services.
6.2. Tests results and modifications

In this section we will summarise not only the final problems that still require some corrections, but also the ones encountered when performing tests at the different development stages. Although they may be already solved, it could be necessary for further improvements to have them recorded.

In relation to how the tests have been performed, we should explain that they have been carried out together with the correction of software errors (also known as bugs), actually being an intensive part of this entire Thesis. To perform the tests the typical steps were the following:

- First, by using our platform as a real UAS test bed, we tried to use a USAL functionality in the real scenario while logging all possible data (information, values, communications, etc.).

- When any error arose, we analysed the logs and attempted to replicate the error in the laboratory where all variables can be better controlled (either in static environments or using the simulator).

- Once we found the source of the error, we fixed it and carried out the test again like in a circular process.

By attaching ourselves to this method, we ensured we did not waste time changing too many things at the same time, what could have eventually led to lose the advantages of using this test bed platform.

6.2.1. Middleware communications

One of the main problems we faced along the testing procedures is related to the communications within MAREA, particularly with the serial transport protocol and the definitions of variables, events, and functions.

When exchanging data between MAREA services, the dichotomy between treating them as variables, events, or functions, is always present. Nevertheless, as USAL is being used basically in simulated environments (with LAN 10/100Mbps bandwidth and zero losses), all three primitives have been working almost in the same way: they always instantly reach its destination. Hence, it had not mattered which primitive we want to use for each value, and how many different information we will probably not need.

However, using real devices and prioritising some messages over the rest, affects not only its latency, but also produces the loss of “variable” packets which are necessary for a proper operation of the system. On the contrary, information unnecessarily classified as an event or function may lead to more “Safe” packets to retransmit than needed.
To illustrate this last problem, we could typically observe in some of our tests the loss of the data link between the ground station and the rover after a certain period of time, being very difficult for the system to recover normal operation. This situation took place every time we had a weak radio link, when the retransmission of lost “Safe” packets saturated the available bandwidth.

Concluding, and although we will take a look in detail at every service, the publication of MAREA primitives needed some adjustments to fulfill the requirements of a real platform. Some of them have already been made, but a major restructuring of the exchange of information would be beneficial for future developments.

In addition to this variable/event decisions, and as it was introduced in section 5.1.2, MAREA mechanisms for subscribing variables and events are not optimized (there is a periodical sent of “Discover” messages to find desired “subscribers” or “publishers”). This method is producing additional packets sent to the channel, thus occupying some of the available bandwidth. Consequently, in order to avoid these messages to be transmitted, it will be important to check in every real scenario that we are only publishing and subscribing the information we are going to use.

6.2.2. Virtual Autopilot Service

Regarding the Virtual Autopilot Service, and thanks to the intensive testing we already made during its development (chapter 2), there were no major modifications needed. However, we will place here some of the problems we encountered during the process:

- Firstly, we needed to drastically reduce the telemetry messages sent to the ground to not stress the serial channel (using 20Hz telemetry requires 62,800bps), leaving only values like position, angles, speed, the heartbeat (with the AP state) and the mission time, all in a 2Hz rate (28,000bps of required bandwidth).

- Secondly, and because the autopilot states we are using are not the standard for UAVs, we needed to allow all special transitions between them (for instance, it is not possible to go from parking to automatic navigation with a plane, so USAL typically prevents that behaviour).

- And finally, we came up with a bigger problem when changing or uploading a modification in the flight plan: the autopilot queue was cleared. This was because the FPMa alerts the VAS that an update has been made by calling the “ClearWP” function, what in fact was clearing the VAS queue. Nevertheless, its meaning would be to alert the VAS to say “Hey, I am going to reload the queue of WP, so keep listening”. Anyway, although a solution for this problem has been made (by waiting until WPs are received before uploading them), this feature should be in higher layers of the VAS in such a way that the autopilot layer has always waypoints in its queue.
6.2.3. Flight Plan Manager (FPMa)

Taking an XML flight plan and turning it into a series of waypoints can sometimes be the focus of several problems. However, as this is one of the core components of the USAL architecture, its operation is very reliable when using it with planes: it performs changes in the flight plan, reactions, etc.

In the case of a rover, though, there is the need for adjusting some of its parameters, where the main one is the minimum distance between waypoints (needed to be lowered to 4-5 meters to our car instead of, for instance, the 100m of an aircraft). Another one is when generating the waypoints for doing a reaction manoeuver, where the distances are not adapted to the small ones of our little car (waypoints are generated far away from our position).

Regarding the declaration of events and variables, and because this service has no interface, it is plenty of different events to allow the visualization and control of the automatic navigation. Nevertheless, it may be necessary to redesign which values are events and which may be variables, as not all may need a “safe” transmission. In any case, we will see now that not all have to be subscribed by the other services, so we need to be cautious about this when flying in real scenarios.

6.2.4. Flight Plan Monitor (FPMo)

As it was previously explained, the FPMo is our main Ground Station, capable of performing all actions except the Joystick navigation. But we must take into account that it has been designed for simulated UAS environments, hence everything is adapted to aircraft operations and it is subscribed to almost any variable, event, or function published to the middleware. This has some implications that we are about to summarise:

- The first aspect is the unaffordable quantity of packets sent and received to/from the serial channel in a real scenario: being subscribed to almost anything produces too many retransmitted messages when having a weak link, resulting in the loss of communications.

- As in the case of the FPMa, the variables and events definitions may need a bit of a rethought. To illustrate that, the change of autopilot state was made by means of a “variable”, so when the packet was lost (or discarded) the autopilot never received the order (as this was mandatory for our rover, this particular change is already made).

- On the other side, the FPMo is not ready for small flight plans and actions like our rover will do. For example, the tab for updating and modifying the parameters of a scan (figure 6.6) has fixed units in Nautical Miles (1,852Km), thus with our typical values being around 4-5 meters (0.0025NM) makes it impossible for us to use these kind of updates.
Another problem we faced during our tests was that the service was not prepared also for transmission latencies. This means that in some occasions, because of serial transmission (or retransmissions) took longer than Ethernet-connected stations and it was not accounting for that, the program crashed or did not perform as it was expected. Although these few problems are already solved, they may be taken into consideration in further developments.

And finally, the last one is related to the map visualization: the different map layers available are not capable of painting well enough our testing spots, as they need closer resolutions (zoom) for our relatively “small” flight plans (as compared to UAVs).

6.2.5. Flight Monitor (FMo)

Although the FMo was not used for controlling the normal operation of the rover, we made the tests to try to use the Joystick to drive it.

Apart from the variable/event/function dilemma pointed out with the other services, in this case the most important issue is the latency of the orders from the computer to the rover: from the moment we turn the joystick till the wheels really move can be up to 10 seconds of lag (at 57600bps baud rate).

This latency depends, apart from on the serial port speed, on the size of the “Fast” messages queue (which we have to select when defining the Serial Transport Protocol). That is because the Joystick is sending values at 20Hz with an approximated packet size of 375bytes each, hence sending more data than the available bandwidth (63.000bps only for joystick values).

When using the Dual Serial Transport option with a baud rate of 115200 in each pair of radios, as we are not using the whole bandwidth the latency of movement is about 1 second (minimum time to pack the message, send it to the air segment, decode it, and send it to the autopilot).
6.3. Conclusions

Finally, to end with this Master Thesis we are going to do a brief summary of all the conclusions we extracted from the different sections, developments and tests.

6.3.1. Defining the UAS test bed platform

Starting from the very beginning, the first conclusion we can say is that all the expected work has been completed, fulfilling in each part its correspondent requirements. Moreover, we have overcome all the difficulties we have encountered, and in some cases we developed more features than strictly necessary which surely will be very useful to future ICARUS research activities.

Also related with the first chapter, now that we have finished we can confirm that the choice of the rover as a test bed and of its APM:Rover autopilot was a extremely good choice. In this sense, taking into account the bunch of problems we have faced to get the whole system working, we realised that trying to use USAL right away with a real plane or a multi-copter would have been catastrophic or almost impossible (at least without breaking or losing a dozen of them).

6.3.2. The APM:Rover

Regarding the characterisation of the autopilot, we have seen that it offers pretty good capabilities for such a reduced price of its autopilot board. It is true that some aspects like parameter tuning or the performance of the board sensors can be improved, but for a price of 200$ for the entire system it is really worth the money. Besides, the most typical complain about APM-autopilots is that they only allow a maximum of 150 waypoints, but we have proved that this limitation can actually be overcome by using the standard and header-lightweight MAVLink protocols (so we can say we also have turned APM:Rover into a very powerful autopilot).

Another conclusion from this chapter is that open-source tools and software are great because they are free, but we must point out the problems of having so many people working in the same project and in their spare time: both the APM Mission Planner and the APM firmware source codes are both chaotic, lacking a coherent structure and therefore losing a lot in terms of efficiency. This not only produce unexpected errors, but also and more important limits the performance capabilities of the hardware (in this case, the APM board).

6.3.3. The USAL integration

In the case of the integration of the rover within USAL, it has been clear that a rover is a very simplified plane, thus it has been the best starting point to test the basic behaviours of the VAS. By doing so, we came up with the fact that it requires some
structural changes, in such a way that it allows easier integration with other possible autopilots.

Related with these structural changes, for instance, the division of the Autopilot Layer APL into Communication and Autopilot managements could be a plausible approach in further developments. In our case, doing so has provided cleaner and clearer objects in the programming environment, on top of getting the Communication Manager reusable for any MAVLink-based autopilot.

On another side, although it was pretty difficult to obtain the standard C# MAVLink DLL library ended up saving us a bit of time (as it assured us to be using standard MAVLink packets), so it was worth it.

Finally, another conclusion from this chapter was realising how important was to perform the tests as accurate as possible, getting always as much information as you can extract. For example, it was not until I logged everything that I could figure out that the errors I was facing were not because of the developed code, but due to a bad radio link. On posterior tests I followed the same strategy, what saved us a lot of time after the tests.

### 6.3.4. The assembly of the Air Segment

From this chapter, the main and first conclusion I have extracted is that sometimes things look much easier than they really are. In this case, although at first glance it appeared to be only to place some electronic components together, it was actually very demanding:

- Firstly, the choice of an on-board computer was easy, but making it work fine was really difficult: some different operating systems, bunches of programs to compare, lots of versions, etc. Moreover, it is not the fastest computer in the world, so changing anything took really long time. At the end, however, we finally managed to have a stable computer, giving us the capabilities we were requiring.

- Secondly, hardware work (carpentry, wires, soldering, etc.) takes a lot of time to be done properly and it is very easy to do any mistake. Anyway, once we have the result we should say it was worth it.

Apart from that, some other considerations can be made to the work performed in this chapter. One point is about the on-board computer, to say that mobile ARM boards are opening a whole new range of possibilities for embedded and aerial systems. Its low power consumption, small size, and growing computation power make them perfect not only for mobile phones but also for aerospace applications.

Another point to be made here is that sometimes the cheaper components may provide really good results if your requirements are not very restrictive (for instance, the regulator price was about 3$, but it performed perfectly for our application). Thus,
what is important is to define the requirements prior to buy or build anything, saving you not only time but money.

And finally, we must highlight that we developed a complete “Air Segment”, a whole platform that can be placed on-board any Unmanned Aircraft (UA). With the addition of a payload, we will have then all the typical systems of an UAS ready to flight in less than 10 minutes.

6.3.5. The communications subsystem

Because of all the variables involved (devices, antennas, services, types of packets, etc.), this subsystem was the most difficult to get working properly. However, we finally managed to have a flexible and very reliable communication protocol, with acknowledgements and retransmissions, fragmentation, prioritising, and limitation of queues. In this sense, as it is a good protocol for links requiring safe communication, it could be used in any MAREA UAS.

To sum up the work done in this chapter, we have seen the effect of the antennas, the power output, and of the operating frequencies in the expected range of a radio link. We have also learned to build and tune antennas, which actually perform better than the stock manufactured ones.

Additionally, we developed a program to test the performances of serial radio devices, which allowed us to see how they carry out the unidirectional and bidirectional communication (if they share the bandwidth or not, if they were full-duplex or not, their throughputs, etc.). This program is ready to be used with any couple of devices (there is no need to adjust anything), so it will always be usable for anybody.

Finally, as they are the users of the data link, we must point out the issue with the large packets of MAREA. In principle, we are using a relatively old (but reliable) version of MAREA, and this problem is going to be solved in coming updates. In any case, however, after these updates will be made we will need to pay special attention to this size of the packets, as they make the whole architecture impracticable for real scenarios.

6.3.6. The final tests

And eventually, it is in the last chapter where most relevant conclusions take place: when testing the whole USAL architecture in a real situation.

Starting with the size of the packets pointed a paragraph above, it has ended being the nest for many different misbehaviours of the platform: suddenly stopping, loosing telemetry, failed when uploading modifications to the flight plan, etc. In this sense, if this is not solved it will be very difficult to improve the developed platform, as we are obliged to reduce everything in the link to ensure the minimum required communications take place.
Anyway, we managed to test the complete architecture, proving that USAL is a powerful tool for managing UAS missions: you can perform scans, holds, iterations, updates of the flight plan, etc. in a comfortable way with its ground station. However, we must say that at this moment it is not an optimized platform for the air segment side, thus not being ready for small UAS scenarios.

Regarding the PandaBoard as an on-board computer, it has been a good choice capable of handling the big packets of MAREA while communicating with the autopilot at the same time. However, it sometimes suffered a bit of slowness, and sometimes it even got overstressed and was rebooted.

The reason behind the Panda problems can be the serial MAREA packets, but surely what it does not manage pretty well is the multi-threading (running different processes in parallel). To solve this problem, the Serial Transport Protocol and the VAS had already been coded with the minimum number of threads, but in the middleware every MAREA messages is sent as an independent thread, requiring a lot of computation power. A different implementation of the publishing/subscription, or the use of a “ThreadPool” instead of a new Thread, may greatly improve the behaviour of the Panda.

So finally, what we can extract from the entire Thesis is that we have implemented, built, and tested, a complete test bed platform for the USAL architecture, solving all the different problems we were finding during the process and consequently accomplishing the initial objective of this work.

6.4. Future Work

Once reached this point, we are going to propose a series of improvements to take into consideration for future work or developments. You will see that they will be mainly focused in the test bed developed platform, but as some of them affect the entire USAL architecture they could be applied to it.

- As it has been introduced in the previous section, the first possible field of work is in the MAREA communications system. Here the main objectives should be twofold: on one side should be to reduce the size of the messages, and on the other should be the management of these messages without creating new Threads for each one. If these two topics are solved in the next version of the software, then there will be a huge improvement in the capabilities of the architecture and of the platform.

- The underlying factor provoking the big sizes of the MAREA messages is also slowing the performance of the PandaBoard computer, and we are referring to the use of the C# programming language. Although it is a really good computer language (because of its flexibility and easiness in terms of coding procedures), it actually produces large objects in terms of size. Moreover, executing C# in the PandaBoard requires using the virtual platform Mono, resulting in more computational power required for pretty simple operations. Thus, another field for future development could be to port the VAS and the FPMa to another programming language, like C or C++.
Regarding the on-board computer, and because during the time this work has lasted there have appeared new models of small boards (like the BeagleBone Black [52]), another improvement could be to try to place everything into a smaller one.

Once the test bed has been developed, now it will be possible to test the rest of available services, being the focus of further developments. Similarly, the continuous updates performed to the services we have already tested will need to be constantly tested.

And last, but not least, as this was targeted to be the test bed for an UAS the next natural step should be to place the designed box on an aerial vehicle (either a plane, multi-copter, or even a helicopter), ending up the path started by this Master Thesis.
**BIBLIOGRAPHY AND REFERENCES**


[16] https://github.com/mavlink/mavlink

[17] https://store.3drobotics.com/products/3dr-radio


[19] https://pixhawk.ethz.ch/mavlink/


[22] http://armdevices.net/


[27] https://wiki.ubuntu.com/ARM/


[34] http://www.mono-project.com/


[38] http://www.gtk.org/


[41] https://www.sparkfun.com/products/9099

[42] https://www.sparkfun.com/products/9411


