TREBALL DE FI DE CARRERA

TÍTOL DEL TFC: Millora de la comunicació Aire-Terra del pilot automàtic Paparazzi

TITULACIÓ: Doble titulació: Enginyeria Tècnica Aeronàutica, especialitat Aeronavegació i Enginyeria Tècnica de Telecomunicació, especialitat Sistemes de Telecomunicació

AUTOR: Xavier Gibert González

DIRECTOR: Juan López Rubio

SUPERVISOR: Gautier Hattenberger

DATA: 17 de juliol de 2012
Títol: Millora de la comunicació Aire-Terra del pilot automàtic Paparazzi

Autor: Xavier Gibert González

Director: Juan López Rubio

Supervisor: Gautier Hattenberger

Data: 17 de juliol de 2012

Resum

El projecte Paparazzi és tant un software de pilot automàtic com un software d’estació de terra gratuïts per Vehicles Aeris no Tripulats, també anomenats drons. L’objectiu d’aquest treball és millorar la comunicació sense fils d’aquest pilot automàtic, inclòs tot el software que pot estar relacionat.

Bàsicament el que canviarem són els següents tres punts: En primer lloc, com els missatges de comunicació són manegats pel sistema, reduint les limitacions que té l’antic sistema. En segon lloc, redefinir el protocol de comunicació, fent-lo més complet i també eficient quan sigui possible. I en tercer lloc, crear una compatibilitat bàsica amb altres softwares d’estacions de terra implementant un nou protocol. Aquest protocol s’anomena Mavlink i és utilitzat per un gran nombre de pilots automàtics i estacions de terra.

Per implementar la millora en el software dos llenguatges de programació seran utilitzats: OCaml i C. Quasi tot el codi en C està programat en forma de Macros per reduir la necessitat de recursos computacionals degut a que és el llenguatge principal utilitzat en el costat del pilot automàtic. La estació de terra està desenvolupada en OCaml. Per crear arxius de configuració s’utilitza XML.

La decisió de fer aquesta millora prové de necessitats expressades en els fòrums de programadors de Paparazzi i també de les necessitats de l’equip Paparazzi ENAC. Necesitats d’incrementar el nombre de missatges de comunicació que poden ser definits, però implementant-ho d’alguna manera ordenada i no fent-ho més caòtic. La necessitat de tenir més llibertat quan s’utilitzen aquests missatges i els camps que contenen. I el desig de ser compatibles amb tercers softwares.

En aquest treball veurem com el software Paparazzi necessitava una renovació en alguns aspectes per que estava arribant als seus límits. Com fer-ho i estar preparats per futures aplicacions i un possible increment d’usuaris. I com fer tot això d’una manera eficient, simplificant l’experiència de l’usuari.
The Paparazzi project is both an autopilot and ground station free software for Unmanned Air Vehicles (UAVs), also called drones. The objective of this work is to upgrade the wireless communication of this autopilot as well as all the related software that can be involved.

Basically what we are going to change are the next three points: first, how the communication messages are managed by the system, decreasing the limitations that the old system has. Second, redefine the communication protocol, making it more complete and also efficient when possible. And third, create a basic compatibility with additional ground station software by implementing a new protocol. This protocol is called Mavlink and it is used by many autopilot hardware and ground station software.

To implement the upgrades two programming languages are going to be used: OCaml and C. Almost all the code in C, due to that this is the main language of the autopilot side, it is done in form of Macros to reduce the need of computing resources. The ground station is developed in OCaml. XML is used to create configuration files.

The decision of doing this upgrade comes from the expressed needs in the Paparazzi developer forums, and also the needs of the ENAC Paparazzi team. Needs of incrementing the number of communication messages that can be defined, but doing it in some ordered way and not making it more chaotic. The need of more freedom when using these messages and the fields that they contain. And the willing of being compatible with third party software.

In this work we will see how the Paparazzi software needed a renovation in some aspects because it was reaching its limits, how we can do it and be prepared for future applications and a possible grow of users, and how to make all of this in an efficient way, simplifying the user experience.
I want to dedicate this project to everyone who help me to realized, and help me during my stay in France:

To Juan López Rubio for getting me the possibility of coming to the ENAC school and guiding me in the realization of this document.

To all the Paparazzi ENAC team for the help provided. Specially to Gautier Hattenberger who help me in the design and implementation of the project, as well as the help configuring my own drone. To Murat Bronz for the information provided about its projects. And to Michel Gorraz and Matthieu Navarro for helping me in the construction of the drone.

I want to mention also Catherine Ronfle-Nadaud, chief of the Drones department in ENAC, for her support and for taking me to the IMAV 2012\(^1\) competition as a member of their team.

\(^1\)International Micro Air Vehicles conference and flight competition in Braunschweig, Germany.
INTRODUCTION .................................................. 1

CHAPTER 1. State of the Art ............................... 3
1.1. MicroPilot ................................................. 3
1.2. OpenPilot ................................................. 3
1.3. ArduPilot .................................................. 4
1.4. Paparazzi .................................................. 4
   1.4.1. Hardware ......................................... 5
   1.4.2. Software .......................................... 6
   1.4.3. Wireless link ...................................... 7
1.5. Mavlink ................................................... 9
   1.5.1. QGroundControl ................................. 10
1.6. Marea .................................................... 11
1.7. Conclusions ............................................. 11

CHAPTER 2. Design .......................................... 13
2.1. Limitations .............................................. 13
2.2. Paparazzi protocol ................................. 15
2.3. Mavlink compatibility ............................. 17
   2.3.1. Data types ...................................... 17
   2.3.2. Basic messages definition and implementation 18
   2.3.3. Mavlink protocol module .................... 18
2.4. Conclusions ............................................. 19

CHAPTER 3. Implementation ............................. 21
3.1. Pre-generator .......................................... 21
3.2. Messages module .................................... 26
3.3. Protocol changes .................................... 27
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>MicroPilot MP2028g.</td>
<td>3</td>
</tr>
<tr>
<td>1.2</td>
<td>Horizon.</td>
<td>3</td>
</tr>
<tr>
<td>1.3</td>
<td>OpenPilot board.</td>
<td>3</td>
</tr>
<tr>
<td>1.4</td>
<td>OpenPilot Ground Control Station.</td>
<td>3</td>
</tr>
<tr>
<td>1.5</td>
<td>ArduPilot board.</td>
<td>4</td>
</tr>
<tr>
<td>1.6</td>
<td>ArduPilot expansion board.</td>
<td>4</td>
</tr>
<tr>
<td>1.7</td>
<td>Paparazzi logo.</td>
<td>4</td>
</tr>
<tr>
<td>1.8</td>
<td>Paparazzi system overview.</td>
<td>4</td>
</tr>
<tr>
<td>1.9</td>
<td>Umarim v1.0 autopilot board.</td>
<td>5</td>
</tr>
<tr>
<td>1.10</td>
<td>IR sensors.</td>
<td>5</td>
</tr>
<tr>
<td>1.11</td>
<td>IR attitude computation.</td>
<td>5</td>
</tr>
<tr>
<td>1.12</td>
<td>Aircraft hardware overview.</td>
<td>5</td>
</tr>
<tr>
<td>1.13</td>
<td>Ground station software.</td>
<td>6</td>
</tr>
<tr>
<td>1.14</td>
<td>Ground control system.</td>
<td>6</td>
</tr>
<tr>
<td>1.15</td>
<td>At left the telemetry messages window and at right two real-time plotter windows monitoring different values.</td>
<td>7</td>
</tr>
<tr>
<td>1.16</td>
<td>Communication between different elements of the system.</td>
<td>8</td>
</tr>
<tr>
<td>1.17</td>
<td>XBee module.</td>
<td>8</td>
</tr>
<tr>
<td>1.18</td>
<td>Special XBee for the ground station.</td>
<td>8</td>
</tr>
<tr>
<td>1.19</td>
<td>Paparazzi protocol packet.</td>
<td>9</td>
</tr>
<tr>
<td>1.20</td>
<td>Communication messages format in the XML definition file.</td>
<td>9</td>
</tr>
<tr>
<td>1.21</td>
<td>Mavlink logo [14].</td>
<td>9</td>
</tr>
<tr>
<td>1.22</td>
<td>Mavlink protocol packet.</td>
<td>10</td>
</tr>
<tr>
<td>1.23</td>
<td>QGroundControl software [13].</td>
<td>10</td>
</tr>
<tr>
<td>1.24</td>
<td>QGroundControl 3D view with aircraft model [13].</td>
<td>11</td>
</tr>
<tr>
<td>1.25</td>
<td>Marea packet options [15].</td>
<td>11</td>
</tr>
<tr>
<td>2.1</td>
<td>Schema of current messages definition system.</td>
<td>13</td>
</tr>
<tr>
<td>2.2</td>
<td>Schema of proposed messages definition system.</td>
<td>14</td>
</tr>
<tr>
<td>2.3</td>
<td>New Paparazzi protocol packet anatomy.</td>
<td>16</td>
</tr>
<tr>
<td>2.4</td>
<td>Mavlink protocol packet anatomy.</td>
<td>19</td>
</tr>
<tr>
<td>3.1</td>
<td>Simplified folder tree of Paparazzi project.</td>
<td>21</td>
</tr>
<tr>
<td>4.1</td>
<td>Field alignment error.</td>
<td>38</td>
</tr>
<tr>
<td>4.2</td>
<td>Class ID range error.</td>
<td>38</td>
</tr>
<tr>
<td>4.3</td>
<td>Repeated class ID error.</td>
<td>38</td>
</tr>
<tr>
<td>4.4</td>
<td>Repeated message ID error.</td>
<td>38</td>
</tr>
<tr>
<td>4.5</td>
<td>Receiving test message in class 1.</td>
<td>39</td>
</tr>
<tr>
<td>4.6</td>
<td>Receiving test message in class 2.</td>
<td>39</td>
</tr>
<tr>
<td>4.7</td>
<td>Link agent debug output with bytes in hexadecimal.</td>
<td>40</td>
</tr>
<tr>
<td>4.8</td>
<td>Messages window showing a message with new data types fields.</td>
<td>41</td>
</tr>
<tr>
<td>4.9</td>
<td>QGC Window to connect the link modem.</td>
<td>41</td>
</tr>
<tr>
<td>4.10</td>
<td>QGC Communication console.</td>
<td>42</td>
</tr>
<tr>
<td>4.11</td>
<td>QGC Message inspector.</td>
<td>42</td>
</tr>
</tbody>
</table>
LIST OF TABLES

1.1 Autopilots for civil UAV applications overview. .......................... 12
2.1 Example of message classes distribution. .................................... 15
3.1 Functors and respective configurations for the module Messages returned. . 26
### SOURCE CODES

3.1 `messages_conf.xml.example`: Example file to determine which classes will be included in the compilation.

3.2 `Makefile`: Piece of the main Makefile file.

3.3 `messages.xml`: Example of generated file.

3.4 `Makefile`: Command to call the macros generator with a specific target.

3.5 `Makefile`: Messages file generator call depending of configuration file.

3.6 `downlink_msg.h`: Final downlink macros file.

3.7 `downlink_msg.h`: Final uplink macros file.

3.8 `pprz_transport.h`: Paparazzi protocol implementation file.

3.9 `link.ml`: Link agent source-code.

3.10 `downlink.h`: General protocol implementation file.

3.11 Macro example with variable length array.

3.12 Macro example with fixed length array.

3.13 `mavlink.xml`: Mavlink module configuration file.

3.14 `pprz_transport.h`: Macro to put the packet header in Paparazzi protocol.

3.15 `mavlink_transport.h`: Macro to put the packet header in Mavlink protocol.

3.16 `mavlink_transport.c`: Array to extract the last checksum value with the message ID.

3.17 `ap_downlink.h`: Example of a periodic Attitude message macro in Paparazzi.

3.18 `mavlink_downlink.h`: Example of a periodic Attitude message macro in Mavlink.

3.19 Paparazzi message with names corresponding to values of field.

3.20 `mavlink.xml`: Example of Mavlink telemetry configuration file.

3.21 `weasel_mavlink.xml`: Example of Mavlink airframe configuration file.

4.1 `c_manual_packets_sender.c`: Testing C program to manually send messages.

4.2 `c_manual_packets_sender.c`: Testing C program to manually send messages.

A.1 `gen_messages.xml.ml`: Part of the pre-generator to obtain messages.xml.

B.1 `messages_common.telemetry.h`: Common telemetry class macros file.

C.1 `ap_downlink.h`: Periodic downlink messages implementation for fixed-wing aircraft.

C.2 `datalink.c`: Received uplink messages handling for fixed-wing aircraft.

C.3 `ac_server.ml`: Received downlink messages handling on ground.

D.1 `pprz.ml`: Part of the interface of the library pprz.ml.

E.1 `pprz.ml`: Ground library, some lines of data types handling.
INTRODUCTION

The Paparazzi project is both an autopilot and ground station free software for Unmanned Air Vehicles (UAVs), also called drones. The objective of this work is to upgrade the wireless communication of this autopilot as well as all the related software that can be involved.

Basically what we are going to change are the next three points: first, how the communication messages are managed by the system, decreasing the limitations that the old system has. Second, redefine the communication protocol, making it more complete and also efficient when possible. And third, create a basic compatibility with additional ground station software by implementing a new protocol. This protocol is called Mavlink and it is used by many autopilot hardware and ground station software.

To implement the upgrades two programming languages are going to be used: OCaml and C. Almost all the code in C, due to that this is the main language of the autopilot side, it is done in form of Macros to reduce the need of computing resources. The ground station is developed in OCaml. XML is used to create configuration files.

The decision of doing this upgrade comes from the expressed needs in the Paparazzi developer forums, and also the needs of the ENAC Paparazzi team. Needs of incrementing the number of communication messages that can be defined, but doing it in some ordered way and not making it more chaotic. The need of more freedom when using these messages and the fields that they contain. And the willing of being compatible with third party software.

In this work we will see how the Paparazzi software needed a renovation in some aspects because it was reaching its limits, how we can do it and be prepared for future applications and a possible grow of users, and how to make all of this in an efficient way, simplifying the user experience.

The document is divided in chapters. Starting by the State of the Art of the autopilots for civil UAVs, focusing in Paparazzi and other third party software. Followed by the design of the upgrade and the implementation. Then an evaluation chapter that will show how is everything working correctly after the upgrade. And a chapter with the environmental impact of the Paparazzi project is included at the end.
Upgrading Paparazzi autopilot Air-to-Ground Communication
CHAPTER 1. STATE OF THE ART

To start the document, some of the main autopilots software and hardware that we can find for Unmanned Vehicles civil applications are going to be reviewed. Focusing in the one that is going to be upgraded, Paparazzi. But also the one that is going to be used to test the compatibility with third party autopilots by the protocol Mavlink, QGroundControl.

1.1. MicroPilot

MicroPilot is an autopilot board specially designed for micro UAVs, also called MAVs. The last released board weighs in 28 grams only with a 10x2cm size. The price goes from 2000 to 8000 USD depending of the version and its capabilities. The ground station software is called Horizon.

![Figure 1.1: MicroPilot MP2028g.](image1)

![Figure 1.2: Horizon.](image2)

1.2. OpenPilot

OpenPilot is an open source project. Its CopterControl platform is an autopilot board for small UAVs. Even its name is CopterControl, can be used for all kind of aircraft: fixed-wing, rotor-craft and multi-rotor-craft. The board is really small (36x36mm) fitting in the smallest aircraft that are being build lately. The price of the board is 90 USD.\(^1\) The ground station software is called OpenPilot Ground Control Station.

![Figure 1.3: OpenPilot board.](image3)

![Figure 1.4: OpenPilot Ground Control Station.](image4)

\(^1\)The original CC board has been discontinued due to lack of availability of the gyro chips that were used. The replacement board is called the CC3D, and it's currently being test flown to verify the re-design performance [6]. (June 2012)
1.3. ArduPilot

ArduPilot is a full-featured autopilot based on the Arduino open source hardware platform [7]. The Ardupilot can be used for all kind of unmanned vehicles: cars, boats and aircraft. The board size is 30x47mm. The price of the autopilot board is 25 USD, 60 USD the version MEGA, but exists an expansion board by 150 USD with the main sensors needed. To change and upload the autopilot software is used the Arduino IDE platform. Any software version that we upload can have different communication protocols implemented. Depending of which one they have we can use different ground station software. Mainly are using Mavlink, see section 1.5.

Figure 1.5: ArduPilot board.  Figure 1.6: ArduPilot expansion board.

1.4. Paparazzi

Paparazzi is a complete system of hardware and software for autonomous aircraft as well as complete ground station mission planning and monitoring software. Using a bi-directional datalink for telemetry and control [12].

Figure 1.7: Paparazzi logo.  Figure 1.8: Paparazzi system overview.
1.4.1. Hardware

The autopilot hardware consists basically in a micro-controller board that includes sensors like: accelerometers, gyroscopes, magnetometers, barometers... Depending of the board and the kind of aircraft designed for (fixed-wing or rotor-craft). It has also some connectors for the different servos of the aircraft, connectors for external sensors and modems, and a USB connector to flash\(^2\) the software with the desired configuration.

The price of the boards goes from 120 to 200 USD depending of the board. And the sizes from 25x50mm for fixed-wing aircraft and from 50x50mm for rotor-craft.

![Figure 1.9: Umarim v1.0 autopilot board.](image)

The main external sensors that we can find are GPS, if it is not included in the board, and IR sensors pack if we want to use ground thermal image to compute the attitude instead of the IMU (accelerometers and gyroscopes).

![Figure 1.10: IR sensors.](image)  ![Figure 1.11: IR attitude computation.](image)

In the next figure we can see the aircraft hardware overview. There are represented the main subsystems that can be found in a small unmanned air vehicle.

![Figure 1.12: Aircraft hardware overview.](image)

\(^2\)The action of flashing the autopilot board consists in replacing the code stored in the program memory by a new one.
1.4.2. Software

The paparazzi software is an open source project hosted on the internet. Everybody is free of downloading it and contributing to the development. The software consist in two parts:

- The **ground station** is mostly developed in OCaml\(^3\) language. It works under Linux and MacOS. It’s possible to run it on Windows using Virtual Machine [3].

- The **airborne autopilot** is developed in C language, using macros\(^4\) when possible to minimize the CPU load.

From the ground station we can select the different configuration files that modifies the autopilot and compile a version for our specific aircraft. Then we need to connect the autopilot board to the ground station to upload this specific version that includes mainly the airframe, the flight plan and the telemetry messages that will send. Once it is unplugged, all the communication is via wireless modem.

In the airframe file there are setting values like which communication protocol to use, maximum and minimum actuator positions, levels of battery that will raise warnings and alarms...

We can monitor the aircraft and change its flight plan from the ground using the Ground Control System window.

![](image1.png)  
![](image2.png)  

---

\(^3\)Objective Caml: is the main implementation of CAML adding a powerful module system and object-oriented layer. Caml is a general-purpose programming language, designed with program safety and reliability in mind. It is very expressive, yet easy to learn and use. Caml supports functional, imperative, and object-oriented programming styles. It has been developed and distributed by INRIA, a French research institute in computer science and applied mathematics, since 1985 [1].

\(^4\)A macro is a fragment of code which has been given a name. Whenever the name is used, it is replaced by the contents of the macro. There are two kinds of macros. They differ mostly in what they look like when they are used. Object-like macros resemble data objects when used, function-like macros resemble function calls [16]. The macros help to optimize the airborne code because it is very dynamic, it depends of many different configurations. Instead of calling functions from other functions depending of global variables, definitions of functions with different names are used. The selection of which name has to have the function is made only when compiling the software and not every time the function is called.
The flight plan works based on Blocks and Way-points. There are blocks like LAUNCH, CIRCLE and EIGHT. And the way-points are references for these blocks. For example when we put the aircraft in the Launch block it will put throttle to maximum and move the elevator to follow a climb path, the direction of climbing is defined by the path between the Home way-point and the Climb way-point. After reaching some defined altitude it will change of block automatically to, for example, start making circles around a Standby way-point.

We can define many different blocks and change between them automatically with exceptions or manually by selecting the block. We can also change the reference way-point during a mission. With a combination of all these procedures we can create a full automatic mission including Take-off and Landing.

We have also other tools to watch and graphic telemetry values.

![Figure 1.15: At left the telemetry messages window and at right two real-time plotter windows monitoring different values.](image)

The messages window shows all the messages that we are receiving from an aircraft. Each message flashes with a green light when is received and if the period between receives is more than one second a counter of seconds appears. If we click one message we can see at right all the fields and their values.

The real-time plotter window shows one field value in a time plot. We can see how they develop during different stages of flight. There is also a plotter window for messages registered in a log file from previous flights.

### 1.4.3. Wireless link

The schema of communications between different elements of the system it’s shown in the next figure:
Upgrading Paparazzi autopilot Air-to-Ground Communication

The wireless link is the part that connects the ground software with the autopilot software. It can use simple serial modems like the XBee modules. Paparazzi has its own protocol for this communication but it is compatible with the API of the XBee modules too. It's up to the user to choose the one that prefers. For other type of modems Paparazzi protocol is mandatory.

The protocols send different messages predefined in a XML file encapsulated in packets adding some other information. There are two kind of different messages travelling in this link: the uplink messages, or datalink (ground-to-air) and the downlink messages, or telemetry (air-to-ground)

With the messages definition file, Paparazzi generates C macro files to allow the microcontrollers to read the packets, extract the values of the fields from the payload and to generate the packets to send.

This communication system has some limitations right now. Limitations in the number of different messages that we can define and the values that we can send. In the sections 2.1. and 2.3.1. these issues will be explained extensively and a design solution will be found.
To know more about Paparazzi UAVs see Annex G where it is described the process of construction of a drone.

### 1.5. Mavlink

Mavlink is an open source communication protocol used for different types of unmanned vehicles, not only aircraft but terrestrial and maritime vehicles too. It can be implemented in many autopilots like ArduPilot.

It has its own message definitions and users can create new definition files with the desired messages for control and telemetry.

The packets anatomy is slightly different to the Paparazzi one. As we can see in the next figure they are using one *packet sequence* byte to control the lost of packets and a *component ID* byte to distinguish messages coming from different elements of the same vehicle.
1.5.1. **QGroundControl**

The software QGroundControl is the main ground station software that uses Mavlink protocol to communicate with the vehicles. It is also in open source and works under Windows, MacOS and Linux with Wine.

The software has integrated Google Earth what gives the possibility of watching the aircraft travelling in a 3D environment. There is an option for creating widgets\(^5\) to control and monitor that we can relate with our own defined messages.

---

\(^5\)The widgets in QGroundControl are little windows with controls like buttons or slide-bars that we can link to some message field to monitor its value or control it by sending messages with the desired value.
Some other Ground station software currently using Mavlink protocol are: HappyKillmore’s Ground Control Station [8], ArduPilot Mega [9] and Copter GCS (Android application) [10].

1.6. Marea

The Marea protocol is currently used in the EETAC University as a communication protocol for UAVs and unmanned vehicles in general. It is very dynamic and powerful compared to other existent protocols. It can use different classes with different packet formats and even sending predefined data structures. It is also capable of controlling lost packets and resending them. But it requires high computing performance. For example the checksum is located at the beginning of the packet what means that all the information has to be processed before start sending the packet. These characteristics that this protocol has make it really good for big UAVs but not so much for MAVs, with the processing capabilities very limited.

1.7. Conclusions

After reviewing all these UAV autopilots we can conclude of the existence of many software and hardware dedicated to civil UAV applications. Most of the software are open source projects and the only sold thing is the hardware needed.
<table>
<thead>
<tr>
<th></th>
<th>MicroPilot</th>
<th>OpenPilot</th>
<th>ArduPilot</th>
<th>Paparazzi</th>
</tr>
</thead>
<tbody>
<tr>
<td>Price (USD)</td>
<td>2000-8000</td>
<td>90</td>
<td>25-60 (+150)</td>
<td>125-200</td>
</tr>
<tr>
<td>Dimensions (mm)</td>
<td>100x20</td>
<td>36x36</td>
<td>30x47</td>
<td>25x50-50x50</td>
</tr>
<tr>
<td>Weight (g)</td>
<td>28</td>
<td>8.5</td>
<td>20</td>
<td>9-24</td>
</tr>
<tr>
<td>GPS</td>
<td>External/Integrated</td>
<td>External</td>
<td>External</td>
<td>External/Integrated</td>
</tr>
<tr>
<td>Performance</td>
<td>High</td>
<td>Medium</td>
<td>Low</td>
<td>Medium</td>
</tr>
<tr>
<td>Energy consumption</td>
<td>High</td>
<td>Medium</td>
<td>Low</td>
<td>Medium</td>
</tr>
<tr>
<td>Protocol</td>
<td>Unknown</td>
<td>UAVTalk</td>
<td>mainly Mavlink</td>
<td>Paparazzi</td>
</tr>
<tr>
<td>Platform</td>
<td>Windows</td>
<td>Windows/Mac/Linux</td>
<td>Depends</td>
<td>Linux/Mac</td>
</tr>
</tbody>
</table>

Table 1.1: Autopilots for civil UAV applications overview.

A good objective to achieve would be to create compatibilities between the different autopilots and ground stations so the user can choose independently both parts. Creating competence between projects because once some user has bought the autopilot stills can change the ground station that is using to the most convenient one.

In the next chapter we are going to propose a design for the upgrade in Paparazzi and try as well to create some compatibility between many different UAV software.
CHAPTER 2. DESIGN

The proposed design consists of different parts. The ideas are to get rid of the limitations that the wireless communication has in Paparazzi, improve the performance and capability of Paparazzi protocol and create a minimum compatibility with Mavlink protocol. Therefore we open the compatibility to many different ground stations and autopilots.

2.1. Limitations

Currently Paparazzi is using a single XML file to define all the messages. It only has one class for each type of communication: uplink, downlink and ground (to communicate the different software parts). Since it is using a single byte as message ID we can have a maximum number of 256 ($2^8$) messages in each class. This limitation is starting to be reached in telemetry messages.

As we can see in the next figure the system takes this messages definition file and using a generator creates two C files. One for the uplink type class and another for the downlink type class. These created files contain the C macros that allow the airborne code to create the communication packets and extract the payload from them.

![Figure 2.1: Schema of current messages definition system.](image)

The new design will consist in different definition files, one for each class. New field in class will specify for which communication type is. And another byte in the packet will be to represent the class ID. That way we will have all the information to translate the message: which message is and from which class.

Instead of having a static message definition file, it will be generated with the desired classes. This selection will consist in another XML file where we will need to include all the desired classes with their name, their ID and the file name where are stored. The type of class will be specified in each class file. Also, instead of having only one C macro file for downlink class and another for uplink
class, Paparazzi will generate one for each class and at the end it will create two files: one including\textsuperscript{1} all downlink type classes files and the other for uplink type classes.

With this change we make almost transparent the modification for the rest of the system because it will be a common definition file and two C macro files like before. But they are dynamic now.

![Figure 2.2: Schema of proposed messages definition system.](image)

Since we were changing all the class handling we correct some things: old datalink class type is now called uplink and datalink is used for bidirectional messages. The class name datalink is now called common\_commands and common\_datalink is used for the main datalink type class. The class type ground is used to share messages between ground agents in a wired connection. Two new types of classes are added to the list:

- **Datalink type:** This class type is to place messages that can be used in uplink and downlink, or between different aircraft. For example TCAS\textsuperscript{2} messages. The aircraft that is sending the message handle it as downlink and the one receiving it handles the message as uplink. What we do with these classes is check the alignment\textsuperscript{3} of fields, as they are uplink type classes, but include their macros also in the downlink file.

\textsuperscript{1}The last two generated files will not have the code of the macros but only a \#include call to the macros files that are selected.

\textsuperscript{2}Traffic Collision Avoidance System: prevents the collision of aircraft in flight sharing position messages between them and, if necessary, computing and sending resolution messages.

\textsuperscript{3}To allow the micro-controller of the autopilot to extract correctly the fields of the packet they must be aligned in terms of micro-controller processing bits. e.g. for a 32 bits we have to sort the fields having in mind the variable length. No field can be split between 32 bits lines: \texttt{[uint8]+[int16]+[uint8]} or \texttt{[uint32]+[uint8]} is correct but \texttt{[uint8]+[int16]+[uint32]+[uint8]} is wrong because the \texttt{[uint32]} variable is between two 32 bit lines.
- **Airborne type**: This class type is to place messages that will be shared between different hardware components inside one aircraft. e.g. between the autopilot board and a camera pointing controller.

This is an example of some classes that we can find in a typical configuration:

<table>
<thead>
<tr>
<th>Class ID</th>
<th>Class file name</th>
<th>Class name</th>
<th>Class Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>downlink.xml</td>
<td>common telemetry</td>
<td>downlink</td>
</tr>
<tr>
<td>1</td>
<td>uplink.xml</td>
<td>common commands</td>
<td>uplink</td>
</tr>
<tr>
<td>2</td>
<td>datalink.xml</td>
<td>common datalink</td>
<td>datalink</td>
</tr>
<tr>
<td>3</td>
<td>custom_downlink.xml</td>
<td>custom telemetry</td>
<td>downlink</td>
</tr>
<tr>
<td>4</td>
<td>custom_uplink.xml</td>
<td>custom commands</td>
<td>uplink</td>
</tr>
<tr>
<td>5</td>
<td>custom_datalink.xml</td>
<td>custom datalink</td>
<td>datalink</td>
</tr>
<tr>
<td>6</td>
<td>ground.xml</td>
<td>ground</td>
<td>ground</td>
</tr>
<tr>
<td>7</td>
<td>alert.xml</td>
<td>alert</td>
<td>ground</td>
</tr>
<tr>
<td>8</td>
<td>sensors.xml</td>
<td>sensors</td>
<td>downlink</td>
</tr>
<tr>
<td>9</td>
<td>debug.xml</td>
<td>debug</td>
<td>downlink</td>
</tr>
</tbody>
</table>

Table 2.1: Example of message classes distribution.

Before we said that one byte of the packet will represent the class ID but as we don’t need 256 classes we decided to use 5 bits for the class ID (32 different IDs) and reserve the other 3 bits for future applications. Some ideas to use them are: a switch bit to tell if the message should be logged or a switch bit to tell if the packet contains a time-stamp in the header.

### 2.2. Paparazzi protocol

In the previous section we already added an extra byte in the packet anatomy of paparazzi protocol: the class ID byte including the 3 reserved bits. Now we are going to add another byte that will represent the packet number. Following a cyclic sequence from 1 to 255. That allow the program to calculate a ratio of lost packets in the communication channel. The number 0 is reserved to tell the system to not count the lost packets. For example when messages are broadcast from ground to many aircraft.

The final packet format will look like this:
The next thing to do is to improve the current message definitions. First of all we need to look for similar messages that can be merged to reduce the amount of messages. For example the message ENERGY will include the fields of the old ENERGY plus the different fields of BAT and MOTOR, without the fields that are not used any more. After that we will remove the originals.

Then we will see which messages would be useful to implement, asking to all the team for needs and suggestions. For example a MISSION_STATUS message that includes the current stage of the flight plan and different time counters. These fields were already in other messages but together with other useless fields for this purpose. As this new message is meant to be sent very frequently we need to put only the necessary fields.

Other messages to add are generic messages for people that is using Paparazzi without an aeronautic background. Those messages will have fields that already exist but with intuitive names. For example the message POSITION_SPEED_ACCEL. The name is telling us that we have fields for the position, for the speed and for the acceleration. The field names are very intuitive as well, for example pos_x, speed_y, accel_z, ref_lat...

After all the changes regarding message definitions the next step is to group them. Put the messages together depending of which component are related to or which kind of information are they giving. Groups like CAMERA, SPECIFIC SENSORS, GPS, ATTITUDE, MISSION, DEBUG...

Once they are grouped we need to select the most important and generic groups to put their messages in the main definition files. All the others will be spread in different files. For example all the messages of the groups GPS, ATTITUDE and MISSION will be in the main files. The messages of the DEBUG and SPECIFIC SENSORS groups will have their own XML definition file. The group of CAMERA and other groups that are not always used but they are generic will go in another file called for example CUSTOM_DOWNLINK for the downlink ones.

When all the messages are in their corresponding files we can redefine their IDs. Each
file will have a 256 limit and we have less messages per file now so we can put all in sequence again and leave space for new possible messages between groups. So in the main downlink file maybe we have the ATTITUDE messages going from ID 4 to ID 7, then we continue to the MISSION messages from 10 to 20 and so on. We will have a lot of space for new groups of messages and also space to add messages to the existing groups.

2.3. Mavlink compatibility

To create a Mavlink compatibility in Paparazzi we need to be able to send the basic common Mavlink messages with the same packet format of their protocol.

2.3.1. Data types

The first problem to solve is that Mavlink uses some variable types in the message fields that are not implemented in Paparazzi. First we will consider the basic types and then the array types.

- In Paparazzi we can use the following variable types: Integers and Unsigned Integers of size 8, 16 or 32 bits and floats\(^4\) (32 bits). In Mavlink some of the basic common messages are using also Integers and Unsigned Integers of size 64 bits and Chars\(^5\). So an implementation of these types must be done to be able to communicate with Mavlink ground stations or receive messages from autopilots using Mavlink protocol.

- Currently Paparazzi has only one possibility to put arrays in message fields: variable-length defined arrays that must be the last field of the message and only of the Paparazzi variable types. In the packets they are preceded by a byte that represents the length of the array. Therefore we have a limit of 256 positions\(^6\).

In Mavlink some of the messages are using arrays between other fields and some times of non Paparazzi variable types. But they are always length defined. In order to create the compatibility we need to allow the use of fixed length arrays and not only at the end of the message. We need to be able to use the new Basic types as well as the old ones. They will be implemented in some way to eliminate the need of sending a byte for the length, so we will not have the limit of positions and we will reduce the packet size for the same information.

\(^4\) Numerical rational values.

\(^5\) Character representation.

\(^6\) In fact we have a more restrictive limit. We only have one byte for the packet length so the whole packet can not exceed 256 bytes. But this is easy to change in the future if needed by adding another byte and if we can prepare the arrays thinking in the future it will be better.
2.3.2. Basic messages definition and implementation

Paparazzi is using its own generator to produce the macros for the messages. Therefore the format of the original Mavlink messages definition file is not recognized by Paparazzi. We need to redefine all the interesting messages in a Paparazzi format XML file. And also we need to sort the order of the fields by inverse size because the mavlink generator does that when generating the macros. That way we will have the same macros as them. Mavlink generator is using an extra value for computing the checksum\(^7\) that is function of the fields size and order. Since we need to redefine all the messages with the fields already sorted these numbers are hard-coded in Paparazzi. Every time we want to add or modify a Mavlink message we need to modify these numbers as well.

After defining all the Mavlink common messages we need to implement the important ones to be able to monitor the aircraft from one of the mavlink ground stations. The implementation consists, like for Paparazzi original messages, in defining a function that calls the macro to construct the packet, giving as parameter the values of the fields. So when we put this message in the telemetry file and compile our version of the autopilot software, it will find the function to send it.

2.3.3. Mavlink protocol module

For the implementation of the protocol itself we will use a module. The modules are parts of code than can be added from the airframe configuration file when we compile a specific autopilot software version for some aircraft. Only when we want to use this protocol we will include this module and disable all the other protocols by selecting DUMMY in the telemetry protocol option (airframe configuration file). If we don’t disable the other protocols, the autopilot will send some undesired messages that are not handled by the Mavlink ground stations. For example each time that the aircraft switch to another stage of the flight plan a message is sent automatically.

The implementation of the protocol in the module will not be difficult because with the changes in the Paparazzi packets they are close now to the Mavlink anatomy. The main difference it’s in the class ID byte. In Mavlink this field represents the component ID from where the message is coming from. For example the autopilot board will have one ID different to the GPS or camera control. But they still have the limitation of the 256 messages while we can create one class of 256 messages for each component. We can not use 256 component IDs because we decided to reserve 3 bits of the byte for future applications. All the reserved bits will be at zero when using Mavlink protocol so the number shown in Mavlink ground stations will correspond to the one sent. If some of the reserved bits are in 1, the total byte number would be different to the class ID.

\(^7\)The checksum is a piece of data included in the communication packet. Its value is computed in function of the other bytes of the packet and its purpose is to detect errors of transmission. The function to compute it is in both sides of the communication and if in the receiver the value received does not correspond to the one computed the packet is dropped. Assuming that some of the data has been modified during the transmission.
The other changes are the length byte that refers to only the field bytes and not all the packet, and the checksum that is computed differently.

2.4. Conclusions

We have seen all the limitations that we want to eliminate or at least reduce, the objectives in terms of performance and in terms of compatibility. We can notice how related are all the limitations. For example by improving the protocol we will be closer to the compatibility as well. And by modifying the possible data types for Mavlink we will be reducing limitations of the system.

Now that we know what we are going to change and why, it is time to define an implementation. We have three main changes highly correlated so the implementation process has to be thought carefully to reduce time but also enough segmented so we can test it progressively.
CHAPTER 3. IMPLEMENTATION

We are going to define the implementation process to achieve all the design specifications. This chapter does not follow the same structure as the Design. Due to that some implementation can be related to different design specifications we tried to follow the same order (Limitations, protocol, compatibility) but with more sections. Each section concerns to one independent implementation.

Thanks to a good design we know how to prevent future errors and save coding by preparing the code for the next changes. Procuring to not interfere in the correct functioning of the system to be able to test it by parts.

In the following sections we will refer to some paths of the project. Here is a schema of the main folders where the code involved in the implementation is located:

![Figure 3.1: Simplified folder tree of Paparazzi project.]

### 3.1. Pre-generator

The first implementation to do is adding a generator of the currently static XML file that has the communication messages definitions. This generator will be called from the Makefile\(^1\) when compiling the program.

We called that Pre-generator because it exists another generator, the one taking the com-

\(^1\)The makefiles are files used by the compiler to follow different processes. That allows to create dynamic compilations depending of configurations and the files that are changed to save time. By creating dependences we can avoid calling generators that will generate exactly the same existing file.
Upgrading Paparazzi autopilot Air-to-Ground Communication

communication messages definition file and producing the downlink and uplink C Macros files. The functions needed to build and read the communication packets. The figure 2.1 shows the generator process and the figure 2.2 shows how will work with the pre-generator plus generator process.

The communication messages definition file it's called messages.xml and it is located in the conf/ folder.

First we create a new folder called Messages inside the folder conf/, where are all the aircraft configuration files. In this folder we will put all the XML files containing the message classes with their messages. We take the classes of the static file messages.xml and we spread them in different files, one per class. As now will be more than one class of each type we need to add a parameter type to the class. If the generator works properly we will get a generated file mostly exactly as the current static one.

Then we need the configuration file that will be used to define which classes we want to include in the generated file. That will be a XML file with a specific structure that includes the class ID, name and source file:

```
<includes>
  <include file="downlink.xml" class_name="common telemetry" class_id="0"/>
  <include file="uplink.xml" class_name="common commands" class_id="1"/>
  <include file="ground.xml" class_name="ground" class_id="2"/>
</includes>
```

Source code 3.1: messages_conf.xml.example: Example file to determine which classes will be included in the compilation.

We call that file messages_conf.xml but we will create an example in another file called messages_conf.xml.example. If the system does not find the first one it will make a copy of the example. The code to do that with .example files is located in the Makefile, we only need to add the configuration file to the list:

```
conf: conf/conf.xml conf/control panel.xml conf/maps.xml conf/messages_conf.xml FORCE
conf/%.xml: conf/%.xml.example
   [ -L $@ ] || [ -f $@ ] || cp $< $@
```

Source code 3.2: Makefile: Piece of the main Makefile file.

The command conf is called when the program is compiled. This command calls the command conf/%.xml many times with the different file names that have an example file related. messages_conf.xml must be added to the list.

To develop the Pre-generator we will use OCaml. We create a new .ml file in the folder sw/tools/, where are all the compiling time generators. The name of the file will be gen_messages_xml.ml.

The process of the generator will be as follows:

1. Look into the configuration file messages_conf.xml and create a list of structures with all the parameters of the included classes (id, name and file).
2. For each structure, use the file source parameter to find the XML class structure file in /conf/messages/. From the XML code of each class, extract the type of class parameter and the XML children (messages with fields) and add them into the structure. At that moment it will check that names and ids are not repeated, and for classes with type different than `ground` also that the ids are inside the range (0 to 31).

3. Using the list of structures it creates the file `messages.xml`. This generated file will be like the original static one but with some differences: now the file has a version parameter with the value `2.0` and the classes have the new parameters `id` and `type`.

```
<protocol version="2.0">
  <class id="0" name="common telemetry" type="downlink">
    <message name="ALIVE" id="1">
      <field name="md5sum" type="uint8[]"/>
    </message>
    <message name="ATTITUDE" id="7">
      <field name="phi" type="float" unit="rad" alt_unit="deg"/>
      <field name="psi" type="float" unit="rad" alt_unit="deg"/>
      <field name="theta" type="float" unit="rad" alt_unit="deg"/>
    </message>
    <message name="ADC" id="8">
      <field name="mcu" type="uint8" values="FBW|AP"/>
      <field name="values" type="uint16[]" unit="none"/>
    </message>
  </class>
  <class id="3" name="ground" type="ground">
    <message name="NEW_AIRCRAFT" id="1">
      <field name="ac_id" type="string"/>
    </message>
    <message name="AIRCRAFT_DIE" id="2">
      <field name="ac_id" type="string"/>
    </message>
  </class>
  ...
</protocol>
```

Source code 3.3: messages.xml: Example of generated file.

The version value will be useful to know how to manage the file structure. Old format files can come from replays and logs so we need to keep the software in compatibility with the old versions.

4. Now, using the list of structures again, it will check each class type and depending of it, different processes will be followed:

- **downlink**: call the macros generator.
- **uplink**: check fields alignment and call macros generator.
- **datalink**: the same as uplink.
- **airborne**: it should do the same as uplink but as the implementation of the airborne alignment checking and macros generator was not implemented at the time those calls are commented.

---

2 The `ground` class type is a special type. We use Ivy to send messages between ground agents and with this protocol we will not send the class ID. Because instead of sending the message ID we are sending the message name. As all the messages names are different we don’t need the class identification to locate them so we can use as many classes as we want.
• **ground**: do nothing.

The checking of fields alignment is used only in messages that are going to be received by the autopilots. The microprocessors need to have aligned the values when processing the packets as we explained in the notes of the section Limitations in Chapter 2.

The macros are only needed when the packets are received through a modem. For **ground** communication we will send directly strings by the Ivy bus so we don’t need the Macros.

See Annex A to find part of the Pre-generator code.

The macros generator is already created, we only have to change for now the name of the file that they create making the process dynamic. This generator was called in the Makefile when the two final macros files were not found (`var/include/telemetry.h` and `var/include/datalink.h`). Now instead of being called directly we only add a command that we will call for each interesting class from the Pre-generator. The name of the macros files will follow the next rule:

```
messages_<class_name>.h.
```

Source code 3.4: Makefile: Command to call the macros generator with a specific target.

We need to add an automatic call to the `messages.xml` generator if the file does not exist yet or the configuration file `messages_conf.xml` has been changed. To do that we create a dependency with this file.

Source code 3.5: Makefile: Messages file generator call depending of configuration file.

5. Create the two final macros files to make it transparent to the implementation changes. They will only include the macros files of each type of communication (**uplink** and **downlink**), except for the **datalink** that will be included in both. See figure 2.2. Those files are renamed to make it more clear now that there is a new type called **datalink**. The names are `downlink_msg.h` and `uplink_msg.h`. We need to change all the references to these files in the software to use the new names.

To see an example of Macro file go to Annex B.

Source code 3.6: `downlink_msg.h`: Final downlink macros file.
After all these system changes we can spread the message definitions in different classes, grouping them and resorting their identifiers. Because we reviewed all the messages in the grouping process we have now a better idea of which of them can be merged, and useful messages that could be defined.

For all the modified or created messages to work properly we need to change some parts of code.

- The periodic downlink messages system in the airborne (sw/airborne/firmwares/fixedwing/ap_downlink.h for fixed-wing aircraft and sw/airborne/firmwares/rotorcraft/telemetry.h for rotor-craft). Here are defined the macros to send periodically the messages defined in the telemetry configuration file. There are specified which data variables correspond to which message field. They only call the generated macros with these specific values.

- The uplink messages handling system in the airborne (sw/airborne/firmwares/fixedwing/datalink.c for fixed-wing aircraft and sw/airborne/firmwares/rotorcraft/datalink.c for rotor-craft). Here the system identifies the message received and we can define actions for each different one. For example changing global variables with the field values or replying with another message.

- The ground aircraft Server (sw/ground_segment/tmtc/ac_server.ml). It receives the messages from the Link through the Ivy bus and takes actions for each different message. Before there were two different servers, one for fixed-wing aircraft (fw_server.ml) and another to rotor-craft (rotorcraft_server.ml). We decided to merge them in one single file due to that all the messages have different names so there is no need to have them separated. Also, there are messages that are common to both and are processed two times. Old servers are saved to handle old message versions in logs or replays. To choose which servers to use, new one or old ones, the new version parameter of the message definition file is used.

- The Ground Control Station (sw/ground_segment/cockpit/gcs.ml). It has the controls to send different messages from the ground. For example the message SETTING to change some configuration values of the autopilot.

- Other software files that are sending messages at some point of the process. Like sending some mission messages when the aircraft is changing the stage of the flight plan.

All these parts have to be modified according to the messages definition to allow the system to recognize and treat them properly. For the interested reader, some code pieces of the previous files can be found in Annex C.
3.2. Messages module

To manage the messages, all the ground agents have a common library. The file is called pprz.ml and is located in the OCaml libraries path: sw/lib/OCAML/. In this file we can find useful functions to convert fields, messages and classes into structures, put them in lists and find them with different methods. To do so there is an OCaml Module\(^4\) that we need to change to implement the new changes in the messages system.

Currently this module is loading all the messages of one class. That means that with the old system, when one ground agent needs to handle downlink messages for example, it will load this module with the parameter \texttt{class=telemetry}. When it needs to get the structure of one specific message it can do it searching by message name or ID.

But now we have different classes for each type so we need to adapt this module. We will keep the possibility to load the module with only one class because at some points we will not have the class ID to search messages. But they are specific cases where we know at the moment of loading which class is\(^5\). We will add the possibility to load the module specifying the class type instead of the name. That will load all the classes of the given type, meaning that we will need to give the class ID as parameter when asking for a message by ID.

To create that multi-choice system we use functors\(^6\). Two different functors will load the same module but with different values in some configuration variables.

<table>
<thead>
<tr>
<th>Functors</th>
<th>mode</th>
<th>sel_class_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>Messages(_\text{of_type}(\text{selection}=\text{class_type}))</td>
<td>Type</td>
<td>None</td>
</tr>
<tr>
<td>Messages(_\text{of_name}(\text{selection}=\text{class_name}))</td>
<td>Name</td>
<td>class_id_of_name(class_name)</td>
</tr>
</tbody>
</table>

Table 3.1: Functors and respective configurations for the module Messages returned.

In the Annex D there is part of the library interface. There we can see how the module and its functions look like after the changes.

The module, before loading the messages, will check the version of the messages.xml file. If it is a 2.0 version it can be treated normally with the new modified functions. But if it does not find any version means that is a 1.0 and a pre-process is needed. This pre-process will add to the XML structure loaded by the module the missing parameters as

\(^4\)OCaml Modules are structures. Groups of data variables, functions and exceptions. They are useful to encapsulate codes and prevent calling functions with similar names by error. There is a module example in the source-code of the pre-generator in the Annex A called MakeCalls. To call a function, value or represent an exception of one module we need to put the prefix ModuleName..\(^5\)For example the class alert is a special class to handle alerts between ground agents. There is only one and is of type ground. There are some software files that only uses this class but not the others so we know the name of class but we will not know the class ID when receiving the messages because in Ivy only the message name is sent as string.\(^6\)OCaml Functors are functions that return a module structure loaded by different configurations. For example we can set specific data variables to some different values depending of which functor are we using, but returning the same basic structure.
the version and the class types. We hard-coded the old class types due to that they were always the same: datalink for uplink type class, telemetry for downlink type class...

3.3. Protocol changes

To make the implementation of the previous section work, we need the class ID in the header of the communication packet. Without this byte we can not distinguish between two messages with the same ID. As we have said in the design we decided to add other information: a packet sequence number and reserve some of the bits in the class ID byte to future applications.

For that we need to change the different protocols implementation files, the general protocol implementation, the macros generator and the receiving system of the airborne.

First of all we have to add the macros that will put the class ID and packet sequence bytes in the different protocols. These files are located in `sw/airborne/subsystems/datalink/`.

---

```
#define PprzTransportPut_uint8(_dev, _byte) { 
    ck_a = _byte; 
    ck_b = ck_a; 
    PprzTransportPut1Byte(_dev, _byte); 
 }

#define PprzTransportPutPacketSequence(_dev) { 
    pprz_down_packet_seq++; 
    if (pprz_down_packet_seq==0) { pprz_down_packet_seq++; } 
    PprzTransportPut_uint8(_dev, pprz_down_packet_seq); 
 }

#define PprzTransportPut_class_uint8(_dev, _name, _byte) PprzTransportPut_uint8(_dev, _byte)

static inline void pprz_parse_payload(struct pprz_transport * t) { 
    uint8_t i; 
    for (i = 0; i < t->trans.payload_len; i++) 
        dl_buffer[i] = t->trans.payload[i];
    dl_msg_available = TRUE;
    if ((dl_buffer[0] == t->trans.packet_seq) && (dl_buffer[0] == 0)) { 
        if ((t->trans.packet_seq + 1) < dl_buffer[0]) { 
            uint8_t jump = dl_buffer[0] - (t->trans.packet_seq + 1); 
            //Do something like increment counter
        } else { 
            uint8_t jump = dl_buffer[0] + (255 - (t->trans.packet_seq)); 
            //Do something like increment counter
        }
    } 
    t->trans.packet_seq = dl_buffer[0];
}
```

---

7 The general protocol is used to create a transparency between protocols and macro calls. The generated macros are using the general one and then, depending of the selected protocol, these macros are linked to the specific protocol ones. E.g. the macro generated has a function DownlinkHeader(device,payload_length) and if we select the Paparazzi protocol it is linked with PprzTransportHeader(device,payload_length).
We also need to change the receiving function in the same file to count the packet sequence and do something if there is a jump in the sequence. For now we will not do anything in this case but it is prepared to work when needed.

We decided to count the packet sequence in each protocol and not in the global protocol because if the system is using two different protocols at once the sequence will be different for each one. If not the system would find jumps every time that a message is sent by the other protocol.

In the function that is extracting the payload we need to add the two new bytes of the header.

For counting the packet sequence and generating it on the ground we have to deal with the Link agent. This agent is the last step before going to the wireless communication. It links the wireless channel with the ground Ivy bus.

The code is located in `sw/ground_segment/tmtc/link.ml`.

```ml
(* Packet Sequence *)
let up_packet_sequences = Hashtbl.create 1
let down_packet_sequences = Hashtbl.create 1

let increase_sequence_loop = fun value ->
  let new_val = value + 1 in
  if new_val >= 256 then 1
  else new_val

let store_seq_jump = fun jump -> (* Do something, receiving down seq with jump *)
  jump

let sequence_jump = fun suposed actual -> (* Considering that the jump is less than 256 *)
  if suposed < actual then
    ignore (store_seq_jump (actual-suposed))
  else begin
    ignore (store_seq_jump (actual + (255-suposed)));
  end

let check_down_packet_sequence = fun ac_id seq ->
  if seq <> 0 then
    if (Hashtbl.mem down_packet_sequences ac_id = false) then Hashtbl.add down_packet_sequences
    ac_id 1;
    try
      let old = Hashtbl.find down_packet_sequences ac_id in
      if increase_sequence_loop old <> seq then sequence_jump (increase_sequence_loop old) seq;
      Hashtbl.replace down_packet_sequences ac_id seq with
      | NotFound -> failwith ( sprintf "Aircraft Id not found in packet sequence hash table (ID=%d)" ac_id )

let get_up_packet_sequence = fun ac_id ->
  if (Hashtbl.mem up_packet_sequences ac_id = false) then Hashtbl.add up_packet_sequences ac_id 1;
  try
    let old = Hashtbl.find up_packet_sequences ac_id in
    let current = increase_sequence_loop old in
    Hashtbl.replace up_packet_sequences ac_id current with
    | NotFound -> failwith ( sprintf "Aircraft Id not found in packet sequence hash table (ID=%d)" ac_id )
```

...
On the ground we have to count and generate not only one packet sequence but as many as aircraft we are flying. The system will count the sequences depending on the aircraft ID byte in the packet and store them in data tables. To generate them, the same thing.

For messages broadcast to every aircraft without an aircraft ID in the fields the packet sequence will be zero. As we mentioned in the design the number zero is used to ignore the packet sequence.

In the general protocol file we need to change the macro `DownlinkStartMessage`. This macro creates the packet header and message header by using the specific protocol macros to put the corresponding bytes. It is the first protocol macro used by all the message constructors in the generated macros. We have to put the new packet sequence and class ID macros calls in the correct place. See figure 2.3.

```
#define DownlinkStartMessage(_trans, _dev, _classname, class_id, _name, msg_id, payload_len) {
  downlink_nb_msgs++;
  Transport(_trans, Header(_dev, DownlinkIDSize(_trans, _dev, payload_len)));
  Transport(_trans, PutPacketSequence(_dev));
  Transport(_trans, PutAcId(_dev));
  Transport(_trans, PutClassUint8(_dev, _classname, class_id));
  Transport(_trans, PutNamedUint8(_dev, _name, msg_id));
}
```

Source code 3.10: downlink.h: general protocol implementation file.

For the macros generator we have to change the message fields alignment checking. With the two new bytes the alignment has to be different. Before we were starting to count from 16 bits due to the old configuration of two message header bytes\(^8\): the sender ID and the message ID. With the new configuration the packets have two more bytes in the message header so we need to start counting from 32 bits, that is the end of one line so we start counting from 0.

Now the Messages Module in the ground library can have all the header bytes that is expecting.

### 3.4. Data types

To implement the new data type for the message fields we need to change some of the previous files again.

The new types are `char`, `int64`, `uint64` and the arrays of those.

First we need to add the new types into the ground library so the Messages Module can

\(^8\)The first two bytes shown in the figure 1.19 are extracted by the modem before arriving to the system software. The same for the last two bytes corresponding to the checksum.
extract the values of these kind of fields.
See Annex E to find some of the lines changed.

Every time that an array is handled we need to add the option of being a fixed length array. The distinction between an array and basic data is made by a function that checks the format of the field type parameter in the xml file. If the type is ended by \[\] means that is a variable length array. For example int8[]. Now we need another function to distinguish between basic data, variable length array and fixed length array. In this last case a match with the format type[length] is made. If there is some integer between the two brackets that means that is the length of a fixed length array.

The ground library is coded in OCaml. In this language there is a special module to manage 64 bits length data. It is called Int64. The problem with this module is that the maximum value is the signed integer value ($2^{63}$), saving one bit for the sign. Even if we are using unsigned integer and we do not need to specify the sign. Having this language limitation in mind we will explain it in the program documentation. We checked if the module used for 32 bits length data has the same limitation and it has. So for data bigger than $2^{31}$ as unsigned variable we must use the 64 bits length variables.

For the airborne receiving system, coded in C, the necessary libraries are already included. We only have to include the types from these libraries and create analogous functions to the ones already existing.

The handling of fixed length arrays by the C macros is really easy. When printing the macros by the generator we specify directly the length and we remove the parameter of the function. All the process is the same.

The function putting the array is different because now it does not put the preceding length byte. For the same reason we do not have the macro to access to this byte and the packet
is shorter.

Now is time to implement the protocol macros to put the fields, included the general protocol. Functions like the one that we were talking before, for the fixed length arrays: DownlinkPutUint8FixedArray(). And for basic data like DownlinkPutChar(). The two examples are of the general protocol but we need to implement the PprzTransport, XbeeTransport, and IvyTransport too.

To end we have to put the new types in the macros generator so it could count properly the bytes and extract correctly the message payload. In the generator we need to add the new types to the fields alignment checking, to be able to count the new types and to distinguish between variable and fixed length arrays. The first one has a preceding length byte but the second one has not.

### 3.5. Mavlink compatibility

After all the previous changes we are now in a better position to reach the Mavlink protocol compatibility. We have a similar packet anatomy and the possibility of using the same data types, including fixed length arrays in any position.

To do so and not interfere in the code when people does not want to use it we will use a Paparazzi module. As we explained in the design the Paparazzi modules are part of code that we can enable from the airframe configuration file. The modules have for example codes for specific sensors that are not always used. The modules consist in a configuration file, located in conf/modules/, and the implementation files, located in sw/airborne/modules/module_folder.

First of all we create the configuration file for the module. The name for the module will be Mavlink so we create a file called mavlink.xml.

```
<module name="mavlink">
  <header>
    <file name="mavlink.h"/>
  </header>
  <periodic fun="PeriodicSendMavlink (MavlinkTransport, DefaultDevice)" freq="60"/>
  <event fun="MavlinkDataLinkEvent()"/>
  <makefile>
    <file name="mavlink.c"/>
    <file name="mavlink_transport.c"/>
  </makefile>
</module>
```


In this module configuration file we specify a periodic calling to the function that will send the periodic downlink messages. And an event to handle the receiving packets. Exactly like the other protocols work but only when the module is included.

Then we need to create the folder for the implementation files of the module. In this folder we will put the protocol itself. This protocol is in fact a copy of the just modified ones. Like for example PprzTransport. Changing the prefix of the macros for Mavlink.

The biggest change is with the packet length and checksum computation:
- The packet length in the Mavlink case only takes into account the length of the fields of the message. For the Paparazzi protocol was counting also the message header.

- The checksum computation takes into account the message length in Mavlink. So we need to change the macro `PutHeader()` to use the macro that is adding the byte to the checksum count first and then calling the macro to put the byte into the packet. For that we need to define this macro before the one to put the header.

![Source code 3.14](pprz_transport.h: Macro to put the packet header in Paparazzi protocol)

```
... 
#define PprzTransportHeader(dev, payload_len) { 
  PprzTransportPutByte(dev, STX);
  uint8_t msg_len = PprzTransportSizeOf(dev, payload_len);
  PprzTransportPut1Byte(dev, msg_len);
  ch_a = msg_len; ch_b = msg_len;
} 
#define PprzTransportPutUint8(dev, _byte) { 
  ch_a += _byte;
  ch_b += ch_a;
  PprzTransportPut1Byte(dev, _byte);
}
... 
```

Source code 3.14: pprz_transport.h: Macro to put the packet header in Paparazzi protocol.

![Source code 3.15](mavlink_transport.h: Macro to put the packet header in Mavlink protocol)

```
... 
#define MavlinkTransportPutUint8(dev, _byte) { 
  MavlinkTransportPutByte(dev, _byte);
  crc_accumulate(_byte, checksum);
} 
#define MavlinkTransportHeader(dev, payload_len) { 
  MavlinkTransportPut1Byte(dev, STXMAV);
  uint8_t msg_len = MavlinkTransportSizeOf(dev, payload_len);
  crc_init(checksum);
  MavlinkTransportPutUint8(dev, msg_len);
} 
... 
```

Source code 3.15: mavlink_transport.h: Macro to put the packet header in Mavlink protocol.

The last version of Mavlink (3.0) is using a final byte to compute the checksum. This byte is different for each message and it depends of the fields and their order. As we explained in the design we can not compute these values because we need to have the fields already sorted in the definition file. So the values are hard-coded in the protocol file. If we want to add or change the Mavlink messages we have to define the message and its value for the checksum.

```
uint8_t crc_extra[256]={50, 124, 137, 0, 237, 217, 104, 119, 0, 0, 0, 89, 0, 0, 0, 0, 0, 0, 0, 0, 0, 214, 159, 220, 168, 24, 23, 170, 144, 67, 115, 39, 246, 185, 104, 237, 244, 222, 212, 9, 254, 230, 28, 28, 132, 221, 232, 11, 153, 41, 39, 214, 223, 141, 33, 15, 3, 100, 24, 239, 238, 30, 200, 183, 0, 130, 0, 148, 21, 52, 124, 0, 0, 0, 20, 0, 152, 143, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
```
Source code 3.16: mavlink_transport.c: Array to extract the last checksum value with the message ID.

In the module folder we will put the periodic downlink messages macros too. We will implement only some of them, the most important ones, with the values that are already used by some paparazzi messages. That will allow a basic compatibility with the Mavlink ground stations.


Source code 3.18: mavlink_downlink.h: Example of a periodic Attitude message macro in Mavlink.

The files with the events that are specified in the module configuration file are called mavlink.h and mavlink.c and they are analogous to the datalink.h and datalink.c that are used for the regular communication. In the telemetry configuration files, the periodic messages are divided in two independent processes: Ap and Fbw. These processes are handled by the datalink files. We will define a new process called Mavlink in which we will put the periodic messages to send of the Mavlink protocol. When using the Mavlink module we have to add the Mavlink process in the telemetry file but we still need the Ap and Fbw processes, even if they are empty.

Now we can enable the Mavlink protocol by including the Mavlink module. But we need to be able to disable the other protocols to prevent sending some messages that shouldn’t be sent to Mavlink ground stations. To do so we will create another protocol called Dummy. This protocol will do nothing when the macros are called. With this method, when this protocol is selected in the airframe file, the system will think that is sending the messages but they will not be sent by the modem. So the system will not show any error. If we have leaved some messages in the Ap or Fbw processes they will not be sent either.

It is time to define the Mavlink messages in one class, called mavlink of course. The definition of the messages is a little bit different in both systems. First we need to remember of sorting the fields in our definition. Then remove some special type names and replace them for the corresponding regular one. For example in Mavlink exists the data
type `mavlink_version` that in fact is like an `uint8`.

Another thing that we can add is the name correspondence to some values\(^9\). In Paparazzi this is made in the same field definition with the parameter `values`.

```xml
<message name="PPRZ_MODE" id="85">
  <field name="ap_mode" type="uint8" values="MANUAL|AUTO1|AUTO2|HOME|NOGPS|FAILSAFE"/>
  <field name="ap_gaz" type="uint8" values="MANUAL|AUTO_THROTTLE|AUTO_CLIMB|AUTO_ALT"/>
  <field name="ap_lateral" type="uint8" values="MANUAL|ROLL_RATE|ROLL|COURSE"/>
  <field name="kill_auto_throttle" type="uint8" unit="bool"/>
</message>
```

Source code 3.19: Paparazzi message with names corresponding to values of field.

In Mavlink the names are defined at the beginning of the messages definition file or even in an external file. So for the most important ones we will find this names and put them in the `values` parameter.

The last part of the compatibility process is to create an example of autopilot configuration using the Mavlink protocol to be able to test it.

The steps to do it are as follows:

1. Include the Mavlink messages class in the file `conf/messages.conf.xml`.

2. Create a telemetry configuration file with some of the most important messages to send periodically. (conf/telemetry/mavlink.xml)

   ```xml
   <telemetry>
   <process name="Ap"> <mode name="default"/>
   <process name="Mavlink">
     <mode name="default">
       <message name="HEARTBEAT" period="0.2"/>
       <message name="SYSTEM_TIME" period="0.1"/>
       <message name="SYS_STATUS" period="1"/>
       <message name="MAV_ATTITUDE" period="0.2"/>
       <message name="VFR_HUD" period="0.1"/>
       <message name="GLOBAL_POSITION_INT" period="0.3"/>
       <message name="LOCAL_POSITION_NED" period="0.3"/>
       <message name="LOCAL_POSITION_SETPOINT" period="1"/>
     </mode>
     <mode name="minimal">
       <message name="HEARTBEAT" period="5"/>
       <message name="SYSTEM_TIME" period="1"/>
       <message name="SYS_STATUS" period="0.5"/>
       <message name="MAV_ATTITUDE" period="0.5"/>
       <message name="VFR_HUD" period="0.5"/>
     </mode>
     <mode name="extremal">
       <message name="HEARTBEAT" period="5"/>
     </mode>
   </process>
   </telemetry>
```


3. Create an airframe configuration file to include the mavlink module and disable the other protocols. (conf/airframes/ENAC/fixed-wing/weasel_mavlink.xml)

   ```xml
   ...
   ```

\(^9\)The names are used to replace the numeric value of the field when it is represented in the ground station. Useful when we want to use different options represented by the identifier of the option.
With these three steps we obtain an autopilot configuration ready to use Mavlink as communication protocol and compatible with the Mavlink ground station software.

### 3.6. Conclusions

After all the implementation process we have changed how the system is generating the macros to manage communication packets and how the communication messages are defined. We changed, removed and created new message definitions. Modified the anatomy of the communication packet. Increased the number of data types available for fields. And finally we created a new compatibility with another protocol. All of that keeping the compatibility for old versions of logs and replays of the old system.

Now we have all the implementation done and we can run the latest tests. Of course we did not wait until now to test the implementation, some previous tests were made. This is what we are going to see in the next chapter.
CHAPTER 4. EVALUATION

In this chapter we are going to explain the different tests done and their results. They will be sequentially explained like if they were done after the implementation but the tests are being executed between implementation processes. Only the tests for the Mavlink compatibility were dependant of all the implementation and that is why they were the last ones.

The first test to run is to check the correct functioning of the new messages definition system with all the generators working together. Recreating bad situations to test that the system is recognizing them and stopping the process raising the correct error message. After that it is going to be tested the ground library messages module together with the new protocol definition. Once it is demonstrated that is working, the test with new data types will be performed. Then, the test with all the previous things plus the Mavlink compatibility will be performed.

As the last point, and to be sure that the other parts of the software keep working well with the new changes, a flight test will be done.

4.1. New message system

To test the new messages system, that allow us to use more than 256 messages per communication type, we need two parts already implemented. First we will test that the generation of the old static messages definition file is working. As well as the dynamic macro files and the final macro files including the previous ones.

Then, after implementing the changes in the ground library and in the protocol packet anatomy, we will test that we are receiving the messages correctly in uplink and downlink communications.

4.1.1. Pre-generator

The Pre-generator test will consist in creating some messages class definition files of different class types. Inside the classes there will be the already defined messages. So if all works correctly we will obtain a generated file similar to the old static one. We include all these classes in the `conf/messages_conf.xml` file. We have to obtain the same macros that before the changes but now spread in one file per class. At the end two files have to be created, including all the datalink and uplink/downlink macros.

We run the test an all is correct. We obtain the expected files with the corresponding code. Now is time to test the errors:

- The alignment has to be checked only for datalink and uplink classes. We can force errors of alignment in one of the classes of each type at once and see what happens. For downlink and ground classes nothing happen, but for datalink and uplink classes an error is shown and the pre-generator stops before producing the XML file.
• The class ID should not exceed 31 for non ground classes. Like in the previous step we can try to use IDs bigger than 31 for one class of each type at once. Except for the ground class, an error is shown and the generator stops before producing the file.

• The class IDs can not be repeated. We can try to repeat an ID and we see that an error is shown, stopping the process.

• No repeated message IDs are allowed inside the same class. To test it we can define two times the same message and see how the program stops with the corresponding error.

4.1.2. Messages module and protocol changes

To test the complete new system after changing the Messages Module and adding the new bytes in the protocol packet, we are going to send messages of different classes. We can not see the messages received by the aircraft during the test so this is the process that we will follow: We will send two messages TEST and TEST2 stored in two different uplink classes. Then we will implement a check and response procedure in the airborne messages handler. So when a TEST message is received, a downlink ACK_TEST will be replied. And for the second one a ACK_TEST2. Trying first only one, then the other and, after that, both at once we can confirm that the system is recognizing the different classes.
To send the test messages from the ground we will need a simple C program. We will call the file containing the code as c_manual_packets_sender.c. It will be located in sw/ground_segment/tmtc/. We have to add a special key in the make file to compile it.

```c
#include <glib.h>
#include <stdio.h>
#include <stdlib.h>
#include <ivy/ivy.h>
#include <ivy/ivyglibloop.h>

gboolean timeout_callback(gpointer data) {
    IvySendMsg(“ME TEST1”);
    IvySendMsg(“ME TEST2 1”);
    return TRUE;
}

int main(int argc, char** argv) {
    GMainLoop *ml = g_main_loop_new(NULL, FALSE);
    IvyInit(“Test”, “Test READY”, NULL, NULL, NULL, NULL);
    IvyStart(“127.255.255.255”);
    g_timeout_add(100, timeout_callback, NULL);
    g_main_loop_run(ml);
    return 0;
}
```

Source code 4.1: c_manual_packets_sender.c: Testing C program to manually send messages.

We need to execute the program in an independent terminal while Paparazzi software is running and connected to the autopilot. In the Messages window we should see how the autopilot is replying with the corresponding messages.

![Message 1](image1)

![Message 2](image2)

Figure 4.5: Receiving test message in class 1.

Figure 4.6: Receiving test message in class 2.

As we can see sending and receiving messages from different classes is not a problem.

Now we have to check that the sent values of the packet sequence are correct. To do it we will run the Link agent in debug mode. In this mode it is showing in the terminal all the packet bytes in hexadecimal format. He have to look to the third byte of some consecutive messages and check that the value is increasing between the range 1-255.
4.2. Data types

To test the possibility of using the new data types in all the communication directions we will send some test messages. We will use the same program as the previous section to manually send the messages that will be replied, but now we are going to add some fields with the new data types. For example some 64 bits length variables and fixed length char arrays. In the airborne the values will be copied to the downlink messages so we should receive the same that we are sending. We will put the array in the middle to confirm that it works in any place, not only at the end.

```c
#include <glib.h>
#include <stdio.h>
#include <stdlib.h>
#include <ivy/ivy.h>
#include <ivy/ivyglibloop.h>

gboolean timeout_callback (gpointer data) {
    unsigned long long i1 = 123456789012345678ULL;
    signed long long i3 = -123456789012345678LL;
    char i2[4] = {'x','a','v','i'};
    IvySendMsg("ME TEST %llu %c,%c,%c,%c %lld 1", i1, i2[0], i2[1], i2[2], i2[3], i3);
    return TRUE;
}

int main ( int argc, char** argv) {
    GMainLoop *ml = g_main_loop_new(NULL, FALSE);
    IvyInit ("Test", "Test READY", NULL, NULL, NULL, NULL);
    IvyStart("127.255.255.255");
    g_timeout_add(100, timeout_callback, NULL);
    g_main_loop_run(ml);
    return 0;
}
```

Source code 4.2: `c_manual_packets_sender.c`: Testing C program to manually send messages.

In the next figure we can see how all the values are received correctly.
4.3. Mavlink compatibility

To test the Mavlink compatibility we are going to use the example created in the implementation. With those airframe and telemetry configuration files we will compile an autopilot version only compatible with Mavlink ground stations. For the test we will use QGroundControl.

With this ground station first we need to check that it is receiving the packets. We have a communication terminal to do so.

First we need to connect the link modem to the ground station. After plug-in it we click the button **Connect Link**, we configure the basic parameters of the modem and we click **Connect**.

Just after the connection the communication console will star to show byte values. Those are the packets arriving to the software. That means that at least the communication between the two modems is working.
Figure 4.10: QGC Communication console.

If QGroundControl recognize the packets\(^1\) it will decode them and we could see them in the message inspector. Here the software shows all the messages with their receiving frequency. We can expand each message to see the values of the fields.

Figure 4.11: QGC Message inspector.

If the basic messages are received and decoded correctly our UAV will be recognized. There is a little window where basic information of the Aircraft is shown: Type of aircraft, throttle, battery level, speed, position...

Figure 4.12: QGC Aircraft overview.

Down of the previous window there is another one with some bars to indicate levels of battery, CPU usage...

Figure 4.13: QGC Aircraft status information.

\(^1\)We had problems with the recognition of the packets even though they were arriving to the communication console. The problem was that the documentation of Mavlink was incorrect about the packet anatomy. Specifically the checksum bytes that were in the opposite order as they tell. We were asked to correct it as the documentation is open to everyone, so we did it and now it is correctly explained.
The next test after seeing that the aircraft is recognized is to see if the artificial horizon and compass are moving according to the attitude of the aircraft. In the following figures we can see two of the infinite positions and movements tested:

![Figure 4.14: QGC Artificial horizon and compass. Position 1.](image1)

![Figure 4.15: QGC Artificial horizon and compass. Position 2.](image2)

Compass and artificial horizon are working properly. The position is working as well but is not shown here because inside the lab there is no GPS coverage so the values are wrong. But if we hard-code some values in the Position messages these values are shown in the different position indicators of QGroundControl.

If we disconnect the autopilot, interrupting the communication with the software, after a while the aircraft overview window starts flashing with red color. That is simulating a communication lost.

![Figure 4.16: QGC Alert when HEARTBEAT message is not received for a while.](image3)

The figures of this Mavlink compatibility test have been extracted from a video filmed to show to the Paparazzi developers around the world the new compatibility. This video can be found on-line [11].
4.4. Flight test

A flight test was performed using an aircraft equipped with the Paparazzi autopilot. The board was flashed with the modified version of the software. And some typical flight plans were followed to ensure the reliability of the upgraded system.

After some minutes flying without problems we can accept the upgrade as airworthy.

![Flight test](image.jpg)

Figure 4.17: Flight test performed in Muret flight field, France.

4.5. Conclusions

As it has being demonstrated in the previous tests, Paparazzi is now totally compatible with Mavlink and basically compatible with the Mavlink ground stations. Now is up to the users to define and implement the messages that they need. To improve the interoperability with the Mavlink ground stations like QGroundControl.

The functionality of the new code has being demonstrated as well, even in a real flight condition using all the systems at one. And also the correct response of the system to possible user errors.

Now it is time to demand to the Paparazzi project managers the integration of the modifications in the newest software version to be released. And create a documentation page in the Paparazzi website explaining the new design and how to use it [18].
CHAPTER 5. ENVIRONMENTAL IMPACT STUDY

This project has been focused in the software part of the Paparazzi project. Therefore the environmental impact is minimal. But in terms of the overall Paparazzi project, as it involves hardware as well, we can define some interesting environmental aspects.

First of all there is the manufacturing of the PCB boards. These boards are ordered to the company EuroCircuits. Regarding their website the manufacturing process involves chemical reactions between Cooper and Alkaline solutions. This kind of reactions releases harmful gases and the alkaline solutions tend to be irritant. In the drilling process some glass fibre and resin residues are produced. As they are small amounts they are thrown to the regular garbage. Also they use X-rays and Lasers which need a considerable amount of energy to work [19].

Once the boards are received, many electrical components and plastic connectors are used. These components are usually built using really dangerous processes and following weak regulations of some Asian countries. To solder them on the boards a special paste is used. This paste is a mix of Tin (Sn), Silver (Ag) and Lead (Pb) mainly. From these ones, the last one is the most contaminant. There are some new mixes, more expensive, without lead. Currently the team is using both types. Neither of them can be thrown to the regular garbage. Thanks to the high and stable temperature of some special ovens this product becomes solid and conducts the electricity to the components and connectors. This process needs a big amount of electrical energy for the oven.

For the manufacturing of the aircraft many different materials are used. From polymer foams to carbon, Kevlar® (aramid) and glass fibres. These fibres can now, or in a near future, be recycled by new processes [20][22][21]. The resin to joint the matrix is called Epoxy and is synthetic.

Figure 5.1: Blender quad-copter made with a sandwich of carbon fibre layers between Airex foam.  
Figure 5.2: New generation of Blender using only carbon fibre.

Blender is a design of Michel Gorraz.
The most commonly used foams are called Elapor®, Rohacell®, Airex® and Expanded polypropylene (EPP). Some of them are recyclable [23][24], but others are not recyclable [25] and the combustion of them is very pollutant. So the best practice is to reuse as much as possible. Parts of broken aircraft are used to create small pieces to hold the hardware and the battery on place.

![Figure 5.3: Reused EPP pieces to hold the battery in the correct place.](image)

The aircraft are powered by electric motors what means that there is no consumption of fossil fuel therefore no pollutants are exhaust to the air.

Finally the last thing to be concerned are the different batteries used. For the aircraft are LiPo¹ batteries, for the safety transmitters are NiMH² batteries and for the ground stations a car battery is used. All these batteries have to be treated correctly after their life cycle.

We could talk about the microwaves emitted by the transmitters but they have a low emission power and there is no comparison with the other frequencies used for example for the television or the 3G.

On the other hand Paparazzi project has help to the research of some science and technology careful with the environment. Following there are some of this green projects:

---

¹Lithium-ion polymer: high current batteries.
²Nickel Metal Hydride: low current batteries.
5.1. Solar Storm

The Solar Storm is an hybrid solar powered micro air vehicle. It was build by one of the PhD in the team, Murat Bronz. It consists in a small glass fibre unmanned aircraft equipped with solar cells in the wings which are capable of supplying around 40% of the total power consumption. It is equipped also with a Maximum Power Point Tracker that maximizes the extracted power from solar cells. This project has help and keeps helping to the research of the most light, resistance and optimized solar cells currently in the market. For the interested reader the Solar Storm poster can be found in the Annex F.

5.2. FLOHOF 2007

FLOHOP is a scientific campaign, it means Flow over and around Hofsjökull. Hofsjökull is a glacier located in central Iceland. The campaign consisted in monitoring the basic meteorological parameters with a high temporal and spatial resolution. Thanks to meteorological stations and unmanned aircraft. The main mission of the aircraft was to take
aerial photos and determine profiles of temperature, humidity and wind [26]. Those aircraft were equipped with Paparazzi autopilots.

5.3. Météo-France cooperation

![Figure 5.6: Paparazzi aircraft equipped with Météo-France sensors.](image)

Météo-France is a big structure for meteorology monitoring and prediction. It is located in Toulouse and works in cooperation with the École Nationale de la Météorologie (ENM). It is one of the main forecasts in France. The cooperation with Paparazzi ENAC team started some years ago thanks to conferences about civil applications for UAVs. On June 2012 a cooperation flight was performed. It consisted in a Paparazzi unmanned aircraft equipped with some Météo-France sensors and a flight plan designed by Paparazzi team according to Météo-France requirements. The objective of the study was to determine the best protection for the sensors of temperature, pressure and humidity on board an UAV. Protection against wind and solar rays. With this information future measurements will be more accurate. On September 2012 a formal relation will start between both institutions thanks to a Government grant for cooperation programs.

5.4. Conclusions

Some of the bad impacts to the environment of the Paparazzi project have been reviewed. Although there is always a try to reduce this impact with re-utilization and using new green products it can not be completely eliminated. On the other hand, projects helping to the environment are performed thanks to this project. And some of the future green technology has being inspired by this project as well. The idea is to get as much knowledge about the impact as possible to reduce it before the massive use of UAVs for civil applications starts. Smaller, lighter, more efficient and even hybrid solar powered aircraft are the starts of this idea.
CHAPTER 6. CONCLUSIONS

We can conclude that the wireless link communication system of Paparazzi is now more efficient. With less restrictions and more clean than before. And also more compatible with other ground station software.

In terms of efficiency we implemented the fixed length arrays. They do not need the preceding byte with the length information. That means less bytes to send per packet with the same information.

Regarding the reduction of limitations we can talk about the new data types for message fields. The fixed length arrays that can be located in any position of the message. And the possibility of having 256 messages x 32 classes for non ground type communication and as many classes as we want of the ground type. When before we only had 256 messages per class and one class per type of communication.

The project is more clean now with the class system. We can split the messages by purpose or component in different files with representative names. Enable and disable classes easily. Before we were working with one single file containing all the messages of all the classes. Now there are 5 class types with clear names, not only 3 with confusing names and different uses. And we can change to third-party communication protocol only by changing a couple of lines in the configuration files. That means that the new code will not interact with the rest of the system when is not needed.

We joint together the codes that were different for fixed-wing aircraft and rotor-craft and there were no need of that. Reducing the code and the CPU usage. To order a little bit more the project we merge similar communication messages and resort them by groups. Even creating some basic and easily understandable ones for users without aeronautical background.

In terms of compatibility, the new possibilities make the software more prepared for future compatibility works. It will be more easy to disable functions than implementing them, and now we have things like the class ID and the packet sequence. Used for many protocols. We have seen also the fully compatibility with the Mavlink protocol and the basic compatibility with its ground station software. Opening the doors for non expert users to implement the rest of messages when needed only by coping the format of the already implemented ones.

To finish with the conclusion we can talk about how this upgrade was thinking in the future. For example reserving the three bits of the packet for future applications, that if needed they can be added to the class ID and increase the 32 current classes up to 256. Also the airborne class type, not fully implemented yet but meant to be useful when the hardware on-board will be more complex. For example for bigger aircraft with many different and separated systems. And the non limitation of length for the fixed length arrays. Allowing in the future to increase the packet length and not being restricted by this limitation.

Many changes that all together make Paparazzi better in some ways and full-fit some of the most important demands that the team had.
GLOSSARY

3D Three-Dimensional, 8
A/C Aircraft, 17
API Application Programming Interface, 6
Bit Binary digit, 13
CPU Central Processing Unit, 4
drone Unmanned Air Vehicle, 5
ENAC Ecole Nationale de l’Aviation Civile (Toulouse, France), 5
GPS Global Positioning System, 3
ID Identifier, 8
IMU Inertial Measurement Unit, 3
INRIA Institut National de Recherche en Informatique et en Automatique, 4
IR Infra-red, 3
LGPL Lesser General Public License, 18
MacOS Macintosh Operating System, 4
OCaml Objective Categorical Abstract Machine Language, 5
QGC QGroundControl, 35
SW Software, 17
TCAS Traffic Collision Avoidance System, 12
UAV Unmanned Air Vehicles, 5
USB Universal Serial Bus, 3
Wine Wine Is Not an Emulator, 8
XML Extensible Markup Language, 5


(++) MAIN MODULE: DOES ALL THE PROCEDURE TO GENERATE THE messages.xml FROM THE messages.conf.xml INCLUDES +

module SpreadMessages = struct

(++) Generates a XML file called messages.xml with all the included classes in messages.conf.xml and their messages +

let generate_messages_xml = fun () ->
  try
    (** Get list of included files +)
    let includes = includes.get_includes () in
    (** Get list of included classes as XML Elements +)
    let classes = List.map Classes.get_class_from_include includes in
    (** Create the complete XML element +)
    let final_xml = Xml.Element("protocol","version","2.0").classes in
    (** User-readable xml formatting +)
    let formatted_xml = Xml.to_string fmt final_xml in
    (** Save the xml code into a xml file +)
    SaveXml.save_formatted_xml;
    (** Prepare and return data for the next function (List of class_ids, class_names and class_types +)
    let class_ids_names_types = List.map Classes.extract_name_type classes in
    class_ids_names_types
    with
    | Includes.Invalid_include_structure exc -> failwith (sprintf "Invalid <include> structure (Exception: %s)" exc)
    | Includes.Duplicated_class_id li -> failwith (sprintf "Duplicated class id (%s) at includes file "% li)
    | Classes.Class_id_out_of_range i -> failwith (sprintf "Class id (%d) out of range [0->31] at includes file. Only ground type classes can exceed 31" i)
    | Includes.Duplicated_class_name n -> failwith (sprintf "Duplicated class name (%s) at includes file "% n)
    | Classes.Invalid_class_structure exc -> failwith (sprintf "Invalid <class> structure (Exception: %s)" exc)
    | Classes.Invalid_class_type typ -> failwith (sprintf "Invalid class type: %s" typ)
    | Classes.XML_parsing_error (file.msg.pos) -> failwith (sprintf "Error parsing XML file: %s (Error: %s Line: %s)" file.msg)
    | SaveXml.Can_not_open_file (file.exc) -> failwith (sprintf "Can not open the file to generate (Exception: %s) file exc)
    | SaveXml.Can_not_move_file err -> failwith (sprintf "Can not move the generated file to the final destination (Error code for command MV: %s)" err)
    | Classes.Invalid_class_xml_node exc -> failwith (sprintf "Invalid <class> xml node in generated file (Exception: %s)" exc)
    | e -> failwith (sprintf "Unhandled exception raised: %s" (Printexc.to_string e))
  end

module MakeCalls = struct

    let make_target = "gen_messages_macros"
    let make_options = ""

    let make = fun class_name class_id check_alignment ->
      let file = Env.paparazzi_home // "Makefile" in
      let macros_target = var.include_path // ("messages","\$(String.lowercase class\_name)\".h") in
      let c = sprintf "make -f %s MACROS\_TARGET=%s MACROS\_CLASS=%s MACROS\_CLASS\_ID=%s MACROS\_ALIGN=%s u %s %s" file macros_target class_name class_id check_alignment make_options make_target in
      let returned_code = Sys.command c in
      if returned_code <> 0 then failwith (sprintf "Make command error (Error code: %d)" returned_code)

    let generate_macros = fun classes ->
      List.map (fun clas -> match clas.g_type with
        | "datalog" -> prerr_endline ("\t DATALOG Class -> Generate macros ("clus.g_name") [}
Check Alignment]: make class.g_name class.g_id 1
| "uplink" -> prerr_endline ("\t Uplink Class -> Generate macros ("class.g_name") [Check Alignment]"); make class.g_name class.g_id 1
| "downlink" -> prerr_endline ("\t Downlink Class -> Generate macros ("class.g_name")"); make class.g_name class.g_id 0
| "ground" -> prerr_endline ("\t Ground Class -> Do nothing ("class.g_name")")
| "airborne" -> prerr_endline ("\t Airborne Class -> Generate macros ("class.g_name") FIXME")
| t -> failwith (printf "Invalid class type in generated file: %s")
) classes;
end
...

{(******************************************************************

Main ******************************************************************)

let () =
  if Array.length Sys.argv <> 7 then
    failwith (printf "Usage: %s <messages config file> <spread messages path> <generated file> <
      var include path>", Sys.argv.(0));
  let classes_info = SpreadMessages.generate_messages_xml () in
  prerr_endline ("---------- GENERATING MESSAGES MACROS (Spread) ----------");
  ignore (MakeCalls.generate_macros classes_info);
  prerr_endline ("---------- GENERATING MESSAGES MACROS (Global) ----------");
  ignore (FinalMacros.generate_files classes_info);
  prerr_endline ("----------");

Source code A.1: gen_messages_xml.ml: part of the pre-generator to obtain messages.xml.
APPENDIX B. MACROS FILE EXAMPLE

Source code B.1: messages_common_telemetry.h: common_telemetry class macros file.
APPENDIX C. MESSAGES HANDLING CODES

Source code C.1: ap_downlink.h: periodic downlink messages implementation for fixed-wing aircraft.

Source code C.2: datalink.c: received uplink messages handling for fixed-wing aircraft.
match msg.Pprz.name with
| "GPS\_INT" ->
  a.unix_time <- LL.unix_time_of_tow (truncate (fvalue "tow" / 1000.));
  a.itow <- Int32.of_float (fvalue "tow");
  a.gps_Pacc <- fvalue "pacc"
| "ESTIMATOR" ->
  a.alt <- fvalue "z";
  a.climb <- fvalue "z\_dot"
| "DESIRED" ->
  ("Trying to be compatible with old logs ... ")
  begin match a.nav_ref with
  Some nav_ref ->
    let x = (try fvalue "x" with _ -> fvalue "desired_x")
    and y = (try fvalue "y" with _ -> fvalue "desired_y") in
    a.desired_pos <- Aircraft.add_pos_to_nav_ref nav_ref (x, y);
    | None -> ()
  end;
  a.desired_altitude <- (try fvalue "altitude" with _ -> fvalue "desired_altitude");
  a.desired_climb <- (try fvalue "climb" with _ -> fvalue "desired_climb");
  begin try a.desired_course <- norm_course (fvalue "course") with _ -> () end
| "NAVIGATION\_REF" ->
  a.nav_ref <- Some (Utm { utm.x = fvalue "utm\_east"; utm.y = fvalue "utm\_north"; utm.zone =
  fvalue "utm\_zone" })
...

Source code C.3: ac_server.ml: received downlink messages handling on ground.
APPENDIX D. LIBRARY INTERFACE PART

```scala
... module type CLASS_NAME = sig
  val class_name : string
end

module type CLASS_TYPE = sig
  val class_type : string
end

type messages_mode = Type | Name

module type CLASS(Xml = sig
  val xml : Xml.xml
  val selection : string
  val mode : messages_mode
  val sel_class_id : int option
end
type msg_and_class_id = {
  msg_id : int;
  cls_id : int;
}

module type MESSAGES = sig
  val xml_version : string
  val formatted_xml : Xml.xml
  val messages : (msg_and_class_id, message) HashTbl.t
  val message_of_name : string -> message_id -> message
  val class_id_of_msg : message_name -> class_id
    (** class_id_of_msg message name -> class_id returns the class id containing the given message *)
  val class_id_of_msg_args : string -> class_id
    (** class_id_of_msg_args class_id name -> class_id returns the class id containing the given message when args ..(0) is the parameter *)
  val class_id_of_msg_args_unsorted : string -> class_id
    (** class_id_of_msg_args_unsorted class_id name -> class_id returns the class id containing the given message when string with semicolons is the parameter *)
  val values_of_payload : Serial.payload -> packet_seq * sender.id * class.id * message_id * values
    (** values_of_payload payload returns the A/C id, class id, message id and the list of (field_name, value) *)
    -> Serial.payload
    (** payload_of_values gen_packet_seq sender_id class_id id vs returns a payload *)
  val values_of_string : string -> message_id * values
    (** values_of_string string -> message_id returns a payload *)
  val values_of_string_unsorted : string -> message_id * values
    (** values_of_string_unsorted string -> message_id returns a payload *)
  val string_of_message : ?sep : string -> message -> values -> string
    (** string_of_message message ?sep msg values Default [sep] is space *)
  val message_send : ?timestamp : float -> string -> string -> values -> unit
    (** message_send send msg name values returns a payload *)
  val message_bind : ?sender : string -> string -> (string -> values -> unit) -> Ivy.binding
    (** message_bind sender msg name callback returns a payload *)
  val message_answerer : string -> string -> (string -> values -> unit) -> Ivy.binding
    (** message_answerer sender msg name callback returns a payload *)
```

Set a handler for a
[message_req] (which will send a [msg_name].REQ message).
[callback asker args] must return the list of attributes of the answer. »

val message_req : string -> string -> values -> (string -> values -> unit) -> unit
(* [message_req sender msg_name values receiver] Sends a request on the Ivy
bus for the specified message. A [msg_name].REQ message is send and a
[msg_name] message is expected for the reply. On reception, [receiver]
will be applied on [sender_name] and attribute values of the values. »
end

module Messages_of_type : functor (Class : CLASS_TYPE) -> MESSAGES
module Messages_of_name : functor (Class : CLASS_NAME) -> MESSAGES
module MessagesOfXml : functor (Class : CLASS.Xml) -> MESSAGES

Source code D.1: pprz.ml: part of the interface of the library pprz.ml.
type Type =
   | Scalar of string
   | ArrayType of string
   | FixedArrayType of string * int

let types = [
  ("uint8", { format = "%w"; glib_type = "guint8"; inttype = "uint8_t"; size = 1; value=Int 42 });
  ("uint16", { format = "%w"; glib_type = "guint16"; inttype = "uint16_t"; size = 2; value=Int 42 });
  ("uint32", { format = "%d"; glib_type = "guint32"; inttype = "uint32_t"; size = 4; value=Int 42 });
  ("uint64", { format = "%ld"; glib_type = "guint64"; inttype = "uint64_t"; size = 8; value=Int 42 });
  ("int8", { format = "%d"; glib_type = "gint8"; inttype = "int8_t"; size = 1; value=Int 42 });
  ("int16", { format = "%d"; glib_type = "gint16"; inttype = "int16_t"; size = 2; value=Int 42 });
  ("int32", { format = "%d"; glib_type = "gint32"; inttype = "int32_t"; size = 4; value=Int 42 });
  ("int64", { format = "%ld"; glib_type = "gint64"; inttype = "int64_t"; size = 8; value=Int 42 });
  ("float", { format = "%f"; glib_type = "gfloat"; inttype = "float"; size = 4; value=Float 4.2 });
  ("double", { format = "%f"; glib_type = "gdouble"; inttype = "double"; size = 8; value=Float 4.2 });
  ("char", { format = "%c"; glib_type = "gchar"; inttype = "char"; size = 1; value=Char 'x'});
  ("string", { format = "%s"; glib_type = "gchar*"; inttype = "char*"; size = max_int; value=String "42" })
]

let is_fixed_array_type = fun s ->
let type_parts = Str.full_split (Str.regexp "[]\]" ) s in
match type_parts with
  | [Str.Text ty; Str.Delim '['; Str.Text len; Str.Delim ']'] -> true
  | _ -> false

let type_of_fixed_array_type = fun s ->
try
  let type_parts = Str.full_split (Str.regexp "[]\]" ) s in
  match type_parts with
  | [Str.Text ty; Str.Delim '['; Str.Text len; Str.Delim ']'] -> begin ignore( int_of_string (len)); ty end
  | _ -> failwith ("Pprz.type_of_fixed_array_type is not a fixed array type")
with
  | Failure str -> failwith (sprintf "Pprz.type_of_fixed_array_type: length is not an integer")

let length_of_fixed_array_type = fun s ->
try
  let type_parts = Str.full_split (Str.regexp "[]\]" ) s in
  match type_parts with
  | [Str.Text ty; Str.Delim '['; Str.Text len; Str.Delim ']'] -> begin ignore( int_of_string (len)); len end
  | _ -> failwith ("Pprz.type_of_fixed_array_type is not a fixed array type")
with
  | Failure str -> failwith (sprintf "Pprz.type_of_fixed_array_type: length is not an integer")

let rec value = fun t v ->
match t with
  | Scalar ("uint8" | "uint16" | "int8" | "int16") -> Int (int_of_string v)
let pipe_rexexp = Str.regexp "["]
let field_of_xml = fun xml ->
  let tt = Ext.Xml.attrib xml "type" in
  let t = if is_array_type tt then ArrayType (type_of_array_type tt) else if is_fixed_array_type tt then FixedArrayType (type_of_fixed_array_type tt, int_of_string (length_of_fixed_array_type tt)) else Scalar tt in
  let format = try Xml.attrib xml "format" with _ ->
    default_format
  in
  let auc = alt_unit_coef_of_xml xml in
  let values = try Str.split pipe_rexexp (Xml.attrib xml "values") with _ -> [] in
  (String.lowercase (Ext.Xml.attrib xml "name"),
   (type = t; format = f; alt_unit_coef = auc; enum_values ))

let rec value_of_bin = fun buffer index _type ->
  match _type with
  | Scalar ("uint8") -> Int (Char.code.buffer.[index]).sizeof _type
  | Scalar "char" -> Char (buffer.[index]).sizeof _type
  | Scalar ("int8") -> Int (int8_of_bytes.buffer_index).sizeof _type
  | Scalar ("int16") -> Int (int16_of_bytes.buffer_index).sizeof _type
  | Scalar ("int64") -> Int64 (int64_of_bytes.buffer_index).sizeof _type
  | ArrayType t ->
    (** First get the number of values *)
    let n = int8_of_bytes.buffer_index in
    let type_of_elt = Scalar t in
    let size = sizeof type_of_elt in
    let size = 1 + n * s in
    (Array (Array.init n (fun i -> fst (value_of_bin buffer (index+i+s) type_of_elt))), size)
  | FixedArrayType (t, i) ->
    (** First get the number of values *)
    let n = 1 in
    let type_of_elt = Scalar t in
    let size = sizeof type_of_elt in
    let size = 0 + n * s in
    (Array (Array.init n (fun i -> fst (value_of_bin buffer (index+i+s) type_of_elt))), size)
  | Scalar "string" ->
    let n = Char.code.buffer.[index] in
    (String (String.sub buffer (index+1) n). (1+n))
  | _ -> failwith "value_of_bin"

...
APPENDIX F. SOLAR STORM POSTER

A HYBRID SOLAR POWERED MICRO AIR VEHICLE

Solar Storm is the first hybrid solar-powered unmanned micro air vehicle in the world which has the autonomy navigation capability. The prototyped prototype shows the feasibility of endurance enhancement on such small vehicles by supplying 40% of total power consumption by solar energy.

Figure F.1: Solar Storm poster.
In this Annex the construction process of a Paparazzi UAV will be described. The following information can help to the understanding of how a drone works and which are the main subsystems used. The aircraft used is a commercial scale model aircraft so the process will not concern the manufacturing of it but only the assembly with all the hardware needed.

After the assembling of the autopilot board and the main subsystems, a first test is performed. This test has the objective of testing the hardware interconnection between components and boards. The modules to test are: the autopilot itself, the GPS, the datalink modems and the USB connection to flash the autopilot. A configuration of the autopilot is needed to be able to connect properly to the modules and treat their data.

With all these components interconnected it is possible to start the ground station and receive messages from the autopilot like if it was the complete UAV. The main messages to check are the IMU sensors information and the GPS data.

Once the main hardware has been tested, it is time to locate the different parts inside the airframe. It is important to have in mind the possible interferences between components. In this case the motor, the GPS, the datalink modem and the safety receiver are located as far as possible between each other. The USB has to be accessible easily when the aircraft is completed, to be able to re-flash it. The autopilot has to be removable to solve some possible problems or plug-in new subsystems. But at the same time has to be strongly fixed when flying to maintain at a correct level the IMU precision. For that, a foam box surrounding the board has been designed. It also helps to put it as close as possible to the centre of gravity of the aircraft to have more accurate values form the accelerometers.
Having all the subsystems in place, with all the servos and the motor connected, another configuration has to be done to calibrate them. Then, connecting the safety receiver and with the help of a regular radio-control transmitter, the manual controls are tested. Checking that all the directions of the servos are right and all the channels correspond to the correct servo output. After that and testing again all the system it is possible to do the final assembly of the aircraft.

Now all the system is tested again and recalibrated. The only thing left is the final calibration of the autopilot control values that has to be done during a real flight. Following
the aircraft in the ground station it is possible to correct deviations of trajectory and other issues by modifying the gain of the control loops until the aircraft is flying smoothly but responding fast.

![Figure G.5: Non calibrated control loops.](image1)

![Figure G.6: Calibrated control loops.](image2)

The UAV is now ready to perform correctly any flight plan that we design. Some of the components that can be added for specific missions are a surveillance camera and a dropping system to release a load in any specific place.