Design of an autopilot communications software based on open source.

Master thesis

| TITLE: | Design of an autopilot communications software based on open source |
| AUTHOR: | Guillem Jornet i Nasarre |
| DIRECTOR: | Gautier Hattenberger (ENAC)  
Juan López Rubio (UPC) |
| DATE: | 14th February 2014 |
Design of an autopilot communications software based on open source.
ABSTRACT

Paparazzi is a free and open-source hardware and software project encompassing an exceptionally powerful and versatile autopilot system for fixed wing aircrafts as well as multicopters. The project includes both ground station and airframe solution.

Due to initial constraints, onboard communications code (as well as other subsystems) had been done in a way that resulted in a very optimized program but lead some limitations as static configurations and non-optimal data treatment. Those constraints affected both communication architecture (concept level) and communication code (programming level).

The goal of this project has been the design of a new communication system for the airframe by developing a new architecture using code solutions that provide dynamic configurations. In addition, this new design introduces priorities on messages transmission while taking into account the difficulties of the prioritized data management in an embedded system.
Design of an autopilot communications software based on open source.
I want to dedicate special agreements to Gautier Hattenberger for bringing me the opportunity of working in this project and joining ENAC’s Drone Laboratory.

I also want to express my gratitude to Juan Lopez Rubio for accepting to be my director from UPC. Without him, this would had not been possible.

Special regards to all people from ENAC’s Drone Laboratory: Catherine Ronflé, Gautier Hattenberger, Michel Gorraz, Alex Bustico, Xavier Paris, Didier Melin, Christophe Dumont, Jean-Phillipe Condomines, Murat Bronz and Savas Sen.

And there are no words enough for Marisa...
Design of an autopilot communications software based on open source.
INDEX

CHAPTER 0. INTRODUCTION .......................................................................................... 12

CHAPTER 1. EXISTING COMMUNICATIONS ARCHITECTURE ............................. 14
  1.1. General communications architecture ......................................................... 14
  1.2. Previous airframe communications solution .............................................. 16
       1.2.1. Downlink software architecture .......................................................... 16
       1.2.2. Uplink software architecture ............................................................... 18
  1.3. Limitations ................................................................................................. 18
       1.3.1. Static behaviour ................................................................................... 18
       1.3.2. Macro usage ......................................................................................... 19
       1.3.3. Memory management .......................................................................... 22
       1.3.4. Source code migration ......................................................................... 22
  1.4. Motivation and Objectives of the project .................................................... 23

CHAPTER 2. DESIGN OF A NEW SOFTWARE COMMUNICATIONS ARCHITECTURE ............................................................................... 25
  2.1. Message layer ............................................................................................. 27
  2.2. Datalink layer ............................................................................................. 28
  2.3. Transport layer ........................................................................................... 29
  2.4. Generic driver ............................................................................................ 30

CHAPTER 3. TECHNICAL IMPLEMENTATION ......................................................... 32
  3.1. Downlink .................................................................................................... 32
  3.2. Message ....................................................................................................... 34
       3.2.1. Code generators ................................................................................... 35
       3.2.2. Data packaging .................................................................................... 35
       3.2.3. Transmission management .................................................................. 36
  3.3. Datalink layer ............................................................................................. 37
  3.4. Transport layer ........................................................................................... 38
  3.5. Generic driver ............................................................................................ 40
  3.6. Priority management ................................................................................... 41
       3.6.1. Transaction based transmission ........................................................... 42
       3.6.2. Transmission queue ............................................................................. 43
  3.7. Dynamic Memory Space ............................................................................ 47
Design of an autopilot communications software based on open source.

FIGURES

Figure 1: General Paparazzi communications architecture ........................................ 14
Figure 2: Paparazzi communications with detailed airframe architecture. .............. 15
Figure 3: Existing downlink software architecture .................................................. 16
Figure 4: Final frame of a Paparazzi message ...................................................... 17
Figure 5: Existing uplink software architecture .................................................... 18
Figure 6: Extract of a possible telemetry makefile establishing static options .......... 19
Figure 7: Specific vs. generic functions .................................................................. 19
Figure 8: Static code (unchangeable options) avoiding macros shadowing code ...... 20
Figure 9: Communication software structure (based in macros) ............................. 21
Figure 10: New Paparazzi communications software architecture ....................... 25
Figure 11: Message layer logical procedure ......................................................... 27
Figure 12: Tx & Rx transport layer concept ........................................................... 29
Figure 13: Different protocol packets ...................................................................... 30
Figure 14: Communications input/output interface design ..................................... 31
Figure 15: Software procedure for downlink transport selection ........................... 33
Figure 16: Software procedure for downlink device selection ............................... 33
Figure 17: Datalink callback registration ............................................................... 38
Figure 18: New Rx transport design ....................................................................... 39
Figure 19: New Tx transport design ....................................................................... 39
Figure 20: Generic driver definition (device.h) .................................................. 40
Figure 21: From generic to specific driver (UART case) ....................................... 41
Figure 22: Empty transmission queue ................................................................... 43
Figure 23: Transmission queue with a single element ......................................... 44
Figure 24: Transmit queue with two elements (Higher priority first) ................. 44
Figure 25: Transmission queue with three elements (Higher priority first) ............ 45
Figure 26: Transmission queue with four elements ............................................. 45
Figure 27: Traces of transmission priority management ......................................... 46
Figure 28: DMS structure and parameters ............................................................ 47
Figure 29: DMS with two slots occupied ............................................................. 48
Figure 30: DMS with a third slot introduced ....................................................... 49
Figure 31: DMS with a fourth slot introduced ...................................................... 50
Figure 33: Paparazzi system overview ................................................................. 56
Design of an autopilot communications software based on open source.

Figure 34: Communications between the Paparazzi agents. ........................................... 57
Figure 35: Airframe modules. .......................................................................................... 58
Figure 36: Up side of Apogee v1.0.0 ............................................................................. 63
Figure 37: Down side of Apogee v1.0.0 .......................................................................... 63
Figure 38: Xbee RF Modem with a whip antenna. ............................................................. 64
Figure 39: Xbee RF Modem with SMA connector for a bigger antenna ......................... 64
Figure 40: Xbee RF Modem for ground station (100mW and high sensitivity). ............... 64
Figure 41: Transport and driver selection from makefile options ..................................... 65
Figure 42: GPS message definition .................................................................................. 66
Figure 43: Actual GPS transmission function ................................................................. 67
Figure 44: Actual GPS data pack function ........................................................................ 68
Figure 45: Data Packaging / unpackaging functions for ATTITUDE message ............. 69
Figure 46: Data packaging / unpackaging functions for DOWNLINK message ............. 70
Figure 47: Existing GPS transmission macro ........................................................................ 71
Design of an autopilot communications software based on open source.
CHAPTER 0. INTRODUCTION

UAS are rising quickly at many levels: application, technology, features, architecture. Specially civil market developments are becoming more and more present in many fields and, among other civil projects, it must be noticed Paparazzi project. Paparazzi is a free and open-source hardware and software project encompassing an exceptionally powerful and versatile autopilot system for fixed wing aircrafts as well as multicopters. The project includes both ground station and airframe solution that are connected using a bidirectional RF wireless link that transmits data encapsulated in messages. Those messages have been specifically designed for Paparazzi project.

As it has been said before, Paparazzi is a project for an entire UAS management. That is, not only the set of messages (Paparazzi protocol) is defined but also all data treatment. This projects talks about enhancing the communications of software architecture of Paparazzi for more clear, comprehensible and versatile communications capabilities.

A first chapter does an analysis of the existing communications software architecture, trying to identify the characteristics and the limitations. The last point from this chapter exposes the objectives for communications enhancement. This proposal tries to identify things that can be done in a more efficient and/or convenient way and new things that are not still introduced but they are considered to be necessary for future capabilities.

The second chapter explains, from a concept level, the new software design and its benefits from the point of view of software architecture design.

Chapter three shows the technical implementation. The objective of this chapter is to explain how things are done in order to match the desired design. This chapter exposes the ideas behind the technical solution and skips the code; for a detailed code analysis, reader can refer to ANNEX B.

Finally, before conclusions, chapter four highlights the tasks that would continue this work.
Design of an autopilot communications software based on open source.
CHAPTER 1. EXISTING COMMUNICATIONS ARCHITECTURE

This chapter introduces the general software communications architecture with particular emphasis on the airframe. Then after, an accurate analysis will show the limitations of the actual solution and the final point will be the proposal of the new solution.

1.1. General communications architecture

As it has already been told, Paparazzi is a full system including both airframe and Ground Control Station (GCS). For further information, please refer to ANNEX A where the topology of the system is described.

Next figure illustrates the communications structure of Paparazzi where it can be noticed three data types:

- **Telemetry**: Data from the airplane to the ground. This type of data contains information about the UAV status.
- **Datalink**: Data from the ground to the UAV. This data contains requests and orders.
- **Ground**: Data shared between different ground agents.

![Diagram of Paparazzi communications architecture](image)

**Figure 1**: General Paparazzi communications architecture
Design of an autopilot communications software based on open source.

Figure 1 shows the elements exchanging data, their relation and the data types they manage. However, it doesn’t show how this data is treated by each element and how it is integrated with other Paparazzi subsystems. Since the interest of this study is the redesign of airframe communications, Figure 2 shows the general communications architecture for the airframe part.

Data is sent and received by a RF modem which is connected to the autopilot board (more information about autopilot boards can be found in ANNEX A). The autopilot software has, for each available connector, a specific driver.

When transmitting downwards (telemetry data), autopilot gives the message to be sent to ‘downlink’ that encapsulates data using the appropriate protocol. Once data is encapsulated, it is passed to the driver connected with the modem.

When transmitting upwards (datalink data), data coming from the driver is expanded by ‘datalink’ using a given protocol (protocols at this level are used for data integrity verification). Just in case data is successfully expanded, it is passed to a parsing function that will treat the information arrived.
1.2. Previous airframe communications solution

As it can be seen in ANNEX A, the first autopilot boards were made with simple processors. The first processor used was an Atmel AVR 8-bits, which had not DSP instructions neither other optimisations. Without optimisations and a just 20MHz clock, autopilot software had to be very optimized.

The first thing to do in order to have a simple and efficient software is to design a simple software architecture. As simple is the architecture, as simple the code will be. Following that rule, the initial solution for Paparazzi communications was a code that had four levels in cascade.

1.2.1. Downlink software architecture

Figure 3: Existing downlink software architecture

Figure 3 shows the downlink (telemetry) software architecture. The first level, message, builds a message according to its definition and passes it to downlink. The first thing downlink does is to add both airframe and message identification numbers to the message. Downlink also knows which are the transport and the proper device for telemetry. Once the two identification numbers are added, downlink passes data to transport layer with a reference to the driver. Transport encapsulates data and finally passes it to driver layer.
The files matching this architecture are:

- **Messages:**
  - messages.xml: This file is not actually a code file. It contains the message definitions. An automatic code generator (a brief exposition is described in section 3.2.1) takes it in order to create the file messages.h.
  - messages.h: This file, created from messages.xml, contains the message sending procedures. For each message, it passes the appropriate data (in the appropriate order) to downlink.h.

- **Downlink:**
  - downlink.h: downlink.h adds both UAV and message identification numbers to data and passes it to the proper transport with a reference to the transmit device.

- **Transport:**
  - pprz_transport.c/.h: It does the Paparazzi encapsulation and passes the final frame (message plus all added data) to the driver. Paparazzi encapsulation has to parts. First part is a header that has two elements, transport identification number (0x99) and data length in bytes (data length includes transport data and is 1 byte long). Second part is a tail containing two checksum elements for data integrity verification.
  - xbee.c/.h: XBee transport is very similar to Paparazzi transport. It also has a header with a transport identification number (0x7e), data length (2 bytes long) and a specific xbee configuration. As in Paparazzi transport, second part is a tail but with just one checksum number.

- **Driver:**
  - uart.c/.h: It contains a generic UART driver.
  - uart_arch.c/.h: Every processor has a specific UART treatment. Thus, for each processor there is a uart_arch.c/.h that supports the generic uart.c/.h.

![Figure 4: Final frame of a Paparazzi message](image)
1.2.2. Uplink software architecture

![Uplink software architecture diagram]

Figure 5: Existing uplink software architecture.

As it can be seen, the files matching this architecture are the same than in previous section 1.2.1 with the exception of `datalink`.

- **Datalink**: 
  - `datalink.c/.h`: this file is the bridge between the arrived messages and the autopilot service destination.

1.3. Limitations

1.3.1. Static behaviour

As exposed at the beginning of this chapter, due to the usage of simple processors the airframe code had to be simple and efficient. The point behind this is that the code needs not to have configuration dependencies. That is, every option has to be applied by the pre-processor. In that way, every step in code is just an execution that needs no more values than the transmitted data. Thus, the first limitation is the static behaviour.

Every subsystem in Paparazzi is compiled using a makefile. The makefile is a file (with extension `*.makefile`) that contains a list of files to be compiled and the pre-processor directives (the options) to be applied in the compilation process. The usage of makefiles for each subsystem is very convenient. In that way, the user can easily manage different subsystems configurations.
Design of an autopilot communications software based on open source.

Figure 6 shows some lines that could appear in a possible ‘telemetry.makefile’. In this example it can be seen the directives that enables the telemetry and defines UART0 as the telemetry device.

```
#telemetry.makefile
...
#--- DOWNLINK OPTIONS---
-telemetry_CFLAGS += -DDOWNLINK
-telemetry_CFLAGS += -DDOWNLINK_DEVICE=UART0
...
```

Telemetry_cflags contains the telemetry pre-processor directives.

-DDOWNLINK stands for “define DOWNLINK”. This enables telemetry communications.

This action establishes UART0 as the device used for telemetry. ‘DOWNLINK_DEVICE’ will be substituted in code by UART0 and then code will be compiled. UART0 will be a static option.

Figure 6: Extract of a possible telemetry makefile establishing static options

1.3.2. Macro usage

In order to develop a pre-processor dependant autopilot, code was done using macros. Macros are pre-processor directives that are very useful for compilation options management. For instance, in Figure 6 telemetry is enabled by the definition of “-DDOWNLINK”.

However, even if it is strongly recommended not to use macros shadowing code lines, macros were not used only in makefiles but also in communications code. Let's explain the reason.

It has been told that a optimal static code would be made by commands or functions only managing user data but not system structures (because system structures would be static). That is

```
UART0Transmit(_data) {...}
UART1Transmit(_data) {...}
```

Instead of

```
Transmit(&uart0, data) {...}
```

Specific functions for each device. No device data has to be exchanged.

Generic function. Device data has to be exchanged.

Figure 7: Specific vs. generic functions
Design of an autopilot communications software based on open source.

Functions exposed in Figure 7 could be found at driver layer (i.e. in \texttt{uart.h}). If the upper layer, transport, managed the driver to use, it could be possible to find something like

\begin{verbatim}
#ifdef UART0
  static inline void PprzTransportPut1Byte (uint8_t data) {
    UART0Transmit(data);
  }
#endif

#elif defined UART1
  static inline void PprzTransportPut1Byte (uint8_t data) {
    UART1Transmit(data);
  }
#endif
\end{verbatim}

Figure 8: Static code (unchangeable options) avoiding macros shadowing code.

Figure 8 could be a possible solution for transport layer (i.e. \texttt{pprz_transport.h}) avoiding the usage of macros shadowing macros but it presents a development and maintenance problem: this example solves 1 action, transmit, over 2 drivers in 1 transport file but what would happen with a larger number of functions, drivers and transports?

Development process would become a hard task with the risk of multiple errors (a good code for UART0, for instance, would no ensure a good operation for UART1). More than that, a modification in a driver would become a big number of modifications over other files, what enlarges the list of possible errors created.

In addition, it must be noticed that this example talks about driver-transport interaction but the same issue could happen at other levels, thus a global code strategy is required.

The solution used was a kind of Front-End solution. The configuration options are taken by a layer (downlink) that manages them. This layer passes the options to the layer bellow (transport), which takes the ones it needs and keeps passing the other options to the layer bellow (driver). In that way, there is just one front for all the options (downlink) and one end for each one (in the case exposed, the driver). The options are transparent for the elements in the middle (transport in this case).

The only way to pass the driver option from downlink layer to driver layer without using parameters is to do it in the pre-processor. This is, using macros.
Design of an autopilot communications software based on open source.

Thus, finally telemetry was made of macros shadowing transport macros... shadowing driver macros... shadowing driver functions.

```
#define DownlinkPutUint8(_trans, _dev, _x) ...

... if '_trans' is PprzTransport...
#define PprzTransportPutUint8(_dev, _x) \     
   ck_a += _x; \ 
   ck_b += ck_a; \  
   PprzTransportPut1Byte(_dev, _x);
#define PprzTransportPut1Byte(_dev, _x) ...

... if '_dev' is UART0...
#define UART0Transmit(_x) ...
```

![Diagram of communication software structure](image)

**Figure 9**: Communication software structure (based in macros).

Up to now, it has been exposed the reason for the use of macros. However, macros usage in this code presents two big problems.

The first one is the code shadowing. Macros are manipulated by pre-processor and there is not a good pre-processor / processor interaction. This means that when the processor finds an error in a code that came from a macro, the only advise it can give to user is a reference to the top macro.

Code shadowing is a problem when a macro shadows multiple macros that contain code. When there is an error under the top macro, developer has to check the multiple macros there are inside (more than that, in Paparazzi those multiple macros contain multiple macros, too!).

The second problem is that code syntax depends on macro definitions. Usually, macros are used in order to enable or disable parts of code but they are not actually code lines. However, in the Paparazzi solution, function names are build using macro tokens. This can be a big problem when different people work on the same project. Nobody wants to modify other people’s tokens because this extremely affects the code. Thus, in order to avoid problems, people creates their own tokens with possibly the same meaning than others... Repeating things creates confusion.
1.3.3. Memory management

As it has been said at the beginning of 1.2, Paparazzi was initially programmed for 8-bit processors. Thus, data was treated in bytes and the code was not prepared for the usage of utilities able to manage bigger data (as DMA or memcopies).

Another existing problem was the unavailability of Dynamic Memory Allocation (malloc / calloc functions). Even if this is still unavailable now, some tasks would benefit.

1.3.4. Source code migration

Finally, another idea is rising. Airframe subsystems become more and more complex every day and they become identities interacting in many different ways. If this tendency is confirmed, a migration to object oriented codes will be required.

The initial code, based on macros, made difficult this jump to object oriented codes. It has not been the purpose of this thesis to change this but to create a new communications code in a way that will make this migration possible in future.
1.4. Motivation and Objectives of the project

The whole Paparazzi system is becoming more and more complex every day and, among others, communications need to be revised in order to fulfil the new requirements. For instance, multi-UAV architecture will possibly require to have data moving around across different UAV. So, the amount of data passing through a UAV and the complexity of subsystems will require a versatile communications software able to overpass the existing limitations.

Since the actual Paparazzi processors are much more powerful, it is not mandatory to have a static and fast execution oriented program. Thus, a revision can be done in order to solve the limitations exposed in 1.3.

In addition, new features are detected as important characteristics that communications software needs to achieve.

Apart from the limitations already explained, there is a new consideration that this thesis wants to take care. A problem that was not considered in previous designs was the data priority treatment. Since the amount of data is increasing, a kind of priority treatment is required.

Actually, priorities are not just a matter of future developments, it has already past that a flight test suffered from data overload due to the large number of messages with traces. These non priority messages can cause overflow in transmit buffer and create link loses between the UAV and the ground station. So, it can be noticed that a kind of priorities are already necessaries.

Another feature that has to be added as a consequence of introducing priorities is dynamic memory management. FIFOs are no longer a valid solution for transmission data storage (priorities are against FIFO principle) and a kind of dynamic memory is required.

In addition, a Dynamic Memory Space (DMS) will be useful for all the other features existing in the autopilot.
Design of an autopilot communications software based on open source.

Finally, this is the summary of the challenge:

A) Improvements on the existing solution.
   - More versatility can be achieved by using C functions instead of macros. In this new code, configuration options are moved from macro tokens (static options) to function parameters (variable options).
   - Code debugging and maintenance have to be easier. A new file codification using functions instead of macros gets this target.
   - Creation of a more comprehensive communications software architecture. This proposal wants to redraw the software considering something like objects instead of layers (that is, an Object Oriented Programming, even if the solution remains written in C for safety considerations). With a Object Oriented software, new software blocks can interact in many different ways and a more convenient software architecture can be found.

B) New features to be added.
   - Incorporate a feature for priority treatment that can rearrange downlink transmission messages. If possible, priority treatment has to be able to remove low priority messages when there is not space enough for high priority messages.
   - Develop a Dynamic Memory Space that can satisfy new memory characteristics required.
CHAPTER 2. DESIGN OF A NEW SOFTWARE COMMUNICATIONS ARCHITECTURE

This chapter is dedicated to explain the new software architecture in blocks and the analysis of what those blocks do at a conceptual level. Accurate analysis doing special emphasis on technical aspects is being done in CHAPTER 3.

Once microcontroller’s capabilities are no longer a limitation, a new versatile design (in architecture and programming) is possible. The new architecture substitutes the ancient idea of layers putted in cascade by another more convenient distribution of the elements. Figure 10 illustrates the new solution.

In this new architecture it can be seen the different elements involved in the communication for both downlink and uplink (some of them already existed but have changed a lot). It can also be noticed the paths followed by telemetry communications (in green) and datalink communications (in pink).
Design of an autopilot communications software based on open source.

A first look on the new architecture reveals a lot of conceptual changes. Let’s summarize them:

- **Above** driver layer it has appeared a *generic driver* layer. This new layer enables the communication system to use a larger number of drivers (with different characteristics) in an easy way. *Transport*, *messages* and *datalink* will always work with a *generic driver*, which has a single definition.

- **Transport** layer is no longer above driver (neither *generic driver*). Now both *transport* and *generic driver* are treated as utilities (actually both have been developed more or less like objects) that upper levels can use separately.

- **Downlink** is no longer in any chain. In the previous definition, *message* layer didn’t have any knowledge about *Transport* or *driver* and *downlink* introduced those elements. Now those elements are function parameters (variables) and *downlink* should not be required anymore. However, since the autopilot has not been modified in order to add those variables, file *downlink.h* is still there doing this definition. (So, *downlink* should disappear someday).

- New supporting elements have made their appearance. At the end of section 1.4 it is mentioned that the new software architecture has to face not only the existing software limitations but also the necessity of priorities. Those priorities are solved by *priorities* layer (it handles prioritized buffers) but then a second problem rises: holes appear in transmission buffer. Since there is not possible to have dynamic memory allocation in an embedded system, DMS (Dynamic Memory Space) provides a kind of dynamic memory management. For further information please refer to the technical aspects in section 3.7.

This first approximation to the final solution shows how new and old elements interact. However it is necessary to do an overview of the most important elements that will help to understand CHAPTER 3.
Design of an autopilot communications software based on open source.

### 2.1. Message layer

Now message layer has become the main manager of downlink (telemetry) communications. It builds the message content, it calls transport layer to encapsulate the message and it interacts with generic driver in order to send the final packet. Figure 11 illustrates the logical procedure.

**Figure 11**: Message layer logical procedure.
Design of an autopilot communications software based on open source.

Few things that have to be noticed from Figure 11:

- Driver is now based on transactions: Transactions contain data information like origin and length. Other data characteristics can be defined in a transaction structure. Driver no longer receive data but a pointer to a transaction referring to data.
- Dynamic Memory Space, DMS: this feature for dynamic memory management has been developed (see section 3.7) and now transactions and data are stored there.
-Reservation on driver's buffer: The number of transactions that a driver can store is limited and it is a waste of time to create a message if, at the end, there is no place in driver's buffer.
-Reservation on DMS: As last point, reservation has been made thinking in a future OOP migration. In case it is not possible to get memory, space in driver's transaction buffer is freed and transmission is aborted.
- If both memory and driver spaces are achieved, the procedure continues normally. However, now everything is much more simple: transport just takes care of encapsulation, driver takes care of transactions and message takes care of message's content.

2.2. Datalink layer

As message layer, datalink is the top level layer but, in this case, for uplink (datalink messages).

Datalink layer connects the received messages to their destinations. Time by time, datalink tells transport to get data from driver and analyse it. In case the transport finds a message, it passes the message using a single (for the existing code) callback function that treats it.

The modification introduced in the developed code is the possibility of different callbacks when a message is found. Thus, datakink use to create a single bridge between transport and the autopilot but now it creates more than one bridge.

Since the objective is not to create a static behaviour, these multiple links are performed using functions specifically designed for it. The execution of these functions are done during the start up (more information on 3.3).
Design of an autopilot communications software based on open source.

### 2.3. Transport layer

Once transport layer has been separated from driver issues, it just takes care of encapsulation / expansion of packets.

In an attempt to create a transport layer definition as intuitive and generic as possible, this layer elements have been coded like objects that can be easily exchanged by third party codes. That is, a transport element is a structure that contains its transport data and its subset of functions (API).

A transmission transport structure only contains the required subset of functions for message encapsulation (header and tail). This structure is used in downlink for telemetry transmission.

A reception transport structure contains the API for message expansion but also the required data for this. So, two elements can be found. This kind of structure is used in uplink for datalink reception.

Figure 12 : Tx & Rx transport layer concept.

With these simple structure is easy to encapsulate / expand messages using transport protocols based on a header and a tail (actually, all transport protocols work like this, no one puts data in the middle). What follows is a list of different transport protocols.
Design of an autopilot communications software based on open source.

Actually nowadays in Paparazzi there are implemented two protocols: Paparazzi and XBee.

XBee (ANNEX A.1.4) is RF modem that can work with its own protocol. However, even if it is included in Paparazzi project (and it has been migrated to this new communications version), it is almost never used.

2.4. Generic driver

Generic driver is a layer that supports a new interaction procedure between the existing drivers and the new communications system. Different reasons make this layer necessary: communications versatility, simplicity and subsystems compatibility.

Old communications software were designed for being used with just one driver available, UART, and this restricted its versatility. In order to improve the new design it was considered the possibility of new devices and, so, other drivers. For instance, new features could be added to telemetry than downlink transmission; they could be send to storage devices connected with drivers like SPI or I²C.

In addition, with the purpose of simplicity in message, datalink and transport layers, all drivers have to share a unique interface even if all they have different features.

Taking into account these considerations, existing driver interfaces could be modified to have a common appearance. However, drivers are used by other
Design of an autopilot communications software based on open source.

systems that would have to be updated. Since this kind of action would affect the whole autopilot, it is impossible to do it in that way.

More than that, it has no sense to write a specific driver with a generic interface. A specific driver must have its own interface. A generic interface that matches with every specific driver has to be designed in an upper layer. So, finally, the generic driver is a layer that has to connect any subsystem layer to any driver in a single way.

![Communications input/output interface](image)

*Figure 14: Communications input/output interface design.*
CHAPTER 3. TECHNICAL IMPLEMENTATION

CHAPTER 2 has shown the new communications software architecture with its elements, their purpose and interactions. The goal of this CHAPTER 3 is to present a technical explanation about the software implementation.

Let's recover the blocks scheme of Figure 10 that shows the in architecture in blocks. It looks like this:

This chapter is going to do an exposition of main blocks performing the new Paparazzi communication software plus the new two auxiliary blocks for transmission priority management.

3.1. Downlink

(files involved: downlink.c/.h)

In previous version, message layer managed data transmission without any knowledge about driver neither transport. Downlink managed those things.

In contrast, with the new version the proper driver and transport to be used are variables in message layer (this introduces the desired dynamic behaviour). Thus, downlink should no longer be required. However, since the autopilot has not been changed to provide those new variables to message, downlink has to be there in order to create them.
Design of an autopilot communications software based on open source.

Figures Figure 15 and Figure 16 show, respectively, the transport and driver selection (Figure 41 in B.1.1 is an extract form downlink.h that explicitly shows the code related to these choices).

The first part of the code manages the downlink transport choice using the pre-processor definition of “DOWNLINK_TRANSPORT” (made in the telemetry makefile). Two options are enabled: PPRZ (for Paparazzi transport encapsulation) and XBEE (for XBee encapsulation). Depending on the definition, a reference to the selected transport variable (transport_tx_PPRZ or transport_tx_XBEE) is assigned to the name DefaultChannel. Finally, autopilot will use DefaultChannel as transport layer reference.

Makefile
Option selection: PPRZ or XBEE
DOWNLINK_TRANSPORT = ...

downlink.h
Transport variable
Global name
transport_tx_PPRZ
transport tx XBEE
DefaultChannel

Figure 15 : Software procedure for downlink transport selection.

The reference to the selected driver (device and driver are used as synonyms in the code) is DefaultDevice and it is linked to driver variable in a similar way that it is for transport. The generic driver variable is obtained by joining ’&dev_’ at the pre-processor option “DOWNLINK_DEVICE” made in the telemetry makefile. For example, if “DOWNLINK_DEVICE” definition was “UART0”, the driver's variable would be ’&dev_UART0’.

Makefile
Option selection: UART0, UART1...
DOWNLINK_DEVICE = ...

downlink.h
Driver variable
Global name
dev_UART0
dev UART1
DefaultDevice

Figure 16 : Software procedure for downlink device selection.

A last comment must be said about driver selection: until now, just UARTs can be selected (from UART0 to UART6) but it is intended to extend this to SPI and I²C in order to improve software versatility.
3.2. Message

(files involved: messages.xml, gen_downlink_macros.ml, messages.h, gen_downlink_macros.ml, messages_data.h and messages_header.h)

The message layer is performed by a set of files that follows the next rule: a Xml file contains telemetry messages definitions and a generator file/s use/s it in order to create a header file/s. Finally, communications software uses the generated header file/s.

In section 1.2.1 it has been explained that message layer has three files in the existing version (one Xml, one generator and one header file). Now there are three files more files. File generator gen_messages.ml has been split into two files (gen_downlink_macros.ml and gen_downlink_data.ml) and, in consequence, messages.h has also been separated into two files, messages.h and messages_data.h. Finally, messages_header.h is a file for some support.

In order to make this exposition easy, file details are in ANNEX B.2 (ANNEX C is also strongly recommended because it highlights the many benefits of the developed design). However, a brief file summary:

- messages.xml contains all the telemetry message definitions.

- gen_downlink_macros.ml is the message macros/functions generator. That is, from this Ocaml file it is possible to create a tool that reads file messages.xml and, for each definition, it automatically creates a set of macros (old version) or functions (new version) for telemetry messages transmission.

- messages.h is the file created by gen_downlink_macros.ml that contains telemetry transmission procedures in the form of functions.

- gen_downlink_data.ml is another file generator wrote in Ocaml that builds messages_data.h (from messages.xml) for data packaging.

- messages_data.h is the file created by gen_downlink_data.ml that has message data pack/unpack functions.

- messages_header.h is a file that contains some helpful macros.
Design of an autopilot communications software based on open source.

### 3.2.1. Code generators

The code generators are Ocaml files that create automatically header files using the information taken from the Xml file. In the new communications software, the original file generator has been split into two in order to create two different headers that take care, each one, of transmission management (overview on section 2.1, further explanation on 3.2.3) and data packaging (section 3.2.2). The developed generators are completely different but the relevant differences have to be noticed on the new header files generated.

### 3.2.2. Data packaging

Special attention on message layer has to be putted on the new generated code for data packaging: messages_data.h (B.2.3).

This C language based file only takes care of data packaging and unpackaging in different ways (Figure 45, page 69, illustrates all data treatment structures and functions required for the ATTITUDE message).

This data treatment has been inspired on MAVLINK protocol management [1]. It provides a flexible, simple and efficient code for data management, specially for encoding (pack and encode functions) due to memcpy function. memcpy is much more efficient moving blocks of memory than copying byte per byte (as it was done with the old method).

However, Paparazzi’s data management has to deal with an issue when introducing this data treatment that is called ’Padding’.

Every data size needs a certain alignment in memory. When different types of data are placed in memory one after the other, it can occur that they cannot be placed in consecutive memory positions due to their different alignment rules. Thus, a gap in memory appears. This is the padding effect.

The global effect is that structures of different data types with a total size of N bytes can use more than N bytes in memory due to meaningless bytes lost in gaps.

An easy solution is just reordering the elements inside a structure, putting bigger elements at the beginning followed by smaller and smaller elements. This avoids padding problems (this is done in MAVLINK).
Design of an autopilot communications software based on open source.

Obviously, transmitted messages cannot contain meaningless bytes. Thus, there are only two possibilities: rearrangement or element-by-element transmission.

Rearrangement is not possible right now because this would affect reception, too (and reception is not going to be modified for the moment). However, moving element-by-element is not as efficient as moving entire messages using memcpy. Finally, the best solution possible for the moment is a padding dependant data treatment.

Packaging/unpackaging functions of messages_data.h, are different depending on the message. If the structure containing message fields has no padding, packaging/unpackaging use memcpy, as in Figure 45 (page 69), otherwise, functions treat data element-by-element as in Figure 46 (page 70).

### 3.2.3. Transmission management

Transmission management, performed by messages.h (B.2.2), has become the main file of downlink software architecture. It is a C language based file where downlink functions are described (one transmission function per message).

In this version it is easy to see the different steps followed during a transmission, the elements involved and how they interact (Figure 11). The transmission function has ten steps and each one is an interaction with another layer (transport, driver, DMS). It is a clear code easy for understanding, debugging and maintaining.

Actually, in addition to the existing transmission management, this new set of downlink functions are able to manage Dynamic Memory Space.

Let’s now summarize the technical properties of the new solution:

- It is based on functions instead of macros.
  
  The use of functions makes this solution better for debugging since errors can be easily found.
  
  In addition, functions are created as inline. This feature tells the compiler to paste the code wherever it is called (as it happens with a macro). Without this property, each function would be placed in memory and their calls would became jumps in memory to function’s position, which is a waste of time.
Design of an autopilot communications software based on open source.

- Transport and driver are function parameters. They both can be dynamically modified. This is another fundamental objective. Once communications system uses functions, both transport and driver must have the possibility of being modified during execution. In future, the autopilot may require flexibility in that way (i.e., redundant transmitters).

- Transport and driver present an API. In order to provide an easy management of transport and driver, both are developed as services, with an Application Programming Interface. Further information on section 3.4

- Messages have a priority parameter. Priority is discussed in section 3.6.

- Information is stored in a Dynamic Memory Space. Memory management is discussed in section 3.7.

### 3.3. Datalink layer

(files involved: datalink.c/.h)

_Datalink_ has been modified in order to be able to call different services when a message is received through a subscribed transport.

In the existing software datalink makes the uplink transport to check time by time its associated driver for new input data. In case a message is found, an associated callback is called.

In the developed design, however, the idea is not to centralize the input message treatment in a single callback. Every service has to be able to subscript a transport providing its own callback. Thus, more than one callback has to be associated to a transport.

In addition, since it is not the objective to introduce pre-processor rules, callback associations have to be done with a new registration function executed during the start-up. _Figure 17_ illustrates the registration process.
Design of an autopilot communications software based on open source.

3.4. Transport layer

(files involved: transport.h, transport_pprz.c/.h and transport_xbee.c/.h)

Transport layer files have been completely re-written in order to be not only a C language based code but also a kind of object oriented code. Section 2.3 has already presented an overview of the new design and there are not much more technical aspects than the code solution.

It is not the purpose of this chapter to present code lines. However, Figure 18 and Figure 19 illustrate the pseudo-code corresponding to Figure 12. The most important Is to notice that transport layer and generic driver (section 3.5) are coded in a very similar way. This makes the code more intuitive.
Design of an autopilot communications software based on open source.

(Uplink transport pseudo-code)

```c
// RX API (and RX_DATA) -------------------------------------------------------
struct transport_rx_data_common {
    // Transport name
    // Payload: data buffer & length
    // Flags: message received, overrun and error
    // Callback functions (array of callback pointers)
    // Associated Rx device pointer
};

struct transport_rx_api {
    (*init) (* data);
    (*message_received) (* data);
    (*register_callback) (* data, (*callback)(* msg_p, msg_len) );
    (*callback) (* data);
};

struct transport_rx{
    void * data; //this points to the transport data struct
    struct transport_rx_api api;
};
```

Reception data struct.
It contains the required elements for received data analysis.

Reception transport API.
It contains the transport functions required for uplink.

Reception transport struct.
It contains both Rx. data and Rx. API structs.

(Uplink transport pseudo-code)

```
// RX API (and RX_DATA) -------------------------------------------------------
struct transport_rx_data_common {
    // Transport name
    // Payload: data buffer & length
    // Flags: message received, overrun and error
    // Callback functions (array of callback pointers)
    // Associated Rx device pointer
};

struct transport_rx_api {
    (*init) (* data);
    (*message_received) (* data);
    (*register_callback) (* data, (*callback)(* msg_p, msg_len) );
    (*callback) (* data);
};

struct transport_rx{
    void * data; //this points to the transport data struct
    struct transport_rx_api api;
};
```

(Figure 18 : New Rx transport design.)

(Downlink transport pseudo-code)

```
// TX API -------------------------------------------------------
struct transport_tx_api {
    header_len; /*header*/
    tail_len; /*tail*/
};

struct transport_tx{
    struct transport_tx_api api;
} ;
```

Transmission transport API.
It contains the transport functions required for downlink.

Transmission transport struct.
It only contains the Tx. transport API.

(Figure 19 : New Tx transport design.)

The new transport design shows a code that looks like something similar to an object oriented code. A transport structure contains its own data (actually this only exists for reception) and its API interface.

This new design has two benefits. First, it is a simple way for other layers to interact with transport. Second, it is a design close to Object Oriented Programming (OPP).

As a final comment, it has to be said that transport.h describes the new transport design and transport_pprz.c/.h and transport_xbee.c/.h implement, respectively, Paparazzi transport protocol and XBee transport protocol.
3.5. Generic driver

(files involved: device.h and device_uart.c/.h)

Generic driver is a new driver interface created in order to solve different problems when connecting messages layer to driver layer.

In order to improve the communications versatility, the driver to be used in a message transmission is now a function parameter (notice ‘device’ parameter in Figure 43). However, as it has been already exposed in section 2.4, the new communications software must be able to work with different types drivers (last board designed by ENAC, see ANNEX A, enables new message destinations than RF devices like a SD card for logs). So, a new generic driver type is required.

A generic driver device provides a unique interface and simplifies input/output tasks. On the other hand, it is still required to have a specific driver for each input/output port type. The solution is a generic driver header file (device.h) - used by messages, datalink and transport layers - and a set of files making the bridge to the specific driver (like device_uart.c/.h for UART).

In addition, with the objective of a more comprehensive code, this layer is coded in a very similar way than transport layer (section 3.4). The goal is to get a design where generic driver elements can be used and shared like objects.

Figure 20 shows the generic driver pseudo-code (very similar to figures Figure 18 and Figure 19).

```
struct device_api{
  //Tx
  (*check_free_space) (* data, *slot_idx);
  (*free_space) (* data, slot_idx);
  (...)
  //Rx
  (*byte_available) (* data);
  (*get_byte) (* data);
};

struct device{
  * data;
  struct device_api api;
};
```

Figure 20 : Generic driver definition (device.h).
Design of an autopilot communications software based on open source.

And Figure 21 shows the specific bridge for UART (in device_uart.c). In this figure it can be shown how generic driver’s functions (like ‘dev_uart_free_space’) are matched with its specific UART functions (like ‘uart_free_space’).

Then after, all generic functions are included into a generic driver’s API. For example, generic driver ‘dev_UART0' contains the pointer to UART0 structure and the set of generic functions.

As a final comment it has to be said that other bridges to specific drivers are still under development (such as device_spi.c/h).

Figure 21: From generic to specific driver (UART case).

3.6. Priority management

(included files: transmit_queue.c/h)

This section is a result of an extra concept introduced during communications update process.

Since the complexity of UAS rises every day with new features (UAV capabilities, mission complexity, multi-UAV designs…) the amount and the types of data being exchanged are also being incremented. In that context not only UAV – Ground_segment, Ground_segment - UAV communications can be expected but also UAV-UAV with a many possible final network architectures on the ground and on the air.
Design of an autopilot communications software based on open source.

In such complex design it is obvious that different levels of importance in the communications domain will be highlighted and a data priority management will be required.

For the moment, even if this complex situation is not an actual topic yet, it is a good idea to start focusing on it. In addition, a tool like a priority manager would be already useful right now since, sometimes, test flights can produce too much test messages and cause overflow on telemetry link (this would create a fake fail of the test flight). Situations like this have already been encountered by ENAC’s developers during their tests.

The solution proposed for priority management has been the substitution of the old driver’s buffer (FIFO). The new buffer is a transmission queue that contains references, ordered by priority, to the messages. Actually this solution has only been implemented over UART driver.

### 3.6.1. Transaction based transmission

The introduction of priorities requires to face two problems: the priority management itself and the memory management of the prioritized data (discussed in section 3.7).

Without priorities, data management can be solved by the usage of FIFOs. In a FIFO buffer, what enters first is sent first and the physical implementation is a piece of memory where data is placed one after the other. On the other hand, once priorities are introduced, messages are placed in memory in a way and extracted for transmission in another. So, the distribution of messages in memory can look like random. However, even if data is in memory with, apparently, any kind of organisation, transmission driver still needs to be able to manage them.

The solution introduced in order to manage the messages with this new transmission feature is the use of transactions (as it is being used in SPI, for instance) because they can manage data as blocks. Transaction is a structure with a pointer to the message, its length and a callback function pointer (which is going to be used when transmission will be finished). A transaction is a structure that refers to a message which can be anywhere.

In conclusion, message priority management has become a transaction priority management.
Design of an autopilot communications software based on open source.

### 3.6.2 Transmission queue

Once transactions have been exposed as the way to exchange data between *message* and *driver* layers, now it is time for introducing priority management over those transactions.

Priority management is being made by `transmit_queue.c/.h` file (actually this file describes a prioritized queue and `transmit` word is there only because it has been originally designed for transmission). As a first approximation, a prioritized queue is an element that stores a list of pointers (for transmission it will store transaction pointers) with their priority and the index of the next pointer (according to priority). Figure 22 shows an empty transmit queue with four positions or ‘slots’. In addition to the slots, the transmit queue has an “index of the first element”.

![Index of the first element](image)

Figure 22: Empty transmission queue.

The transmission queue is designed as a chained list what means that each element indicates who is the next one. The benefit of this solution is a better way to rearrange data; in case the transmission order has to be modified, it is not necessary to move transaction elements, it is enough to modify their ‘next index’. Not moving data across the memory optimizes management execution.

In order to expose the rules for filling this new transmission queue, what follows is an example of different events that can happen. In this example (for simplicity) there are two types of priorities, High and Low. The example shows different transactions (drown like coloured boxes) with letters H or L, depending on if the transaction corresponds, respectively, to a High or to a Low priority message.

Figure 23 shows the queue when an element (transaction) with Low priority arrives. The element is placed in the first free slot (slot 1) and, since there is any other element, it is set as the first element in the queue. So, “index of the first element” indicates the slot 1. The ‘next index’ for this slot is empty because there is any other element in the queue.
Then, when another element arrives, the next free slot is assigned (slot 2) and the queue is reordered according to priorities. For the rearrangement, the queue is checked starting from the “index of the first element” (in this example, it indicates slot 1 for the moment). In Figure 24, the new transaction has a High priority and, since it has higher priority, the “index of the first element” is modified to point slot 2 and slot 2 points to 1. This is an insertion at the beginning.

Figure 25 corresponds to a third element with Low priority again. Slot 3 is assigned and the queue is checked for rearrangement. First element (slot 2) is checked; the new one has lower priority and, so, it can not be placed in front. Then next element (slot 1) is checked. Now the priority is the same. In that case, the order of arrival is respected and, in consequence, position of slot 1 is preserved. Since the end of queue has been reached, the slot 3 is placed there. This is an insertion at the end.
Finally, Figure 26 shows the arrival of a fourth element with High priority. Slot 4 is assigned and the order of the queue is revised. First element to check is slot 2, that has the same priority; since in that case the order of arrival is respected, it is necessary to keep following the queue. The next element is slot 1, with lower priority; thus, slot 4 has to be placed between slot 2 and 1. ‘Next index’ of slot 2 will now indicate slot 4 and ‘next index’ of slot 4 will indicate slot 1. This is the last possibility that can happen, an insertion in the middle.

Different considerations have to be summarized about this priority management:

- This is the simplest and effective way to manage priorities.
- The time required for free slot location depends on queue status.
- The time required for queue reordering depends on queue status.
Design of an autopilot communications software based on open source.

The variability of a subsystem's time response makes difficult to analyse the full system behaviour and should be avoided. However, since the transmission queue is not so big, the uncertainty introduced can be considered negligible and the benefits introduced are much more important.

Another point of interest (at a programming level) is that the transmission queue code takes into account the possibility of threats trying to interact at the same time with the queue. Since order rearrangements have to be done atomically, the code contains semaphores.

Also at the programming level It has to be noticed that transmission queue code can be compiled with different levels of optimisation, what prevents from error usages during development and fast execution at the end.

For finishing this section, what follows are the traces of a transmission made as in the example exposed. Four elements (red, green, blue and pink) are introduced into the queue with different priorities (one for green and pink, zero for red and blue) and then they are transmitted in the proper order (green, pink, red and blue).

```
transmit_queue: check_free_space:  ASSIGNED SLOT 0
transmit_queue: insert_slot_atomic:  EMPTY QUEUE. SLOT 0 INSERTED IN FIRST PLACE (PRIORITY = 0)
   Transmit queue order (slot):   { 0, }
   Transmit queue order (priority):   { 0, }
transmit_queue: check_free_space:  ASSIGNED SLOT 1
transmit_queue: insert_slot_atomic:  SLOT 1 INSERTED IN FIRST PLACE (PRIORITY = 1)
   Transmit queue order (slot):   { 1, 0, }
   Transmit queue order (priority):   { 1, 0, }
transmit_queue: check_free_space:  ASSIGNED SLOT 2
transmit_queue: insert_slot_atomic:  SLOT 2 INSERTED AT THE END (PRIORITY = 0)
   Transmit queue order (slot):   { 1, 3, 0, 2, }
   Transmit queue order (priority):   { 1, 1, 0, 0, }
transmit_queue: check_free_space:  ASSIGNED SLOT 3
transmit_queue: insert_slot_atomic:  SLOT 3 INSERTED IN THE MIDDLE (PRIORITY = 1)
   Transmit queue order (slot):   { 1, 3, 0, 2, }
   Transmit queue order (priority):   { 1, 1, 0, 0, }

→ Here starts the transmission!!
transmit_queue: extract_slot:  SLOT 1 EXTRACTED FROM QUEUE
   Transmit queue order (slot):   { 3, 0, 2, }
   Transmit queue order (priority):   { 1, 0, 0, }
transmit_queue: check_free_space:  ASSIGNED SLOT 3
transmit_queue: insert_slot_atomic:  SLOT 3 EXTRACTED FROM QUEUE
   Transmit queue order (slot):   { 0, 2, }
   Transmit queue order (priority):   { 0, 0, }
transmit_queue: check_free_space:  ASSIGNED SLOT 0
transmit_queue: insert_slot_atomic:  SLOT 0 EXTRACTED FROM QUEUE
   Transmit queue order (slot):   { 2, }
   Transmit queue order (priority):   { 0, }
transmit_queue: check_free_space:  ASSIGNED SLOT 2
transmit_queue: insert_slot_atomic:  SLOT 2 EXTRACTED FROM QUEUE
   Transmit queue order (slot):   { },
   Transmit queue order (priority):   { }
transmit_queue: extract_slot:  ANY SLOT IN QUEUE --> QUIT
```

Figure 27: Traces of transmission priority management.
3.7. Dynamic Memory Space

(involved files: dynamic_buffer.c/.h)

At the begging of this report it has been explained that priority management is considered to be an important tool for the communications software. Section 3.6 has shown how this feature has been added. However, priorities lead a problem concerning to messages management while they are in the transmitter's temporary buffer. Since data is no longer sent by order of arrival to transmitter (due to priorities), FIFO solutions for transmission buffer cannot be used anymore. Because of that, a new kind of memory management has been developed under the name of Dynamic Memory Space (DMS).

In a first version, this new memory management was introduced as a part of the transmission queue (see 3.6.2) but then after it has become a stand alone element because in that way it can be a used by other applications. The origin of this memory space (as a part of transmit queue) makes this feature to look like the transmission queue (at code level).

The DMS mainly consist on a range of memory and a list with descriptions of the memory segments allocated in that range. Every descriptor of the list is called slot and, as for the transmission queue, the list is a chained list (this means that every slot indicates which is the slot describing the next segment in memory). Figure 28 illustrates how this DMS looks like.

Figure 28: DMS structure and parameters.
Design of an autopilot communications software based on open source.

For a better exposition of how this memory is managed, what follows is an example of some DMS management actions.

Figure 29 is the status of a DMS at some point. An “index of the first element” (a DMS variable) indicates the slot of the chained list (slot 1 in this example) that defines the first segment occupied in the DMS memory range. As mentioned, each slot describes the segment (for slot 1, initial point at Byte 56 and 65 Bytes long) and indicates the slot with information of next segment (slot 2 in this case).

![Index of the first segment](image)

When more memory has to be allocated, DMS first checks for a free slot that could manage it and looks for free memory space (memory allocations cannot be fractioned and so it is necessary to find one single block of memory big enough for the allocation). If any slot is free, DMS rejects the memory allocation solicitude even if there is free memory because it is no possible to handle it. Of course, on the other hand, even if there is a free slot that could manage a new allocation, the solicitude is rejected if there is not free memory enough.

The most important problem that has to be faced in DMS is this allocation process. It introduces different problems. First, allocation process duration depends on the DMS status and this introduces uncertainty on system time response. Another problem is to find the best strategy for allocation. Since it is not possible to fragmentise data into different blocks, filling process has to be done in such way that memory remains as compact as possible.
The solution selected for allocation procedure is to put new data in the smallest free memory block among all blocks possible. In addition, when a block or segment is introduced, it is aligned with the previous element in memory.

![Diagram of allocable memory and list of segments](image)

**Figure 30**: DMS with a third slot introduced.

Still in the DMS example, Figure 30 shows how a new segment of 38 Bytes long is introduced in memory. This third new segment is assigned to slot 0 and placed in the central hole of DMS. All of the three existing holes in DMS are big enough but the smallest one is selected. This selection rule tries to keep the bigger holes for bigger data blocks that can arrive in future.

Figure 31 is another example of this allocation procedure. In this case a fourth element arrives. Slot 3 is assigned and there are 3 holes in DMS. However, since the hole in the middle is 13 Bytes long and the segment to be placed is 51 Bytes long, selection has to be done between the holes placed at the beginning and at the end. The beginning is selected because it is smaller.
Design of an autopilot communications software based on open source.

![Allocable memory](image)

**List of segments**

<table>
<thead>
<tr>
<th></th>
<th>Index</th>
<th>init</th>
<th>len</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>121</td>
<td>38</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>56</td>
<td>65</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>172</td>
<td>38</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>0</td>
<td>51</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 31: DMS with a fourth slot introduced.

Few comments and remarks for ending this section:

- This is not a message management feature (it was in a first version), this is a memory manage feature. This means this is a general tool that will be useful for every part of the code, allowing other programmers to take profit of it.

- The user API is very simple and understandable, minimizing the difficulty of learning for new programmers.

- This will provide a more efficient way for memory usage. Allocating specific memory ranges for each service always relays in a bigger amount of memory and, moreover, there are more problems than with one single memory space (i.e., saturation of a buffer while another is free).

- This provides a better way to exchange data between different services. Data doesn't need to be copied and even pointers and segment lengths can be avoided. Only the slot number of the chained list is required.

- There are few concerns about DMS behaviour. One is the system time response, which is not deterministic. The second one is the allocation strategy.
CHAPTER 4. FUTURE TASKS

After finishing this project many things have been detected as interesting points to consider and improve. Here there is a brief description of them

- Message rearrangement:
  - In section 3.2.2 it has been exposed a new data packaging procedure inspired in MAVLINK protocol (inspiration takes data management, not data content) that is more efficient.
  - However, when performing this MAVLINK data treatment on Paparazzi it was detected the appearance of padding in some message due to data arrangement. So it is not possible to take advantage of MAVLINK protocol for those message. A simple fields rearrangement could solve this problem but ground control software has to be modified, too.

- Priority analysis:
  - A priority analysis has to be done in order to assign the proper level to each message.
  - There have been some discussions on this topic and three levels were detected, but still as a first approximation.

- Suppression of low priority messages:
  - A very powerful feature in transmission queue would be the possibility to suppress those messages with very low priority. In case of overflow, high priority message could remove low priority ones in order to ensure a safety link.

- Improvement of generic driver layer:
  - Generic driver has been developed with the objective of using any driver available. However, just UART works perfectly and some SPI features. More work has to be done in that field.

- Improvement of DMS:
  - The dynamic memory tool developed is really useful for the purpose of this project. However, it presents some warnings:
    - It has safety access functions but it also gives free user access.
    - Performances are limited (i.e., Memory cannot be compacted).
Design of an autopilot communications software based on open source.

- Time response is not constant. Time searching free space depend on the memory status and the size of space required (more information on [2]).
CHAPTER 5. CONCLUSIONS

First of all, let’s remark that in general, due to the huge development of new applications being carried out by drones, it is more and more necessary to provide versatility to drone’s software in order to simplify management and make them able to work with different types of data with more or less priority.

Regarding to the amount of data, systems have to be dimensioned taking into account a large quantity of data in movement. This means that a temporary storage place (buffer) has to be created for different types of data. Due to the fact that embedded systems doesn’t have operational systems, because it is necessary to ensure a deterministic behaviour of the airplane, it is required to develop software tools for a dynamic management of memory. This project has presented a solution, DMS (see section 3.7), but a work has still to be done on it because its time response depends on the DMS status at each moment. However, this feature is not disturbing for who has done this project because there are already coming new studies [2] that will provide a more accurate solution for Paparazzi system. Alex Bustico (from ENAC) is already working on its implementation. Actually, the DMS interface has been done very simple because, in that way, any improvement could be introduced easily without affecting the rest of the software.

In addition, due to the diversity of data types, it is necessary to supply onboard systems with the capacity of prioritizing and routing communications. Let’s use the competition IMAV 2014 as a sample. Drones will have to work in a urban environment and it will be necessary to use drones as a communications relay. Thus, there will be two problems: how to distribute data across the net and how to prioritize information due to the large amount of data passing through the relay.

Let’s take a look to data routing through the net (even if this is not a problem faced by this project). In Paparazzi system there is already a solution being developed by Fabien (…?). This solution, instead of being based on “sender – recipient” structures (like IP communications), is based on subscriptions to types of messages, as in CAN bus philosophy, that forces to continuously redefine message headers depending on if it is more interesting to be subscribed to the whole information coming from a single drone or if it is more interesting to subscribe a certain type/s of data coming from all drones (i.e., the position). So, a conclusion that can be seen from data routing is that data structure is related to the net design, which has a long way to go through.
Design of an autopilot communications software based on open source.

The consequence is that software for data management has to be versatile and modifiable in order to fit any new net philosophy requirements. The redesign of communications software made in this project has been done in that way.

Regarding to the priority management capabilities, it has been detected that different types of preference have to be defined due to large amount of data with different importance. This project has developed the tools necessary for priorities management up to 256 levels of priority. At the same time it has tried to identify the fundamental levels. With the help of Gautier Hattenberger, it has been defined 3 fundamental levels:

- High: Acknowledgement messages. It is very important for the ground station to receive feedback of the commands sent.
- Normal: Telemetry data for air vehicle status guidance (speed, position, altitude, supply voltage, etc.).
- Low: Payload. This data is not necessary for flight safety neither for guidance and so it has the lowest level. In case of buffer overflow, it would be useful that in the future the transmission queue could remove this kind of data in favour of higher levels.

As a summary of key points:

There are 3 key points for the development of onboard communications software.

- **Dynamic memory**: Onboard systems need a way for dynamic memory management. DMS is a temporary solution, versatile and useful, with the problem of a variable time response. Future versions are already coming and DMS has been developed in order to accept improvements easily.

- **Scalable, upgradable and versatile software**: Topics like data routing in multi-UAV systems or payload data treatment will require a lot of investigation works until the final solution will be found. By the way, it is necessary to have a message management code versatile. Several things are required to the communications code:
  - To be comprehensible.
  - To be easy for debugging.
  - To be easy to modify protocols, message headers, order/content of data and priorities.

All this has found its solution with the new message management code. The new code, written in C language, is easy to understand and debug at the same time that optimization of execution has been kept the maximum possible. The usage of parameters and
the development of generic structures simplify the change of protocols and input/output ports.

- **Priorities**: A long way has to be done for the final definition of all priorities. An objective of this project has been the introduction of priorities management and the definition of the 3 fundamental levels. However, the priority management system accept up to 256 levels and, from those 3, it will be easy to define much more levels. As it has been seen during the software development, the most important factor that will determine the priorities is the topology of the future multi-UAV net which, at the same time, will depend on the applications requirements (i.e. airplane focused vs. data type focused). Thus, let’s say about priorities conclusions that management tools have been developed with the fundamental levels but it will be the final multi-UAV architecture (still under development) who will have the key for priorities definition.
ANNEX A. PAPARAZZI COMMUNICATIONS OVERVIEW

Paparazzi is a full system that comprises all aspects related to an Unmanned Aircraft System (UAS) management. This means that it includes both onboard and ground control elements required for mission planning and monitoring.

At the same time, Paparazzi includes software and hardware designs which can be freely download from [3] since it is an open-source project.

This annex presents the system communications overview. On it, both software and hardware characteristics are described, doing special emphasis on the onboard communication software since this project works on it in order to introduce some improvements and features.

A.1. Communications overview

As mentioned, Paparazzi has both ground control and airframe solutions. The link between them are wireless communications. Here it can be distinguished the safety link (a RF link used by safety pilot when any problem occur during the flight) and the data link. Figure 33 shows the Paparazzi system focusing on the communications.
Design of an autopilot communications software based on open source.

Data link works in both directions, Downlink (downwards) and Uplink (upwards), Messages going from air to ground (so, through downlink) is called telemetry. On the other hand, messages going from ground to air (uplink) is called datalink.

![Communications between the Paparazzi agents](image)

**Figure 34**: Communications between the Paparazzi agents.

### A.1.1. Software

Paparazzi software can be split into two big elements, ground station and onboard software.

- **Ground station** provides an interface for flight and mission management. Rx downlink data can be shown in a visual user interface. The user can monitor the aircraft position, status, flight plan, etc. He can also send orders or requirements through uplink.

- **Onboard software** integrates all the elements required on the aircraft. Thus, this part of software includes very different types of software as communications, flight plan/mission management or sensor data treatment.

The onboard software covers a wide area of purposes. The Figure 35 shows the airframe structure with all the elements that autopilot software will take care of (battery, datalink device, GPS, etc.).
Design of an autopilot communications software based on open source.

Figure 35: Airframe modules.

A.1.2. Hardware

As mentioned, Paparazzi system includes hardware solutions in addition to software. Different elements, such as autopilot, IR sensors, IMUs, RF modems and GPS have been developed. All their CAD, schematics, gerbers and BOMs are freely available on paparazzi repository. Pictures below show some hardware designs.

The most interesting elements for this project are those related to the onboard communications treatment. That is, the autopilot and the RF modem.
A.1.3. Autopilot

Paparazzi project has a big number of autopilot hardware designs and all them are being developed under two different types of microcontrollers units (MCU): STM32 and LPC21xx. LPC21 series (Paparazzi actually uses LPC2148) has been used for long time but nowadays new designs are based on STM32 due to its power (more RAM, more Flash, more clock speed). Let’s start summarizing the microcontrollers properties and then let’s see a summary of boards design.

A.1.3.1. Microcontroller

- **LPC2148**

  LPC2148 is a 32-bits microcontroller from NXP. Less powerful than STM32 microcontrollers, it works at 60MHz (slow clock speed for a microcontroller) and has no much RAM neither Flash memory but, in the other hand, it has a lot of peripheral devices. Thus, this microcontroller, even if it was not very powerful, had enough versatility for first board designs.

- **STM32**

  STM32 stands for the 32-bits microcontrollers family of ST Microelectronics. All STM32 have ARM-Cortex Mx architecture basis (ARM is a standard architecture and Cortex Mx, \( x = 0 \) to \( 4 \), indicates different levels of optimisation).

  Paparazzi project uses two families of STM32. One is F1 series (STM32F1), used by Delft University designs, that has a clock speed up to 72MHz and is Cortex M3 based (Cortex M3 uses ARMv7). Actually different F1 models have been used different Paparazzi autopilot boards with few differences in FLASH, RAM and peripheral devices.
Design of an autopilot communications software based on open source.

On the other hand, ENAC’s autopilot design uses F4 series that has a clock speed of 168MHz and a Cortex M4 basis (ARMv7). Cortex M4 means higher optimization that includes DSP instructions and Floating Point Unit.

Cortex M4 must be remarked since it makes a big difference in arithmetic algorithms. For instance, multiplication takes 1 clock cycle instead of 3 cycles required by Cortex M3. In addition, DSP instructions increase a lot arithmetic performances. DSP optimisation depends on each microcontroller design but the most common is the capability of doing two multiplications and sum their results in just one clock cycle. Filters are mainly based on this operation and, thus, time consumption of signal processing can be rapidly decreased.

- MCU summary

Next table summarizes the main characteristics of these used MCUs.

<table>
<thead>
<tr>
<th>MCU</th>
<th>STM32</th>
<th>LPC2148</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>F4 series</td>
<td>F1 series</td>
</tr>
<tr>
<td>Clock speed (MHz)</td>
<td>168</td>
<td>72</td>
</tr>
<tr>
<td>FPU</td>
<td>⨿</td>
<td>---</td>
</tr>
<tr>
<td>DSP instructions</td>
<td>⨿</td>
<td>---</td>
</tr>
<tr>
<td>Flash (kB)</td>
<td>1024</td>
<td>256 - 512</td>
</tr>
<tr>
<td>RAM (kB)</td>
<td>192</td>
<td>64</td>
</tr>
<tr>
<td>I²C</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>UART</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>SPI</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>PWM</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>CAN</td>
<td>⨿</td>
<td>⨿</td>
</tr>
<tr>
<td>USB</td>
<td>⨿</td>
<td>⨿</td>
</tr>
</tbody>
</table>

Table 1. Summary of MCUs characteristics used in different Paparazzi’s Autopilot design.

Information for table 1 has been extracted from [4], [5] and [6]

**A.1.3.2. Boards**

There are 12 hardware designs for Paparazzi autopilot. Table 2 (next page) shows the features of all those designs. This table shows the most relevant elements for software design. Other information, as weight, dimensions and power management can be found in an exhaustive table in [3].

Among the others, the most remarkable designs are the newest ones: Lisa/S (Designed by Delft University) and Apogee v1.0.0 (designed by ENAC).
Design of an autopilot communications software based on open source.

- Lisa/S

The most surprising thing in Lisa/S are its dimensions; Lisa/S has a square shape of 20mm x 20mm and a weight of 2.8g.

The most noticeable characteristic could be that it includes a GPS. On the other hand, because of its dimensions, this design presents a few number of communication interfaces (1 UART and 1 I²C).
Design of an autopilot communications software based on open source.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Lisa/L v1.1</th>
<th>Lisa/M v2.0</th>
<th>Lisa/S</th>
<th>KroozSD</th>
<th>Apogee v1.00</th>
<th>Umarim v1.0</th>
<th>Umarim Lite v2</th>
<th>NavGo v3</th>
<th>Tiny v2.11</th>
<th>TWOG v1.0</th>
<th>YAPA v2.0</th>
<th>HBmini/v2.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Part</td>
<td>STM32F103RE</td>
<td>STM32F105R</td>
<td>STM32F103RE</td>
<td>STM32F405RG</td>
<td>STM32F103RE</td>
<td>STM32F105R</td>
<td>STM32F405RG</td>
<td>LPC2148</td>
<td>LPC2148</td>
<td>LPC2148</td>
<td>LPC2148</td>
<td>LPC2148</td>
</tr>
<tr>
<td>Onboard Sensors</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEMS IMU</td>
<td>no</td>
<td>aspirin</td>
<td>yes</td>
<td>krooz/ext</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Magnetometer</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Barometer</td>
<td>yes</td>
<td>Yes</td>
<td>yes</td>
<td>yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Diff Pressure</td>
<td>yes</td>
<td>No</td>
<td>no</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>GPS</td>
<td>no</td>
<td>No</td>
<td>yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Input/Output &amp; Communication Interfaces</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UART</td>
<td>3 &amp; 1RX</td>
<td>2 &amp; 2RX</td>
<td>1 &amp; 1RX</td>
<td>3</td>
<td>3 &amp; 1Rx</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>I2C</td>
<td>2</td>
<td>1 + 1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>SPI</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>ADC</td>
<td>3 (12bit)</td>
<td>3 + 2 (12bit)</td>
<td>0</td>
<td>4 + 1 (12bit)</td>
<td>0 + 3 (12bit)</td>
<td>0 + 4 (10bit)</td>
<td>8^6</td>
<td>0 + 4 (10bit)</td>
<td>0 + 1 (10bit)</td>
<td>8 (10bit)</td>
<td>6 (10bit)</td>
<td>8 (10bit)(16bit)</td>
</tr>
<tr>
<td>PPM</td>
<td>6</td>
<td>6 + 2</td>
<td>6</td>
<td>10 + 1^5</td>
<td>6 + 1</td>
<td>6</td>
<td>6</td>
<td>0 + 1</td>
<td>8</td>
<td>8</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>PPM Capture</td>
<td>1</td>
<td>0 + 1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>R/C serial</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>1</td>
<td>1 (standard &amp; S.BUS)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GPIO</td>
<td></td>
<td></td>
<td></td>
<td>?</td>
<td>0 + 1</td>
<td>2 + 1</td>
<td>0 + 4</td>
<td>0 + 4</td>
<td>0 + 4</td>
<td>0 + 2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Onboard LEDs</td>
<td>8</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>USB Peripheral</td>
<td></td>
<td></td>
<td></td>
<td>Onboard USB</td>
<td>JTAG + UART</td>
<td>bootloader</td>
<td>no</td>
<td>bootloader</td>
<td>Bootloader</td>
<td>Bootloader</td>
<td>bootloader</td>
<td>bootloader</td>
</tr>
<tr>
<td>CAN</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>no</td>
<td>no</td>
<td>No</td>
<td>No</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>MODEM connector</td>
<td></td>
<td></td>
<td></td>
<td>Xbee</td>
<td>no</td>
<td>No</td>
<td>No</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>Xbee</td>
</tr>
<tr>
<td>Onboard Peripherals</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SD card/interf</td>
<td>yes/SPI</td>
<td>yes/SDIO + USB storage</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>RTG</td>
<td></td>
<td></td>
<td></td>
<td>yes + backup</td>
<td>yes</td>
<td>No</td>
<td>No</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Others</td>
<td>Overo w I/O incl. USB Host</td>
<td>On-board micro-USB B header</td>
<td>On-board mini-USB B header</td>
<td>On-board mini-USB B header</td>
<td>RS232 options</td>
<td>Buzzer</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2. Different hardware designs for Paparazzi autopilot and their main features.
Design of an autopilot communications software based on open source.

- Apogee v1.0.0

Apogee v1.0.0 is the last hardware design made by ENAC. It has the most powerful processor and a high connectivity, what provides a lot of possibilities for communication designs.

![Up side of Apogee v1.0.0](image1)

**Figure 36 : Up side of Apogee v1.0.0**

![Down side of Apogee v1.0.0](image2)

**Figure 37 : Down side of Apogee v1.0.0**

Some important facts related with communications:

- This design includes serial port interfaces as UART and SPI. This kind of connectors are used, for instance, for GPS and telemetry.
- It is possible to have Wifi communications over UDP protocol.
- SD Card can be used for log storage of different information types. Among others, telemetry can be saved.
A.1.4. RF Modem

The most important element required for data transmission is the RF device. XBee is a good solution because it provides a UART / SPI interface to the RF modem that accomplishes standard 802.15.4. The onboard chipset is small, what is really important for many UAV designs but has low power output (50mW). On the other hand, the ground antenna is much bigger.

Figure 38: Xbee RF Modem with a whip antenna.

Figure 39: Xbee RF Modem with SMA connector for a bigger antenna.

Figure 40: Xbee RF Modem for ground station (100mW and high sensitivity).
ANNEX B. CODE

In order to simplify the exposition of CHAPTER 3, code samples are placed in this chapter. At the same time it provides a more detailed explanation of each file.

B.1. Downlink

(files involved: downlink.c/.h)

There is not much code to show in downlink layer since there is just one file and, in the new software design, it takes care of just few assignments.

B.1.1. downlink.h

This file (Figure 41) only takes care, in the new design, of transport and driver variables selection.

According to the makefile definition of ‘DOWNLINK_TRANSPORT’ and ‘DOWNLINK_DEVICE’, the desired transport/driver variable is assigned to the general references DefaultChannel/DefaultDevice.

```c
// Transport
#if DOWNLINK_TRANSPORT == PPRZ
  #define DefaultChannel &transport_tx_PPRZ
#elif DOWNLINK_TRANSPORT == XBEE
  #define DefaultChannel &transport_tx_XBEE
#else
  #error DOWNLINK defined but no DOWNLINK_TRANSPORT found.
#endif

// Device
(...)
#define __dl_join(_y, _x) _y##_x
#define _dl_join(_y, _x) __dl_join(_y, _x)
#define dl_join(_chan, _fun) _dl_join(_chan, _fun)
#define DefaultDevice dl_join(&dev_, DOWNLINK_DEVICE)
```

Figure 41: Transport and driver selection from makefile options.
Design of an autopilot communications software based on open source.

B.2. Message

(files involved: messages.xml, gen_downlink_macros.ml, messages.h, gen_downlink_data.ml, messages_data.h and messages_header.h)

Notice that messages_header.h is not explained since it provides not much information.

B.2.1. messages.xml

This file define all the messages available in the Paparazzi communications protocol. For each message there is a name, and ID and a set of fields. Figure 42 shows an example of a message definition in a Xml file.

The message name says which message it is and which information it contains. When a message is sent, message name should be attached because, by doing this, it is no necessary the meaning of each data it contains.

Actually, communications software uses the message ID instead of message name. Message ID has the same meaning that message name but it is a number stored in 1byte (thus, Paparazzi protocol cannot have more than 256 different message definitions). Obviously, different messages cannot have the same name neither the same ID.

The different fields in message definition determine the message data content. The order is important because in that way, when a message is performed, only field data values have to be included, not their names. If transmission order has been respected by the sender, the receiver will be able to interpret the message content without field names.

Finally field types are important for software generation and treatment and units are important for field interpretation.

```
<message name="GPS" id="8">
  <field name="mode" type="uint8" unit="byte_mask"/>
  <field name="utm_east" type="int32" unit="cm" alt_unit="m"/>
  <field name="utm_north" type="int32" unit="cm" alt_unit="m"/>
  <field name="course" type="int16" unit="decideg" alt_unit="deg"/>
  <field name="alt" type="int32" unit="mm" alt_unit="m"/>
  <field name="speed" type="int16" unit="cm/s" alt_unit="m/s"/>
  <field name="climb" type="int16" unit="cm/s" alt_unit="m/s"/>
  <field name="week" type="int16" unit="weeks"/>
  <field name="itow" type="uint32" unit="ms"/>
  <field name="utm_zone" type="uint8"/>
  <field name="gps_nb_err" type="uint8"/>
</message>
```

Figure 42 : GPS message definition.
Design of an autopilot communications software based on open source.

B.2.2. messages.h

messages.h has become the main file for downlink transmission data management. It provides a set of functions (instead of macros in the existing version) that very easy to understand and debug/maintain. From this top level file, drivers, transports and even dynamic memory are managed.

Figure 43 is an example of the developed transmission software.

```c
static inline void DOWNLINK_SEND_GPS(struct transport_tx *tp, struct device *dev, const uint8_t *mode, const int32_t *utm_east, const int32_t *utm_north, const int16_t *course, const int32_t *alt, const uint16_t *speed, const int16_t *climb, const int16_t *week, const int32_t *itow, const uint8_t *utm_zone, const uint8_t *gps_nb_err) {
    uint8_t dev_slot, buff_slot;
    uint8_t ta_len = dev->api.transaction_len;
    uint8_t tp_hd_len = tp->api.header_len;
    uint8_t tp_tl_len = tp->api.tail_len;

    /* 1.- try to get a device's 'transaction' slot */
    if(dev->api.check_free_space(dev->data, &dev_slot)){
        /* 2.- try to get a slot in dynamic buffer */
        if(dynamic_buffer_check_free_space(&dynamic_buff, (ta_len + tp_hd_len + MSG_HD_LEN + DOWNLINK_DATA_GPS_LENGTH + tp_tl_len), &buff_slot)){
            /* 3.- get buffer pointer */
            uint8_t *buff = dynamic_buffer_get_slot_pointer(&dynamic_buff, buff_slot);

            /* SET TRANSACTION */
            /* 4.- set transaction in buffer */
            dev->api.transaction_pack(buff, (buff + ta_len), (tp_hd_len + MSG_HD_LEN + DOWNLINK_DATA_GPS_LENGTH + tp_tl_len), NULL, 0, &message_callback);

            /* SET MESSAGE */
            /* 5.- set transport HEADER in buffer (it depends on transport layer) */
            tp->api.header(buff + ta_len, DOWNLINK_DATA_GPS_LENGTH + MSG_HD_LEN);
            /* 6.- set message HEADER in buffer */
            msg_hd.msg_id = DL_GPS_ID;
            memcpy((buff + ta_len + tp_hd_len), &msg_hd, MSG_HD_LEN);
            /* 7.- set message DATA in buffer */
            downlink_data_GPS_pack((buff + ta_len + tp_hd_len + MSG_HD_LEN), mode, utm_east, utm_north, course, alt, speed, climb, week, itow, utm_zone, gps_nb_err);
            /* 8.- set transport TAIL in buffer (it depends on transport layer) */
            tp->api.tail(buff + ta_len, DOWNLINK_DATA_GPS_LENGTH + MSG_HD_LEN);

            /* SUBMIT TRANSACTION */
            /* 9.- send message */
            dev->api.transaction_submit(dev->data, dev_slot, buff, DL_GPS_PRIORITY);
            break;
        } else {
            /* 10.- release device's slot */
            dev->api.free_space(dev->data, dev_slot);
            break;
        }
    }
}
```

Figure 43: Actual GPS transmission function.
Design of an autopilot communications software based on open source.

**B.2.3. messages\_data.h**

What follows is a set of samples of different `messages\_data.h` functions. These examples show the different functions for message data treatment and the possible differences that can be found due to padding problem.

```c
static inline void downlink\_data\_GPS\_pack(uint8\_t *buff, const uint8\_t *\_mode, const int32\_t *\_utm\_east, const int32\_t *\_utm\_north, const int16\_t *\_course, const int32\_t *\_alt, const int16\_t *\_speed, const int16\_t *\_climb, const int16\_t *\_week, const uint32\_t *\_itow, const uint8\_t *\_utm\_zone, const uint8\_t *\_gps\_nb\_err) {
    uint8\_t *ptr = buff;

    DOWNLINK\_PUT\_1\_BYTE(\_mode);
    DOWNLINK\_PUT\_4\_BYTE(\_utm\_east);
    DOWNLINK\_PUT\_4\_BYTE(\_utm\_north);
    DOWNLINK\_PUT\_2\_BYTE(\_course);
    DOWNLINK\_PUT\_4\_BYTE(\_alt);
    DOWNLINK\_PUT\_2\_BYTE(\_speed);
    DOWNLINK\_PUT\_2\_BYTE(\_climb);
    DOWNLINK\_PUT\_2\_BYTE(\_week);
    DOWNLINK\_PUT\_4\_BYTE(\_itow);
    DOWNLINK\_PUT\_1\_BYTE(\_utm\_zone);
    DOWNLINK\_PUT\_1\_BYTE(\_gps\_nb\_err);
}
```

*Figure 44: Actual GPS data pack function.*
Design of an autopilot communications software based on open source.

Figure 45: Data Packaging/unpackaging functions for ATTITUDE message.
Design of an autopilot communications software based on open source.

Figure 46: Data packaging / unpackaging functions for DOWNLINK message.
ANNEX C. MESSAGE MANAGEMENT COMPARISON

This annex exposes the differences between the existing and the developed message management at file level.

C.1. Existing solution

The existing message management is performed by macros defined in file messages.h. This file is automatically generated by gen_messages.h and, for every Paparazzi message (see B.2.1), it has a set of macros for different management actions. For example, Figure 47 is the transmission macro for GPS message.

```c
#define DOWNLINK_SEND_GPS(_trans, _dev, mode, utm_east, utm_north, course, alt, speed, climb, week, itow, utm_zone, gps_nb_err){ 
    if (DownlinkCheckFreeSpace(_trans, _dev, DownlinkSizeOf(_trans, _dev, 27))) {
        DownlinkCountBytes(_trans, _dev, DownlinkSizeOf(_trans, _dev, 27)); 
        DownlinkStartMessage(_trans, _dev, "GPS", DL_GPS, 27) 
        DownlinkPutUint8ByAddr(_trans, _dev, (mode)); 
        DownlinkPutInt32ByAddr(_trans, _dev, (utm_east)); 
        DownlinkPutInt32ByAddr(_trans, _dev, (utm_north)); 
        DownlinkPutInt16ByAddr(_trans, _dev, (course)); 
        DownlinkPutInt32ByAddr(_trans, _dev, (alt)); 
        DownlinkPutInt16ByAddr(_trans, _dev, (speed)); 
        DownlinkPutInt16ByAddr(_trans, _dev, (climb)); 
        DownlinkPutInt16ByAddr(_trans, _dev, (week)); 
        DownlinkPutUint16ByAddr(_trans, _dev, (itow)); 
        DownlinkPutUint8ByAddr(_trans, _dev, (utm_zone)); 
        DownlinkPutUint8ByAddr(_trans, _dev, (gps_nb_err)); 
        DownlinkEndMessage(_trans, _dev )
    } else 
        DownlinkOverrun(_trans, _dev );
}
```

Figure 47: Existing GPS transmission macro.

Different things must be noticed over this macro:

- Transmission macros use ‘DownlinkXXX’ macros (where XXX stands for the concrete name of an action; i.e. ‘DownlinkStartMessage’ does the start up actions for message transmission). Thus, in this example is possible to confirm that:
  - In the existing software architecture downlink layer is under message layer.
  - The existing software has macros over macros.
Design of an autopilot communications software based on open source.

- Message transmission management is performed by macros:
  - `DownlinkCheckFreeSpace`
  - `DownlinkCountBytes`
  - `DownlinkStartMessage`
  - `DownlinkEndMessage`
  - `DownlinkOverrun`

- Message packaging is performed by different ‘Put’ macros like ‘`DownlinkPutUint16ByAddr`’ and ‘`DownlinkPutInt32ByAddr`’ (there is a macro for each data type. Each of these macros passes a data value to the driver Byte per Byte.

As a resume, three things have to be kept in mind about this message management:
1. Macro based solution.

**C.2. Developed solution**

The new solution is performed by functions defined in two files: `messages.h` and `messages_data.h`. In order to avoid repetition please refer to the file definition sections (B.2.2 and B.2.3 respectively).

The characteristics of this message management are explained in section 3.2. However, those that are relevant for the comparison are:
1. C language based solution.
2. Separated transmission management and data packaging.
3. Block memory treatment (when possible).

**C.3. Comparison**

The new message management accomplishes all the objectives considered. In comparison it has
- More versatility
- Easier understanding.
- Easier debugging and maintenance.
- The new priority management feature.
Design of an autopilot communications software based on open source.

Bibliography


[5] STM32F103xC / STM32F103xD / STM32F103xE Datasheet. ST Microelectronics
