

UNIVERSIDAD DE ALCALÁ

Escuela Politécnica Superior

Grado en Ingeniería de Computadores



TRABAJO FIN DE GRADO

QUADROTOR FLIGHT CONTROL
REAL-TIME EMBEDDED MODULE

Christopher T. Hyde

May 2016

UNIVERSIDAD DE ALCALÁ
Escuela Politécnica Superior
Grado en Ingeniería de Computadores
Proyecto Fin de Grado
QUADROTOR FLIGHT CONTROL
REAL-TIME EMBEDDED MODULE

Autor: Christopher T. Hyde

Director: Javier de Pedro Carracedo

TRIBUNAL:

Presidente: Melquiades Carbajo Martín

Vocal 1º: Ignacio Parra Alonso

Vocal 2º: Javier de Pedro Carracedo

CALIFICACIÓN:

FECHA:

Contents

List of Figures	I
List of Tables	III
Preface	IV
Acknowledgments	V
Abstract	VI
Resumen	VII
Extended Abstract	VIII
1 Introduction	1
1.1 Objectives	3
2 Embedded Systems	5
2.1 Raspberry Pi	8
2.2 Raspbian	10
2.2.1 Raspbian Installation	10
2.3 Linux kernel	13
2.4 Real-Time Operating Systems	15

2.4.1	Xenomai	18
2.4.1.1	Xenomai Installation	18
2.5	Developer Tools	24
2.5.1	Flight Control Module	24
2.5.2	Autopilot Panel	26
2.5.3	Quad-copter test-bed	27
3	Software Development Process	29
3.1	System requirements	30
3.2	System design	31
3.3	System implementation	36
3.3.1	Autopilot control panel implementation	36
3.3.2	Flight control module implementation	42
3.3.2.1	Interfaces priorities protocol	45
3.3.2.2	Flight control module file system	49
3.3.2.3	Real time tasks and queues definitions	50
3.3.2.4	Interface apc220_comm	51
3.3.2.5	Interface controller_comm	53
3.3.2.6	Interface xplane_comm	54
3.3.2.7	Interface panel_comm	58
3.3.2.8	Flight control module; autopilot program	68
3.3.3	System compilation	70
3.4	System testing and deployment	70
3.5	Project schedule (Gantt chart)	72
3.6	Project estimate costs	73
3.6.1	Hardware costs	73
3.6.2	Workforce costs	74
4	User Guide	75
4.1	Pilots guide	78

5 Proposed improvements	85
5.1 Remarks	87
References	89

List of Figures

2.1	Raspbian configuration window.	12
2.2	Linux kernel. [Source: www.commonswikimedia.org]	13
2.3	User Space and Kernel Space. [Source: www.quora.com]	14
2.4	real-time Kernel. [Source: www.ibm.com]	17
2.5	Eclipse ssh.	25
2.6	Quad-copter Test-bed.	28
3.1	Software development process.	29
3.2	Flight Control Module Components.	32
3.3	X-Plane Datagrams window.	33
3.4	Modules communications.	34
3.5	Interface messages.	35
3.6	Autopilot control panel window.	36
3.7	Autopilot control panel file system.	37
3.8	Flight control module implementation.	44
3.9	Flight control module interfaces.	45
3.10	Interface priority and message type.	48
3.11	Flight control program file system.	49
3.12	Automated takeoff procedure.	61
3.13	Automated landing procedure.	62
3.14	Automated yaw left maneuver.	64
3.15	Automated yaw right maneuver.	65

3.16	Automated roll left maneuver.	66
3.17	Automated roll right maneuver.	67
3.18	Automated forward maneuver.	67
3.19	Automated reverse maneuver.	68
3.20	Gantt chart.	72
3.21	Workforce costs.	74
4.1	Manual controller enable.	79
4.2	Manual controller disable.	79
4.3	Manual controller climb maneuver.	80
4.4	Manual controller descend maneuver.	80
4.5	Manual controller forward maneuver (pitch down).	81
4.6	Manual controller reverse maneuver (pitch up).	81
4.7	Manual controller yaw left maneuver.	82
4.8	Manual controller yaw right maneuver.	82

List of Tables

2.1	Embedded Systems modules specifications. [Source: Wikipedia]	7
2.2	Embedded Systems and Operating Systems.	8
2.3	Raspberry Pi modules specifications. [Source: Wikipedia]	9
2.4	Xenomai native skin API and most used real-time functions.	23
3.1	X-Plane byte values format.	57

PREFACE

This book is intended for programmers, embedded product designers, academic researchers and others who are interested in designing and working with embedded and real-time systems as applied to unmanned aerial vehicles (UAV). Basic skills in understanding and developing in ANSI C programming language, real-time operating systems and communication protocols are recommended.

Included is an in-depth explanation of an embedded real-time system designed to control a quad-rotor helicopter, a detailed explanation of an autopilot control panel and a flight manual.

ACKNOWLEDGMENTS

I would like to thank the following people for providing me with invaluable support, advice, knowledge and feedback, before, during and after I decided this was the project most applicable to my personal interests. Javier Martinez Cantos, as he has been my mentor for over a year, preparing me for whatever lies ahead in the embedded and real time world. My university professors Julio Pastor Mendoza and Oscar Rodriguez Polo, for helping me discover this fascinating field of study and for giving me the basic knowledge needed to build my future career upon. And last but not least I would like to give a very special thanks to my family and friends without whose support I could not have completed this course of study nor this project.



ABSTRACT

This project application has been designed to control a quad-copter helicopter, both manually and automatically. To achieve this, an embedded system with a real-time kernel is employed, enabling the use of many simultaneous tasks, thus providing continuous exchange of information between all the different components in the system which includes: flight simulator, manual controller (joystick), autopilot panel and real world quad-copter. This flow of information is read and treated accordingly and then sent to the corresponding module. The system creates an interface for each component in order to read and/or write data streams to and from input/output devices.

Keywords: *Embedded System, real-time, interface, flight simulator, manual controller.*

RESUMEN

La finalidad de este proyecto es el control tanto manual como automático de un quadrotor. Para ello se utiliza un sistema embebido con un sistema operativo en tiempo real, gracias al cual se consigue el procesamiento simultáneo de varias tareas. Esto facilita el intercambio de datos entre los distintos módulos que componen el sistema: simulador de vuelo, controlador manual o joystick, panel de autopiloto y quad-copter real. La información proveniente de cada módulo es leída y tratada correspondientemente para luego ser escrita en el módulo solicitante. El sistema crea un interfaz por cada componente con la finalidad de leer y/o escribir flujos de datos a cada dispositivo de entrada y salida.

Palabras clave: *Sistema Embebido, tiempo real, controlador manual, simulador de vuelo.*

EXTENDED ABSTRACT

There are two different approaches to flying a quad-motor helicopter. The first and most basic mode is *manual flight*. In order to fly manually, a manual controller is needed. For this project it is recommended to use a generic controller, which increases the flexibility of the system and makes the controller readily available and affordable to find or purchase. For this reason, the Sony DualShock®3 Wireless Controller has been selected.

This controller also has the advantage of having two joysticks, which enables the control of a higher number of parameters of the rotary wing aircraft, such as climb, descend, pitch, yaw and roll.

The second flight mode is by *autopilot*. The autopilot function is designed to send the desired flight commands to the *embedded system* which processes these flight commands and sends the corresponding flight instructions to the quad-copter helicopter.

The *embedded system* is built on a *Raspberry Pi model 1 b+*. This device offers reasonable performance at a low price. It is equipped with 4 USB ports, 2 more than the *Raspberry Pi 1 A model*, which allows the use of more peripherals, for example: wifi dongle, keyboard, manual controller, etc.

The selected operating system is *Raspbian*, which is a Debian based operating

system optimized for the Raspberry Pi family. Raspberry Pi and Raspbian offer the advantage of having the presence of an extensive support community.

To create a real-time system, the *Raspbian* kernel must be modified. *Xenomai* provides a real-time sub-system seamlessly integrated into Linux. Therefore, it has to be built as part of the target kernel. This can be achieved with cross-compilation or native compilation. With *Xenomai* installed on the native operating system, the system is now ready to use real-time API's as well as time constraints. *Xenomai* provides a dual-kernel configuration, namely user-space and kernel-space.

The flight controller module uses the real-time API's to create all the needed tasks and queues to send and receive data flows between the components. The main process of the *embedded system* creates the interfaces that communicate with each connected device.

The manual controller interface creates 2 real-time tasks. The controller device is connected to the *Raspberry Pi* via serial cable. The first task constantly checks the connection state of the manual controller by checking the device port status, returning an error if it is disconnected. If the manual controller is enabled, the second task is initiated. This task is responsible for reading the input from the manual controller. When new data arrives, this information is checked and translated into motor commands. To achieve this, the values that the manual controller provides must be interpreted. The joystick values range is from -32767 to 32767 on each axis. This range is converted into a 0 to 100 value range, which corresponds to valid motor values. The most important value that has to be taken into consideration is the *Center Null Zone “0”*. In order to make the flight control easier for the pilot, this value has to be changed into a recommended *hover* value. When the data becomes readable for the rest of the system, it is passed into a *LIFO (Last In First Out)* real-time queue in non-blocking mode, which is accessible by other tasks.

The autopilot panel is a control panel created with *Qt*. It is composed of a set of buttons that corresponds to different flight maneuvers: takeoff, land, climb,

descend, hover, yaw, roll and pitch. When one of the buttons is pressed, a *TCP* connection is established with the *embedded system* and a numerical value that identifies the action is sent to the control panel interface of the embedded real-time system.

The main task in the autopilot interface is in charge of creating a *TCP* server, which is continuously listening to the incoming port. When data is sent by the autopilot panel and then received by the *embedded system*, this task reads the integer value and it initiates the corresponding task depending on that value. This newly initiated task reads the values from a real-time queue provided by the real world quad-copter sensors or the flight simulator flight status values. Then it processes them and generates the motor speed values needed in order to perform the flight maneuver correctly. This new motors values are then passed into a *LIFO* (*Last In First Out*) real-time queue in non-blocking mode, which is accessible by other tasks.

The flight simulator used in this project is X-Plane 10. X-Plane provides a fairly easy and comfortable method to read and write flight parameters via *UDP* protocol and at the same time it is available for different platforms (Linux, Windows and Mac), which makes it very flexible and easy to deploy. The data sent and received by the simulator is in byte streams format. This will have to be carefully considered when transmitting data to and from X-Plane.

The X-Plane interface in the *embedded system* is in charge of reading and writing information to and from X-Plane. To achieve this, the interface initiates two different tasks, one to read values from X-Plane and the other one to write to X-Plane. The task in charge of writing reads the values stored in the queues created by other interfaces and translates each value into byte format. Then it sends these values via *UDP* to X-Plane. The task in charge of reading from X-Plane reads flight status values, translates the byte format values into decimal values and transfers them into a *LIFO* (*Last In First Out*) real-time queue in non-blocking mode, which is accessible by other tasks.

Finally, the radio communications interface is responsible for sending and receiving information to and from the real-world quad-copter helicopter via radio communications. This interface opens the serial port where the radio antenna is connected. It then initiates two tasks, one in charge of sending information to the quad-copter while the other is responsible for reading the incoming information from the quad-rotor. The information sent is normally an instruction containing the decimal values of the motors desired speed, while the data being read corresponds to an input of the quad-rotor sensors. The writing task gets the values from a queue that was previously created by other tasks and sends them via serial port to the quad-copter. At the same time the reading task reads the data received from the quad-rotor and processes it to make it readable by other components by transferring the received values into a *LIFO* (*Last In First Out*) real-time queue in non-blocking mode, which is accessible by other tasks.

The system is created with a hierarchy structure, giving each interface different priorities of execution. The interface with the highest priority is the communications interface, which is the most critical component because it represents a potentially dangerous device if not in control.

The joystick interface has the second highest priority, followed by the autopilot interface and finally the X-Plane interface.

There are several scenarios where the *embedded system* could work:

- Quad-rotor active and controlled with joystick: In this case there is no use for the sensor values read from the communications interface.
- Quad-rotor active and autopilot control panel active: In this scenario the quad-copter sensor values are read by the control panel interface and motor values are returned in order to complete the requested flight maneuver.
- Quad-copter and X-Plane simulator active: In this case the information retrieved from the sensors are sent to the X-Plane interface and then via

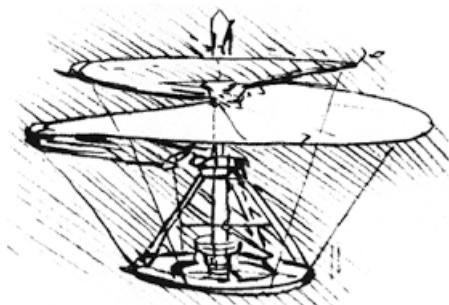
UDP to the simulator.

- Quad-rotor and manual or automatic control enabled and simulator enabled:
This scenario will behave as the previous three scenarios combined.
- Quad-copter inactive, X-Plane, manual controller or autopilot enabled: In this case, the flight sensors are received from the simulator rather than the quad-copter. The other interfaces make the simulator their primary sender and receiver.

Keywords: *Embedded System, Raspberry Pi, Raspbian, Qt, Xenomai, Interface, TCP, UDP, Serial, X-Plane, Autopilot, Radio Communications.*

CHAPTER 1

INTRODUCTION



LEONARDO DA VINCI was the first man that looked up into the sky with the dream that mankind would some day conquer the power of flight. He was a pioneer in aviation.

Although the first helicopter was not built until the 1940's, Da Vinci drew his first sketches presumably in the fifteenth century. Da Vinci's ideas of flying machines never left the drawing board but he set the first stones of what would become one of man's greatest technology races and achievements: being able to

fly! But it wasn't until 1903 when two brothers from Ohio actually took to the skies for the first time with powered flight. For the first time in history a man was lifted off the ground in a controlled flight.

In 1923 a Spanish civil engineer, Juan de la Cierva from Murcia, invented the first auto-gyro, which became the first rotary wing aircraft to take flight. The helicopters as we know them today didn't appear until 1939, when Sykorsky incorporated a single main rotor and tail rotor design.

The aviation industry has come a long way since then. World War II and the Cold War became the greatest boost periods for this industry. During the Cold War mankind started to look behind the clouds, looking up to the stars and giving birth to the space era. Our civilization was now developing the first computers and integrating them in aircraft. With new power-plants and new computers, man started to go faster and further than ever before. But this fast development came with a price. Aircraft needed skillful pilots capable of understanding and controlling all the components of the aircraft in order to achieve what is also the greatest strength in the aviation industry: safety and security.

As computers become more efficient, robust, faster, lighter and cheaper, they are taking over the flight controls and flight operations and they are slowly taking the spotlight away from pilots. As computers take control of aircraft, pilots are becoming systems supervisors and responsible for configuring the aircraft software to perform the desired maneuver. This is slowly leading into an unmanned era, where aircrafts are completely autonomous or remotely operated.

Although the first use of unmanned aerial vehicles (UAV) can be traced back to the late nineteenth century, when balloons loaded with explosives were used, it is not until the beginning of the twenty first century when computers taken over completely and UAV's, as we know them, were created. These UAV's were finally able to deploy in any mission and return home without the interaction of men. At first, this technology was only used in military applications, but it slowly grew in importance for the civilian world.

Today, UAV's are used in many areas in the civilian world, such as reconnaissance, perimeter control, filming, mail or package delivery and for personal amusement. The prices of these machines have also experienced a big change. They have become so inexpensive that almost anyone can acquire their own drone.

As UAV's become more affordable, a large community has been created around this technology eaguer to extract more functionality suitable to their UAV's.

1.1 Objectives

This project strives to accomplish several objectives. First of all, it will attempt to explain, as clearly and detailed as possible, how an embedded system with a real-time kernel is created to host an unmanned aerial vehicle flight control program.

Because this project aims to serve educational and research purposes, it will also provide the tools and steps necessary to help other developers to modify, adapt and expand the project as needed with new functionality.

This project functions include manual controlled flight, automatic control flight and flight emulation with a flight simulator. But many other capabilities can be added, for example a remote view of an attitude indicator, enabling IFR flight operations, live camera feed or autonomous flight procedures among many others. The sky is the limit!

The project has been created using independent standalone modules with their own private tasks. It has been designed with the intention of following the next schematic:

$$1 \text{ SYSTEM FUNCTION} \Leftrightarrow 1 \text{ PROGRAM} \Leftrightarrow 1 \text{ MODULE}$$

This feature enables easy integration of new modules to the system. This modularization approach has been found to be very effective when applied to flight control systems. Modules can work independently from one another, thus giving a good response in the event of systems failures. It also enables a fairly simple program debugging.

Modules communicate with each other through message queues, which makes the system very flexible and easy to implement new modules with new functions as everything comes down to read/write operations.

CHAPTER 2

EMBEDDED SYSTEMS

EMBEDDED SYSTEMS are complex computer systems that are designed to achieve a specific functionality. Opposed to general purpose computers, embedded systems normally do not include monitors or keyboards. This definition covers over 90% of all computing systems, for example, cell phones, printers, digital cameras, microwaves, etc.

Embedded systems are normally “embedded” into bigger systems. These devices are created with software embedded into hardware, which makes the system dedicated for an application or specific part of an application or product or part of a larger system. It is usually required that an embedded system satisfies a set of non-functional requirements including low-power consumption and real-time predictability.

Other features and requirements of embedded systems include:

- Non-stop operation, system never shutdowns.
- Executes a single program repeatedly.

- Embedded systems are typically designed to meet real-time constraints.
- Embedded systems must meet high safety and reliability standards.

An embedded system is composed of a microcontroller, memory, inputs and outputs. If the hardware is designed for specific purposes, this system is referred to as system on a chip (SoC) which holds a complete system: processor, floating point unit, memory cache and interfaces on a single integrated circuit. SoCs can be made as a special-order application-specific integrated circuit (ASIC) or by using a field-programmable gate array (FPGA) which can be programmed.

The other hardware design available for embedded systems are the computer boards. These systems also include a microcontroller, memory and inputs and outputs, but they are also designed to function under a light-weight operating system, in many cases real-time operating systems (RTOS). These devices can be programmed in a high-level programming language and also provide common operating systems functionality and software, which makes them a very good option if the engineer is looking for a flexible system. There are currently many varieties of boards available from different producers. Thanks to this wide range of systems in the market, these devices have become very affordable and they offer the necessary specifications to perform many of the requirements needed.

Embedded systems are designed to carry out specific functions with high safety and reliability requirements. For this reason, they are very important systems in the medical and aviation field. A flight control system for a quad-copter fits this definition as it requires high level of reliability and safety at low cost. There are a variety of boards for this application that could fulfill the required task, so an in depth study of the different devices must be carried out in order to select the proper board to meet the project specifications.

The following list shows some of the most well known boards available:

- Odroid.
- HummingBoard.

- BeagleBone.
- Raspberry Pi.
- Mbed.
- Arduino, etc.

The hardware included in all of these boards is very similar. The following table shows a comparison between these devices.

Specifications	Raspberry Pi Model B+	Odroid C1	Humming Board
SoC	Broadcom BCM2835	Amlogic S805	Freescale i.MX6 Dual
CPU	700 MHz single-core ARM1176JZF-S	ARM Cortex-A5 1.5Ghz	ARM Cortex-A7 1Ghz
GPU	Broadcom VideoCore IV @ 250 MHz	ARM Mali-450 MP2 GPU @ 600MHz	Vivante GC2000 Quad Shader OpenGL ES 2.0 1080p H.264
Memory	512 MB (shared with GPU)	1Gb	1Gb
Storage	MicroSD slot	MicroSD slot	MicroSD slot, mSATA
Networking	10/100 Mbit/s Ethernet	10/100 Mbit/s Ethernet	10/100 Mbit/s Ethernet
USB	4	4	4
Power Ratings	600 mA (3.0W)	600 mA (3.0 W)	600 mA (3.0 W)
Power Source	5V	5V	5V

Table 2.1: Embedded Systems modules specifications. [Source: Wikipedia]

The previous table shows the differences between the various modules. Each system offers different CPU speeds, memory capacity and number and types of inputs and outputs. But a board must not be selected relying only on its hardware specifications. Before selecting a board on which to build the project, it is recommended to take other issues into consideration. In this instance, a deeper investigation must take place in order to avoid future problems.

After considering the hardware resources available, it is also very important to know what operating systems are compatible with each board. In this manner, the Arduino and Mbed boards rely on bootloaders and do not use operating systems.

This project, the flight controller module, requires an operating system, so these boards do not meet the needed requirements.

The other boards listed are all available with various operating systems for each of them. Most of the operating systems are Unix based, but there also a few versions of Windows available for some devices, for example, Raspberry Pi 2 can operate a light version of Windows 10.

The next table shows a relation between the boards and the available operating systems.

	Raspberry Pi Model B+	Odroid C1	Humming Board
Operating Systems	Linux (Raspbian, Debian, OpenELEC, Fedora, Arch, Gentoo), FreeBSD, NetBSD, etc.	Ubuntu 14.04, Android KitKat	Forked kernel recommended, BLOB, Android, Ubuntu, Debian Linux, Raspbian.

Table 2.2: Embedded Systems and Operating Systems.

After evaluating the available operating systems, the last and probably *most important resource to consider making a selection is the existence of a development community* for the desired board. The bigger the community the more resources and libraries become available, and it also makes it easier to solve possible problems. From all the devices previously mentioned, *the board that offers the largest development community, the proper operating system compatibility and a good quality to price ratio is the Raspberry Pi*.

2.1 Raspberry Pi

Raspberry PI, as defined by their developers, “is a series of credit card-sized single-board computers developed in the UK by the Raspberry Pi Foundation with the intention of promoting the teaching of basic computer science”. The development of the Raspberry Pi (RPi) boards began in 2006 based on the Atmel ATmega644 microcontroller. But it wasn’t until 2011 that they were finally launched and

made available to the public. By 2015 there have been over five million Raspberry Pi's sold worldwide, making it the second fastest selling British made personal computer.

Since they first appeared, there has been different RPis developed with different specifications.

Specifications	Model A	Model A+	Model B	Model B+	2 nd Generation Model B
SoC	Broadcom BCM2835 (CPU, GPU, DSP, SDRAM, one USB port)				BroadcomBCM2836
CPU	700 MHz single-core ARM1176JZF-S				900 MHz quad-core ARM Cortex-A7
GPU	Broadcom VideoCore IV @ 250 MHz OpenGL ES 2.0 (24 GFLOPS) MPEG-2 and VC-1, 1080p 30 H.264/MPEG-4 AVC high-profile decoder and encoder.				
Memory	256 MB (shared with GPU)	512 MB (shared with GPU)	1 GB (shared with GPU)		
Storage	SD/MMC/SDIO card slot	MicroSD slot	SD/MMC/SDIO card slot		MicroSD slot
Networking	none		10/100 Mbit/s Ethernet, (8P8C) USB adapter		
USB	1 (direct from BCM2835 chip)	2 (via the on-board 3-port USB hub)	4 (via the on-board 5-port USB hub)		
Power Ratings	300 mA (1.5 W)	200 mA (1 W)	700 mA (3.5W)	600 mA (3.0W)	800 mA (4.0 W)
Power Source			5V via MicroUSB or GPIO header		

Table 2.3: Raspberry Pi modules specifications. [Source: Wikipedia]

As explained earlier, the Raspberry Pi was the board that has been found to best fit this project requirements. It has the biggest community, which can provide invaluable support, and it also has many robust libraries and software available. Many operating systems are also available for this unit.

It is also very important to be able to modify the operating system's kernel in order to provide real-time functionality. Kernel modification is not an easy task, but thanks to the developers community behind the Raspberry boards, this task has been previously dealt with and has been found to be feasible. When applying a real-time patch onto a new kernel, it is necessary to have the specific patch available for the hardware architecture it is going to be mounted on. After studying the design of all the previously mentioned boards and the real-time

kernel patch available, the Raspberry Pi 1 B+ model, due to its version of ARM architecture, was the only board that could have its kernel modified.

Raspbian, a Debian distribution, was the operating system which offered the correct Unix kernel used to configure the real-time system on the Raspberry Pi B+ model.

2.2 Raspbian

Raspbian is defined by its developers as “a free operating system based on Debian optimized for the Raspberry Pi hardware. Raspbian is an unofficial part of Debian Wheezy armhf with compilation settings adjusted to produce optimized “hard float” code that will run on the Raspberry Pi. This provides significantly faster performance for applications that make heavy use of floating point arithmetic operations. All other applications will also gain some performance through the use of advanced instructions of the ARMv6 CPU in Raspberry Pi. It comes with over 35,000 packages and a pre-compiled software bundled in a nice format for easy installation on your Raspberry Pi”. The installation of Raspbian onto an SD Card depends on the operating system used to create the operating system image.

2.2.1 Raspbian Installation

- Mac OS.
 1. Insert SD Card into slot and format. It is recommended to use a 8Gb or bigger memory card.
 2. Download the newest Raspbian Distribution compatible.
 3. Download RPi SDCard Builder (available only for mac).
 4. Uncompress the OS image and follow the program’s instructions.

5. Unmount and remove SDcard from the computer and insert it into the Raspberry Pi.
 6. Run `sudo raspi-config` to expand the file system and configure the system.
- Linux. (This method can also be used with Mac OS).
 1. Insert SD Card into slot and format. It is recommended to use a 8Gb or bigger memory card.
 2. Download the newest Raspbian Distribution compatible.
 3. Run the “dd” command:
`dd bs=4M if=2015-05-05-raspbian-wheezy.img of=/dev/sdd`
 4. Unmount and remove SDcard from the computer and insert it into the Raspberry Pi.
 5. To expand the file system and configure the system, run:
`sudo raspi-config`

If a monitor is connected to the Raspberry Pi, Raspbian will offer two different visualization formats. The first is a desktop-like screen, enabling the use of a mouse, a keyboard and the normal windows and icons on the screen. This view has the drawback of needing a lot of the machine resources in order to function. Another available method is via a terminal-like screen, where you only have access to a command-line. This offers many advantages by reducing the use of hardware resources due to the lack of graphics.

Raspbian makes it very easy to change from one view to another.

By typing: `$ sudo raspi-config` in a command window, the following window will appear.

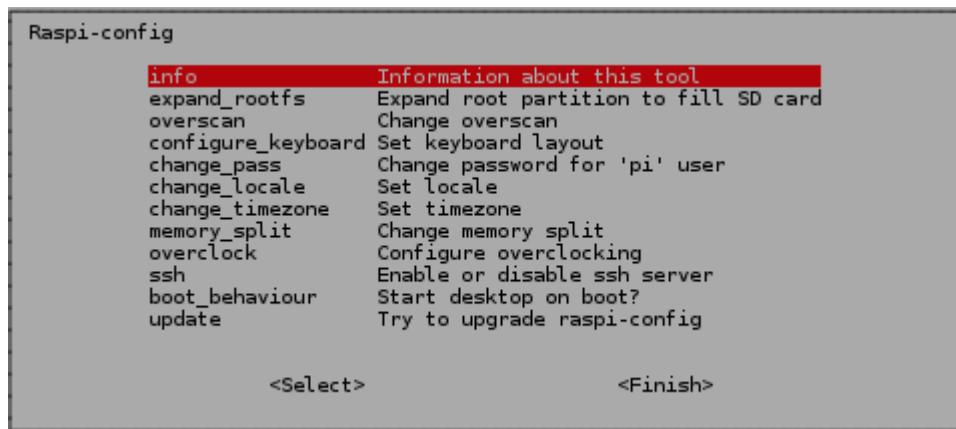


Figure 2.1: Raspbian configuration window.

Through this window Raspbian gives you access to many different options. Some of which are very important, for example, the ability to overclock the CPU. Although dangerous, this is found to be quite a normal procedure for the Raspberry Pi. Other options include the configuration of keyboard, time zone configuration, etc. It also provides the possibility to enable or disable ssh server.

As it is intended to act as an embedded system, the normal procedure for communicating with the Raspberry would be remotely, for which enabling ssh server becomes a necessity.

After the operating system has been properly setup, the next step is to configure the required peripherals that are going to be needed in order to operate the unit properly . First of all, in order to be able to establish ssh connection with the Raspberry, a network connection must be available. Due to the project requirements, it is necessary to have wifi communications available. These are the steps needed to configure a wifi connection via command line.

1. `sudo vi /etc/network/interfaces`
 2. Add the following to `/etc/network/interfaces`:
- ```
allow-hotplug wlan0
```

```
iface wlan0 inet dhcp\nnewline
wpa-conf /etc/wpa_supplicant/wpa_supplicant.conf
iface default inet dhcp

3. sudo vi /etc/wpa_supplicant/wpa_supplicant.conf

4. Add the following to /etc/wpa_supplicant/wpa_supplicant.conf
```

```
update_config=1
network={
 ssid="YOURSSID"
 psk="YOURPASSWORD"
 # Protocol type can be: RSN (for WP2) and WPA (for WPA1)
 proto=WPA\nnewline
 # Key management type can be: WPA-PSK or WPA-EAP
 #(Pre-Shared or Enterprise)
 key_mgmt=WPA-PSK
```

Once the network has been configured, the Raspberry will be remotely accessible and all the basic configurations will have been completed. The next step is to prepare the kernel to handle real-time operations.

## 2.3 Linux kernel

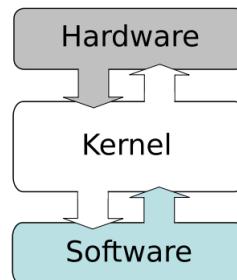


Figure 2.2: Linux kernel. [Source: [www.commons.wikimedia.org](http://www.commons.wikimedia.org)]

The kernel is the central module of an operating system. It is the part of the operating system that loads first and it remains in main memory. Because of this, it is important for the kernel to be as small as possible while still providing all the essential services required by other parts of the operating system and applications. The kernel code is usually loaded into a protected area of memory to prevent it from being overwritten by programs or other parts of the operating system.

Typically, the kernel is responsible for memory management, process and task management and disk management. The kernel connects the system hardware to the application software. It is responsible for managing input and output requests from software. The kernel performs its tasks, such as executing processes and handling interrupts in kernel space, whereas all operations performed by users are executed in the user space. This separation is made to prevent user data and kernel data from interfering with each other and thereby diminishing performance or causing the system to become unstable (and possibly crashing).

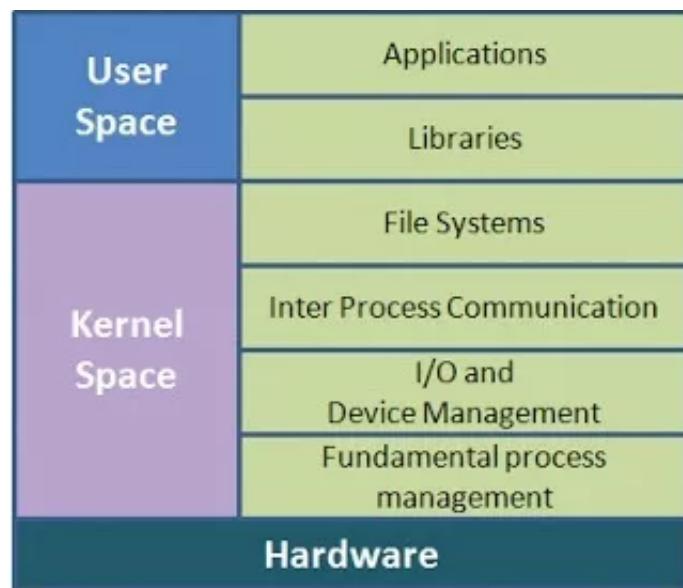


Figure 2.3: User Space and Kernel Space. [Source: [www.quora.com](http://www.quora.com)]

This figure illustrates the separation and functions of the user space and the kernel space.

The kernel must be modified to have access to real-time operations. To achieve this, the kernel is patched in order to provide new functionality. Once this is completed, the system will now be a fully functional real-time Operating System.

In following chapters there will be a detailed explanation on how to install a real-time kernel patch onto the Raspberry Pi and Raspbian operating system.

## 2.4 Real-Time Operating Systems

Real-Time operating systems are systems that guarantee a certain capability within a specified time constraint.

A key characteristic of an RTOS is the level of its consistency concerning the amount of time it takes to accept and complete an application task. As it has been specified earlier, real-time systems must also be very reliable.

There are two types of real-time systems.

1. Soft real-time systems.
2. Hard real-time systems.

Soft real-time systems are those in which missing time constraints does not lead to a system failure. In other words, missing deadlines is not considered to be an error. Some applications on which these systems are used are: desktop audio and video, virtual reality, internet telephony, non-critical real-time systems etc.

On the other hand, hard real-time systems operate within the confines of a stringent deadline. If deadlines are not met, the system is considered to fail. These systems are the basis of mission critical devices. Examples of hard real-time systems can be found in pacemakers, anti-lock brakes, avionics, etc.

The kernel must be modified to host real-time operations, and an analysis of the existing RTOS must take place in order to select the proper system that is compatible with the current kernel and hardware.

There are many RTOS available, most of which could be installed on the Raspberry Pi B+ model. The following is a list of some proprietary RTOS available.

- VxWorks.
- QNX.
- Windows CE.

As this project is designed for educational and research purposes, an open source RTOS is preferred. The following is a list of some open source RTOS available:

- FreeRTOS.
- ChronOS.
- RTLinux.
- Xenomai.
- Others.

All of the real-time operating systems listed are available with extended API's. Another important resource that real-time systems provide is the use of real-time POSIX standards.<sup>1</sup>

The following illustration is an example of how a system kernel is modified after a real-time patch has been installed.

---

<sup>1</sup>POSIX (Portable Operating System Interface for Computer Environments) standard, which is based largely on the UNIX System. POSIX is a group of IEEE standards that define the API, shell, and utility interfaces for an operating system.

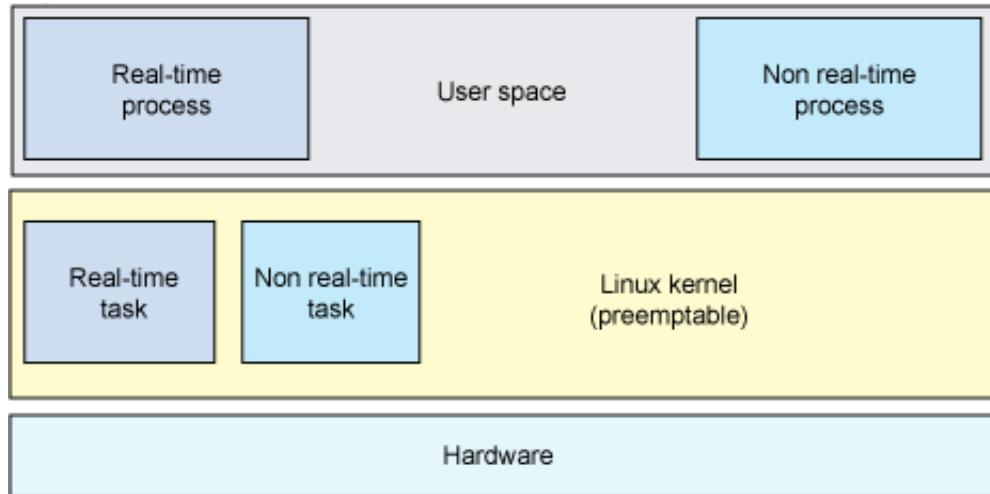


Figure 2.4: real-time Kernel. [Source: [www.ibm.com](http://www.ibm.com)]

As shown in the preceding figure, tasks become available with real-time system support. Tasks are the equivalent to threads in non real-time systems. Each task created in a real-time system is created with a specific priority and periodicity. The unit responsible for the execution of each created task is the scheduler. The scheduler is the part of the kernel responsible for deciding which task should be in execution at any particular time. The kernel can suspend and later resume a task many times during the tasks lifetime.

Systems without real-time support have a preemptive kernel. In most multithreading environments (also called multitasking) a preemptive kernel allows the thread with a higher priority to receive more time on the processor. And conversely, a lower priority thread will have less time with the processor. In this scenario, no particular thread can monopolize the services of the host processor permanently, regardless of its priority. Thanks to this, programs will never hang up even if an arbitrary thread of the program goes into a “forever loop”. On the other hand, on real-time systems, the kernel supports preemption and real-time features.

This means that a task<sup>2</sup> can run forever when the following conditions are met:

1. It is not blocked by synchronized resources (Mutex, Semaphores, ...).
2. It is not preempted by threads which may have equal or higher priority.

The RTOS used for this project is Xenomai because it offers an API that is simple to use and it has already been installed and tested on Raspbian OS by the Raspberry developer community.

### 2.4.1 Xenomai

Xenomai as defined by Xenomai.org is “a real-time development framework co-operating with the Linux kernel, to provide a pervasive, interface-agnostic, hard real-time support to user space applications, seamlessly integrated into the Linux environment”. Another definition from the Xenomai team reads: “Xenomai is about making various real-time operating system APIs available to Linux-based platforms. When the target Linux kernel cannot meet the requirements with respect to response time constraints, Xenomai can also supplement it for delivering stringent real-time guarantees based on an original co-kernel technology”. As previously indicated, the kernel must be configured in order to install Xenomai on the Raspbian.

#### 2.4.1.1 Xenomai Installation

This installation procedure is intended for kernel cross compilation<sup>3</sup> and not native compilation. It is also recommended to use 32bit Ubuntu to do the compilation.

---

<sup>2</sup>Real-time equivalent to threads.

<sup>3</sup>Cross compilation requires compilation on Linux kernel on a secondary machine.

1. Download setup files.

- Create a working directory.

```
$ mkdir /home/user/cross
```

- Download and install the compilation tools.

```
$ git clone --depth=1 git://github.com
/raspberrypi/tools.git
```

- Download the kernel.

```
$ git clone -b rpi-3.8.y --depth 1 git://github.com
/raspberrypi/linux.git linux-rpi-3.8.y
```

- Download Xenomai.

```
$ git clone git://git.xenomai.org/xenomai-head.git
xenomai-head
```

2. Apply Xenomai patches.

- Apply pre-patch.

```
$ patch -Np1 < ../xenomai-head/ksrc/arch/arm/patches/
raspberry/ipipe-core-3.8.13-raspberry-pre-2.patch
```

- Apply patch.

```
$ xenomai-head/scripts/prepare-kernel.sh --arch=arm
--linux=linux-rpi-3.8.y \--adeos=xenomai-head/ksrc/arch
/arm/patches/ipipe-core-3.8.13-arm-4.patch
```

- Apply post-patch.

```
$ patch -Np1 < ../xenomai-head/ksrc/arch/arm/patches/
raspberry/ipipe-core-3.8.13-raspberry-post-2.patch
```

3. Configure the kernel.

Once the patch is set on the new Linux kernel, it is time to configure it. The easiest way is visually with the menuconfig command.

- Create a build directory in Linux folder.

```
$ mkdir linux-rpi-3.8.y/build
```

At this point a .config file is needed. This file can be accesible from a third party resource or it can be configured entirely to meet desired standards.

- For a third party .config file:

```
$ cp downloaded_config_file linux-rpi-3.8.y/build
/.config
```

- Create personal .config file:

```
$ make mrproper
$ make ARCH=arm O=build menuconfig
```

#### 4. Compile kernel.

- To compile the new kernel:

```
$ make ARCH=arm O=build CROSS_COMPILE=/home/$USER
/cross/tools/arm-bcm2708/
arm-bcm2708hardfp-linux-gnueabi/bin/
arm-bcm2708hardfp-linux-gnueabi-
```

- Install modules.

```
$ make ARCH=arm O=build INSTALL_MOD_PATH=dist
modules_install
```

- Install headers:

```
$ make ARCH=arm O=build INSTALL_HDR_PATH=dist
headers_install
```

#### 5. Install kernel.

Compilation can take approximately from fifteen minutes to an hour. Errors during compilation are normally produced by enabling components on the menuconfig window that are not supported by the Raspberry Pi Hardware. In this case the process must be restarted.

- Transfer the kernel image If compilation was sucessful, find the kernel image under:

```
$ linux-rpi-3.8.y/build/arch/arm/boot/
```

Rename "Image" file to "kernel.img" and paste into /boot/ directory of the SD card. Replace existing. Finally, copy the contents of:

```
$ linux-rpi-3.8.y/build/dist
```

into the OS in the SD card under /lib/modules.

## 6. Install Xenomai API on Raspberry Pi.

The next step is to start up the Raspberry Pi and set up a networking interface to download xenomai.

```
$ git clone git://git.xenomai.org/xenomai-head.git
xenomai-head
```

- Install build-essentials and Xenomai.

```
$ sudo apt-get update build-essentials
$./configure
$ make
$ sudo make install
```

## 7. Test Xenomai installation.

```
$ cd /usr/xenomai/bin
$ sudo ./xeno latency
```

If this guide was correctly followed, a Xenomai real-time system integrated on a Raspbian OS and on the Raspberry Pi B+ should now be fully functional.

With a fully functional Embedded real-time system, the next step is to understand the API's that is now available. Xenomai provides several different API's. Under Xenomai and according to xenomai.org, "A Xenomai API can impersonate an existing traditional RTOS interface such as VxWorks or provide an original

programming interface for some particular purpose, such as RTDM. Each API makes Xenomai look like a different RTOS albeit all of them are based on the same common core. This is the reason why we call an implementation of such API, a Xenomai skin”.

Xenomai is based on an abstract RTOS core, usable for building any kind of real-time interface over a nucleus which exports a set of generic RTOS services. Any number of RTOS personalities called “skins” can then be built over the nucleus, providing their own specific interface to the applications by using the services of a single generic core to implement it. This project will use the “Native Skin”.

The following table shows the Native Skin API components and some of the most important functions of the most frequently used real-time resources in this project.

| Native Skin                                                                                                                                                                                                                                                                                                                    | task.c functions                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    | queue.c functions                                                                                                                                                                                                                                                                                                                   |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"> <li>• alarm.c</li> <li>• buffer.c</li> <li>• cond.c</li> <li>• event.c</li> <li>• heap.c</li> <li>• intrd.c</li> <li>• module.c</li> <li>• mutex.c</li> <li>• pipe.c</li> <li>• <b>queue.c</b></li> <li>• sem.c</li> <li>• syscall.c</li> <li>• <b>task.c</b></li> <li>• timer.c</li> </ul> | <ul style="list-style-type: none"> <li>• rt_task_create()</li> <li>• rt_task_start()</li> <li>• rt_task_suspend()</li> <li>• rt_task_resume()</li> <li>• rt_task_delete()</li> <li>• rt_task_yield()</li> <li>• rt_task_set_periodic()</li> <li>• rt_task_wait_period()</li> <li>• rt_task_set_priority()</li> <li>• rt_task_sleep()</li> <li>• rt_task_inquire()</li> <li>• rt_task_notify()</li> <li>• rt_task_set_mode()</li> <li>• rt_task_slice()</li> <li>• rt_task_send()</li> <li>• rt_task_recieve()</li> <li>• rt_task_reply()</li> </ul> | <ul style="list-style-type: none"> <li>• rt_queue_create()</li> <li>• rt_queue_delete()</li> <li>• rt_queue_alloc()</li> <li>• rt_queue_free()</li> <li>• rt_queue_send()</li> <li>• rt_queue_write()</li> <li>• rt_queue_receive()</li> <li>• rt_queue_read()</li> <li>• rt_queue_flush()</li> <li>• rt_queue_inquire()</li> </ul> |

Table 2.4: Xenomai native skin API and most used real-time functions.

## 2.5 Developer Tools

The flight control module is composed of several different interconnected units. Each unit has distinctive requirements depending on the hardware available, the functionality, etc. Each unit or module will have to have its own software development. There are three major components that require software development. As it was said before, depending on the modules functions, each unit will be developed using the tools that best fit the unit's deployment requirements.

### 2.5.1 Flight Control Module

The first and main unit is the flight control embedded module. This unit is built on a real-time Linux system. The real-time operating system provides an API developed in ANSI C. For this reason, using this same ANSI C programming language is the proper language to use to develop the flight control module. One of the best Software Development Kits or SDK's available is Eclipse. This SDK is open source and offers many helpful functions.

The flight control module is, as explained earlier, an embedded system without display or keyboard. In this scenario, eclipse will have to be used in remote (ssh) mode in order to code in the system. This is done in Eclipse by opening the *Remote System Explorer* and creating a new ssh connection to the embedded system.

Once the remote has been selected, the next step is to select “create remote project” on the main Eclipse window under the “project” folder. This will create a temporary local copy of the project.

When working remotely with Eclipse, the auto-complete functions and the debugging tools of the SDK are lost. This can become at some point a glitch, but it is easy to adapt to this circumstance.

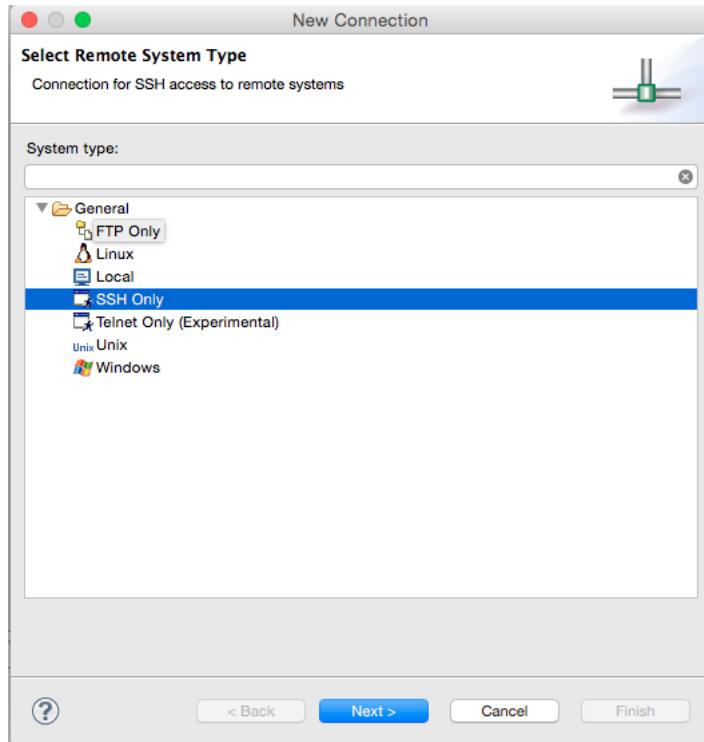


Figure 2.5: Eclipse ssh.

Due to the lack of debugger capabilities when working with Eclipse in remote mode, the best alternative to debugging is the use of the command line program GDB, which is a powerful command line tool used to debug programs. It is highly recommended to use this software to search for and resolve errors and conflicts of the program. It also allows the developer to view and analyze the contents of memory position, pointers and variables used.

Finally, this project has a considerable code size. For this reason, it is recommended that a version control system such as GIT be used. Github is used for this project. Github is an open source web-based repository.

To summarize, this list shows the needed tools to develop the flight control module:

- ANSI C programming Language.
- Eclipse SDK with remote project (SSH).
- GDB debugging tool.
- Git repositories for version control.

### 2.5.2 Autopilot Panel

The Autopilot panel is another important component of this project. It is responsible for sending the requested flight maneuver command to the flight control module. These commands are sent via a TCP connection, making this a remote unit. Because of this, it is appropriate for this unit to be able to execute on any platform.

The autopilot panel is a GUI (Graphical User Interface) with several different flight command buttons.

The programming language Qt and the Qt SDK offer excellent tools to create GUI's and multiplatform programs. Unlike the JAVA programming language, Qt does not rely on a Java Virtual Machine or anything similar to operate. It is normally compiled for the architecture of the local machine, but the same code can be transferred onto another architecture and then recompiled.

Qt is defined as: “a cross-platform application framework that is widely used for developing application software that can be run on various software and hardware platforms with little or no change in the underlying codebase”. The Qt language is based on C++ and uses the GCC compiler.

### 2.5.3 Quad-copter test-bed

The quad-rotor test-bed is used as a test-bed for the real quad-rotor. The test-bed is created due to the absence of the real-world quad-copter helicopter in the first stages of this project. In order to test communications of the *apc220\_comm interface*, the test-bed includes basic Arduino's read functions. This test-bed is designed using an Arduino board. In this case, the Arduino Mega 2560 to emulate the real quad-copter.

It is noteworthy that this project needs the existence of a real-world quad-copter to deploy its full functionality and capabilities.

The following is a list of the components included in the Arduino quad-copter test-bed.

- Arduino Mega 2560.
- MPU 6050 gyroscope and accelerometer.
- RF APC 220 radio transmitter.
- 4x20 LCD display.

Arduino's programs are developed with Arduino's software. As defined by the Arduino's development team: "the Arduino language is based on C/C++. It links against AVR Libc and allows the use of any of its functions."

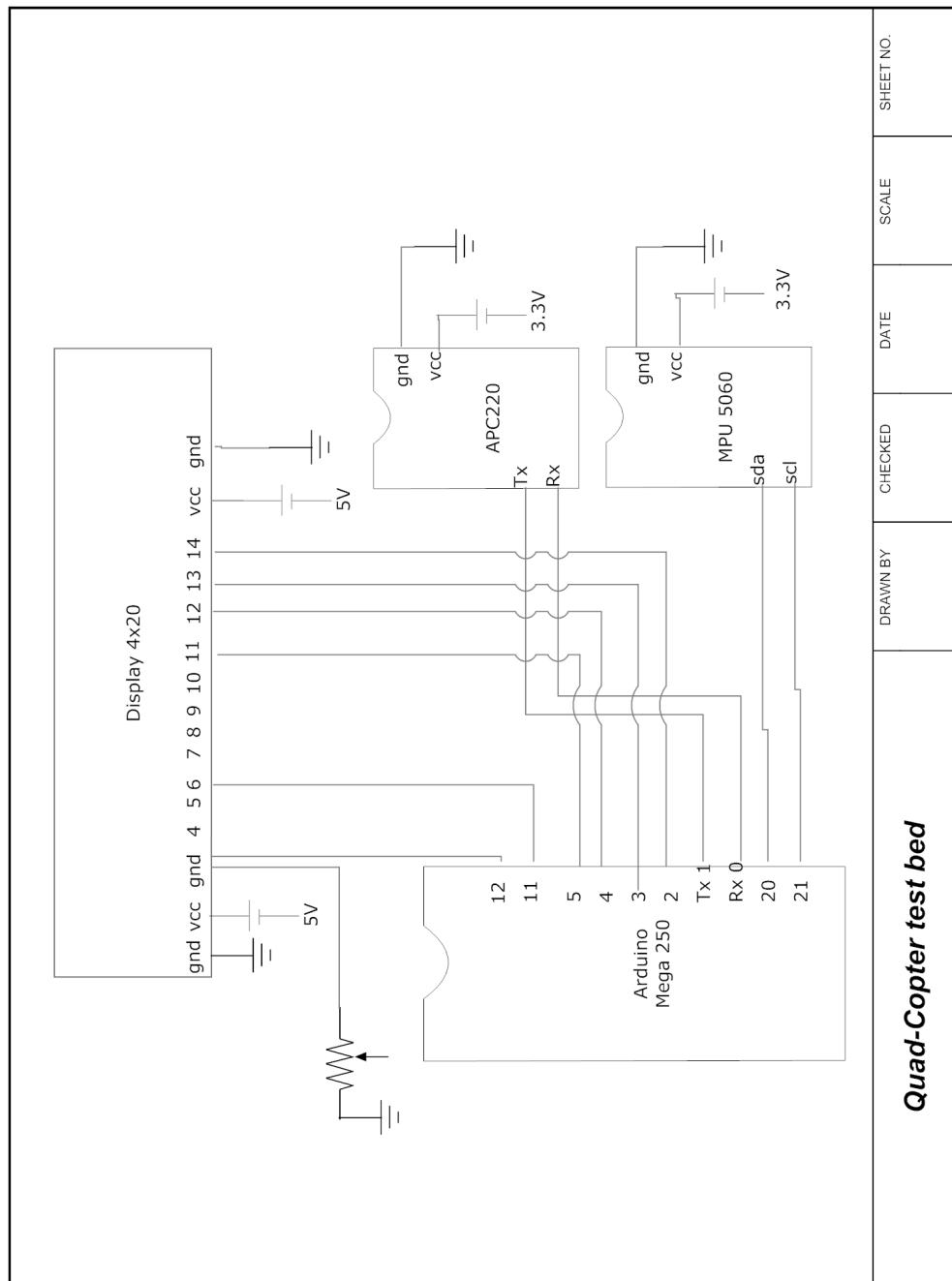


Figure 2.6: Quad-copter Test-bed.

---

## CHAPTER 3

### SOFTWARE DEVELOPMENT PROCESS

DESIGNING an Embedded Flight Control Module must include a detailed software development process in order to establish proper workflow guidelines.

- Project requirements.
- Project design and planning.
- Project construction.
- Project testing.
- Project deployment.

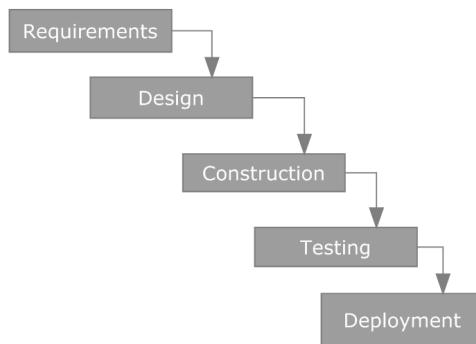


Figure 3.1: Software development process.

## 3.1 System requirements

- Modularization.

Each hardware or software component that interacts with the flight control module must have its own standalone program or interface which will be integrated into the global system. By creating this set of independent programs, debugging becomes more straightforward and it also enables testing each component's hardware separately for faults.

Modularization also helps the learning process required to fully understand how the components operate. Moreover, creating interfaces for each module helps the testing and deployment process. If a module needs to be worked on, the engineer will only have to deal with the corresponding interface, working with a small amount of code rather than working with the entire system.

Another advantage of modular design is that it offers the possibility of not having to execute all modules for the system to run correctly. For example the control panel module does not need to be connected in order to execute commands to the flight simulator or to read commands from the manual controller.

- Fault recovery.

Another key requirement for the flight control module is the capability for error recovery. As the system is required to run in a non stop cycle, it is safe to anticipate that system malfunctions or errors will occur at some point. For this reason, the system must be able to work around these issues. There are many ways to handle system faults. In some cases the system will enter panic mode, shut down and restart the faulty component.

- Reliability.

Another important requirement is reliability. The system must be able to interconnect all the components between each other with real time constraints. Each component will generate a massive flow of information that has to be available to the other modules in the system. Due to the large quantities of information being transmitted, it is acceptable to lose some data without incurring errors.

## 3.2 System design

The Flight Control system is composed of a central managing module called Autopilot and peripherals. Peripherals are modules that connect to the Autopilot as interfaces of external hardware or software units.

These are the current modules available:

- Autopilot or Flight Control Module.
- X-Plane 10 simulator.
- Sony DualShock®3 Wireless Controller.
- Autopilot Control Panel.

An interface is created for each hardware and software component that connects to the flight control module. Each interface is designed to do read and/or write operations.

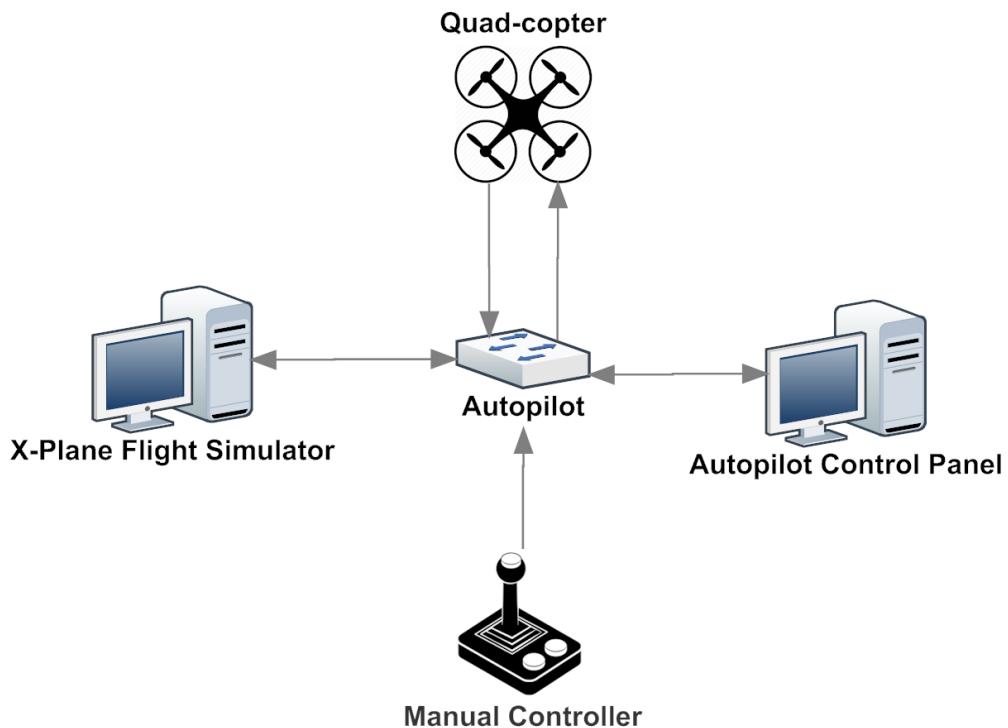


Figure 3.2: Flight Control Module Components.

Each module communicates with the Flight controller in different ways. A communication protocol is selected depending on the mission of each component.

1. X-Plane 10 UDP communication protocol.

X-Plane offers excellent advantages over other flight simulators. It allows developers to make use of flight values allowing them to either read or write to the simulator. X-Plane communicates via UDP. UDP is used because it does not expect acknowledgement messages from the receiver. The user can select from a wide range of values from the simulator, the ones that will be needed to be sent via UDP.

From every box selected in the X-Plane data window, the corresponding byte value data will be added to the datagram. When the user enables a

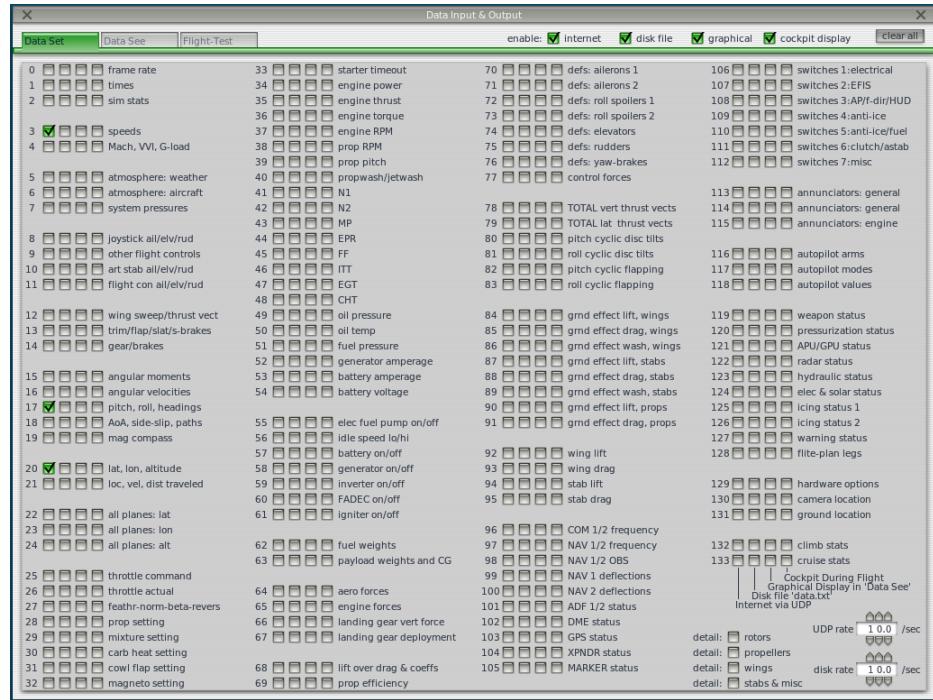


Figure 3.3: X-Plane Datagrams window.

box, X-Plane tries to read incoming UDP datagrams to verify if that value is being written. If this is not being received, X-Plane adds the value to the output datagram.

## 2. Autopilot Control Panel TCP communication protocol.

Contrary to the UDP datagrams sent and received by X-Plane, the control panel communicates via TCP protocol. TCP protocol is based on an acknowledge message system. When the sender sends a packet, it waits for a respond from the receiver. This is the most appropriate communication protocol for the Autopilot Control Panel. This module is only required to send one message when the user selects a flight maneuver. It could also be a critical message for which expecting an answer from the Flight Control module is necessary.

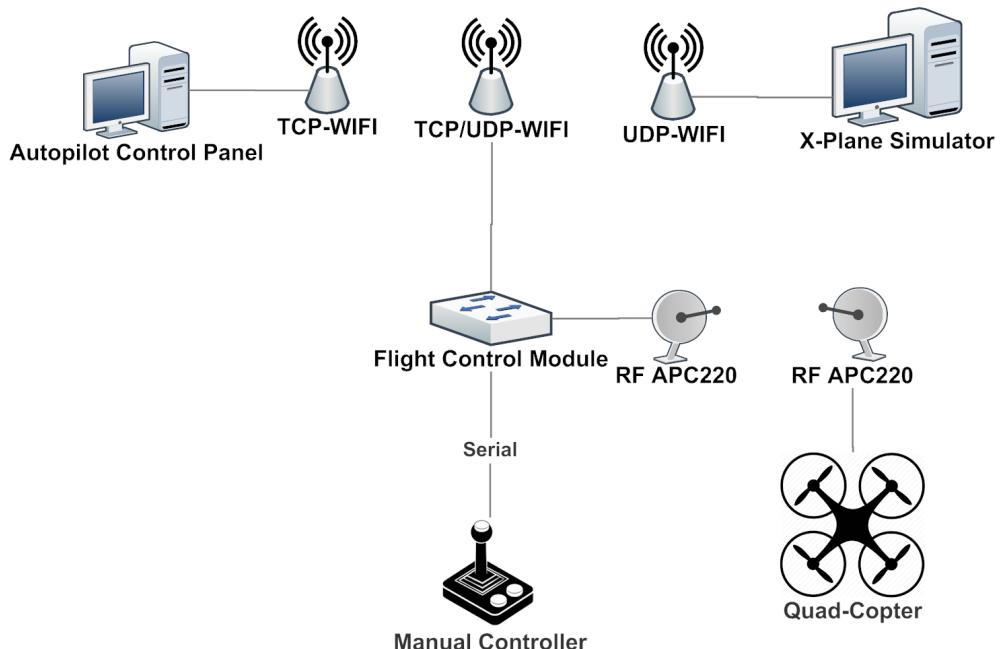


Figure 3.4: Modules communications.

### 3. Manual controller serial communications.

The Sony DualShock®3 Wireless Controller communicates with the Flight Controller via serial cable. It can use wireless capabilities (Bluetooth) but it will be connected through a USB cable for the initial stages of this project .

### 4. Quad-rotor APC220 radio frequency communications.

The selected radio unit for communicating with the real world quad-copter helicopter has a 1 km signal range. The unit is mounted on the Flight control module and the quad-copter. The module connected to the Flight control module is connected through a USB adapter, converting it into a serial communications protocol.

All communications with the flight control module are designed to be wireless, enabling the unit to be on-board the real world helicopter. The manual controller

will be the only wired device for these first stages of development. Once the communications are established, the system generates two types of messages to be sent between the components:

1. *Instruction messages.*
2. *Status messages.*

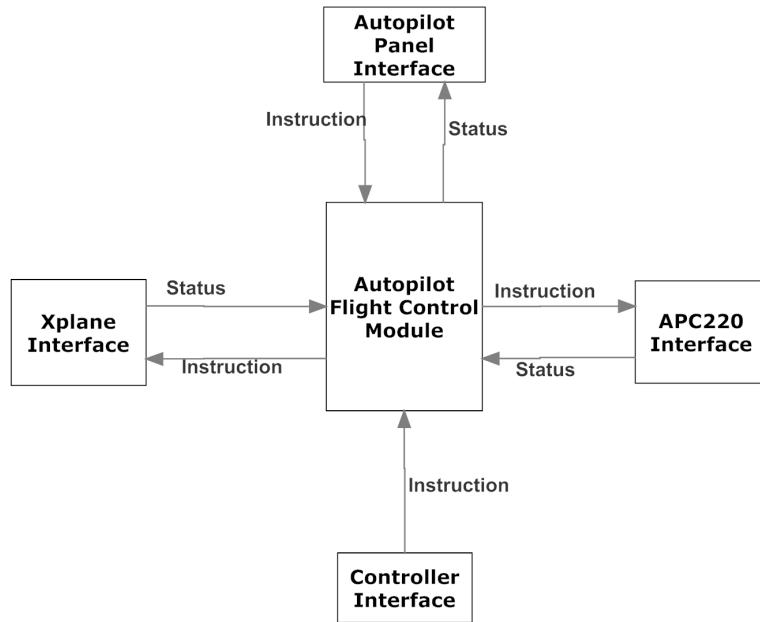


Figure 3.5: Interface messages.

The instruction messages are those in charge of ordering new flight commands. These messages are created by the Autopilot Control Panel and the manual Controller. They represent motor values that are sent to the requesting interface either the quad-copter or flight simulator. These values are typically sent in float format, but if it is necessary for transition requirements, they are changed into string format in order to be buffered.

On the other hand, status messages are responsible for transmitting the current flight status of the real-world quad-copter helicopter and the flight simulator drone.

These messages contain real time sensor values. They are generated by the real-world quad-copter helicopter or by X-Plane and used primarily by the autopilot control panel and also by the flight simulator when the system is in emulation mode.

## 3.3 System implementation

### 3.3.1 Autopilot control panel implementation

The autopilot control panel is a GUI window designed to provide the user with the needed controls “buttons” to perform automatic flight maneuvers.

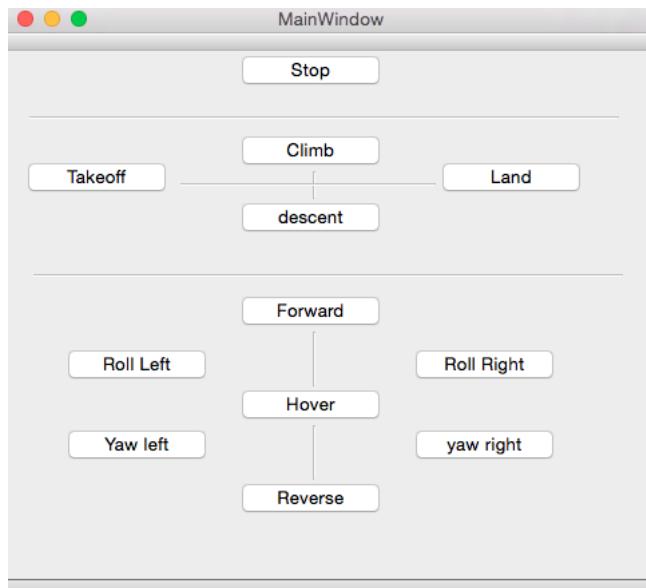


Figure 3.6: Autopilot control panel window.

Each button has a integer number assigned, which is sent via TCP to the flight control module. This integer on the receiving side executes the corresponding flight maneuver function.

Autopilot control panel file system:

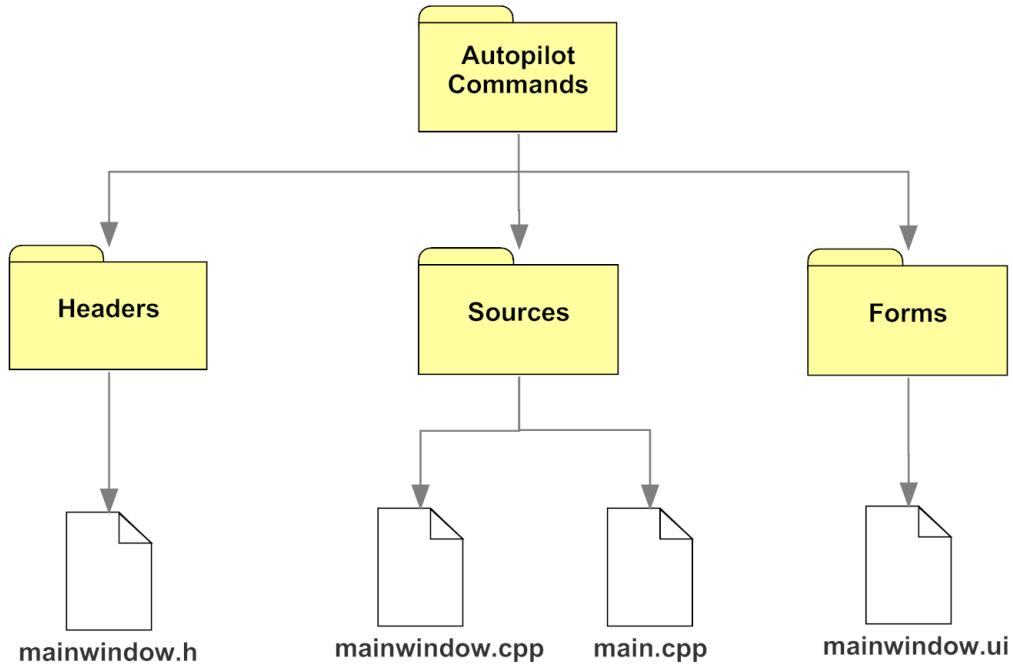


Figure 3.7: Autopilot control panel file system.

The **header** file **mainwindow.h** contains the information of the correspondence between button function and value to send via TCP.

|                  |                   |    |
|------------------|-------------------|----|
| <i>STOP</i>      | $\leftrightarrow$ | 0  |
| <i>TAKEOFF</i>   | $\leftrightarrow$ | 1  |
| <i>LAND</i>      | $\leftrightarrow$ | 2  |
| <i>HOVER</i>     | $\leftrightarrow$ | 3  |
| <i>CLIMB</i>     | $\leftrightarrow$ | 4  |
| <i>DESCEND</i>   | $\leftrightarrow$ | 5  |
| <i>ROLLELEFT</i> | $\leftrightarrow$ | 6  |
| <i>ROLLRIGHT</i> | $\leftrightarrow$ | 7  |
| <i>YAWLEFT</i>   | $\leftrightarrow$ | 8  |
| <i>YAWRIGHT</i>  | $\leftrightarrow$ | 9  |
| <i>FORWARD</i>   | $\leftrightarrow$ | 10 |
| <i>REVERSE</i>   | $\leftrightarrow$ | 11 |

There are two **source** files in this project, the **main.cpp** only contains the main function and the **mainwindow.cpp** source file which contains the functions for each button.

### • **main.cpp** functions

- int main(int argc, char \*argv[])

This function creates a mainwindow object and executes the visualization function to display the mainwindow window.

### • **mainwindow.cpp** functions

- MainWindow::MainWindow(QWidget \*parent)

Mainwindow constructor. Mainwindow extends from Qwidget.

- MainWindow:: MainWindow()

Mainwindow destruct.

- void MainWindow::on\_btnTakeoff\_clicked()

*Takeoff* button clicked function.

Given an IP address and a port, it opens a communication socket with the server. It then writes TAKEOFF  $\Leftrightarrow$  1 to the socket, waits for a respond from the server and closes the communication. If there is no response, it outputs an error message.

- void MainWindow::on\_btnStop\_clicked()

*Stop* button clicked function.

Given an IP address and a port, it opens a communication socket with the server. It then writes STOP  $\Leftrightarrow$  0 to the socket, waits for a respond from the server and closes the communication.

- void MainWindow::on\_btnLand\_clicked()

*Land* button clicked function.

Given an IP address and a port, it opens a communication socket with the server. It then writes LAND  $\Leftrightarrow$  2 to the socket, waits for a respond from the server and closes the communication. If there is no response, it outputs an error message.

- void MainWindow::on\_btnHover\_clicked()

*Hover* button clicked function.

Given an IP address and a port, it opens a communication socket with the server. It then writes HOVER  $\Leftrightarrow$  3 to the socket, waits for a respond from the server and closes the communication. If there is no

response, it outputs an error message.

- void MainWindow::on\_btnClimb\_clicked()

*Climb* button clicked function.

Given an IP address and a port, it opens a communication socket with the server. It then writes CLIMB  $\leftrightarrow$  4 to the socket, waits for a respond from the server and closes the communication. If there is no response, it outputs an error message.

- void MainWindow::on\_btnDescent\_clicked()

*Descend* button clicked function.

Given an IP address and a port, it opens a communication socket with the server. It then writes DESCEND  $\leftrightarrow$  5 to the socket, waits for a respond from the server and closes the communication. If there is no response, it outputs an error message.

- void MainWindow::on\_btnYawLeft\_clicked()

*Yaw left* button clicked function.

Given an IP address and a port, it opens a communication socket with the server. It then writes YAW LEFT  $\leftrightarrow$  8 to the socket, waits for a respond from the server and closes the communication. If there is no response, it outputs an error message.

- void MainWindow::on\_btnYawRight\_clicked()

*Yaw right* button clicked function.

Given an IP address and a port, it opens a communication socket with the server. It then writes YAW RIGHT  $\leftrightarrow$  9 to the socket, waits for a respond from the server and closes the communication. If there is no

response, it outputs an error message.

- void MainWindow::on\_btnRollLeft\_clicked()

*Roll left* button clicked function.

Given an IP address and a port, it opens a communication socket with the server. It then writes ROLL LEFT  $\Leftrightarrow$  6 to the socket, waits for a respond from the server and closes the communication. If there is no response, it outputs an error message.

- void MainWindow::on\_btnRollRight\_clicked()

*Roll right* button clicked function.

Given an IP address and a port, it opens a communication socket with the server. It then writes ROLL RIGHT  $\Leftrightarrow$  7 to the socket, waits for a respond from the server and closes the communication. If there is no response, it outputs an error message.

- void MainWindow::on\_btnForward\_clicked()

*Forward* button clicked function.

Given an IP address and a port, it opens a communication socket with the server. It then writes FORWARD  $\Leftrightarrow$  10 to the socket, waits for a respond from the server and closes the communication. If there is no response, it outputs an error message.

- void MainWindow::on\_btnReverse\_clicked()

*Reverse* button clicked function.

Given an IP address and a port, it opens a communication socket with the server. It then writes REVERSE  $\Leftrightarrow$  11 to the socket, waits for a respond from the server and closes the communication. If there is no

response, it outputs an error message.

- **mainwindow.ui**

This file contains the user interface specifications. It is here where the window is configured, where the buttons are fixed to the desired position and also where the button actions are created.

### 3.3.2 Flight control module implementation

The flight control module is designed to be able to send and receive a continuous flow of information between each connected component. Every read and write operation is executed with real-time **tasks**. In order to share information between real-time tasks, Xenomai provides a set of tools where these tasks can store or read information. Reviewing the Xenomai Native skin API table (table 2.4), two resources can be found to create this functionality. **Queue.c** and **pipe.c**. Both of these files provide the tools necessary to communicate real-time tasks.

- A message pipe is a two-way communication channel between Xenomai tasks and standard Linux processes using regular file I/O operations on a pseudo-device. Pipes can be operated in a message-oriented fashion so that message boundaries are preserved and also in byte streaming mode from real-time to standard Linux processes for optimal throughput. Xenomai tasks open their side of the pipe using the `rt_pipe_create()` service. Standard Linux processes do the same by opening one of the `/dev/rtpN` special devices, where N is the minor number agreed upon between both ends of each pipe.
- Message queueing is a method by which real-time tasks can exchange or pass data through a Xenomai-managed queue of messages. Messages can vary in length and be assigned different types or usages. A message queue can be created by one task and used by multiple tasks that send and/or

receive messages to the queue. A message queue object allows multiple tasks to exchange data through the use of variable-sized messages. A message queue is created empty. Message queues can be local to the kernel space or shared between kernel and user-space.

The main difference between the **pipe** and the **queue** is that pipes are designed to communicate tasks one to one. In other words, there is one sender, one receiver and one message. On the other hand, queues are designed as an standalone real-time object. When a queue is created, any running task can write or read from the queue. Also and very importantly, it stores information. When a task writes new data into a queue, this value remains in the queue until it is read by another task. This is very important for the correct operations of this project, because a task that needs to read from a queue might read values in a slower or faster manner than the writer. By using queues it is assured that if the reader is slower, it will always have a value to read.

The drawback that queues have, is that they require more memory and cpu processing than pipes, but they are still found to best fit this project.

With the use of queues, the messages sent between the interfaces can be now established as show in the next figure.

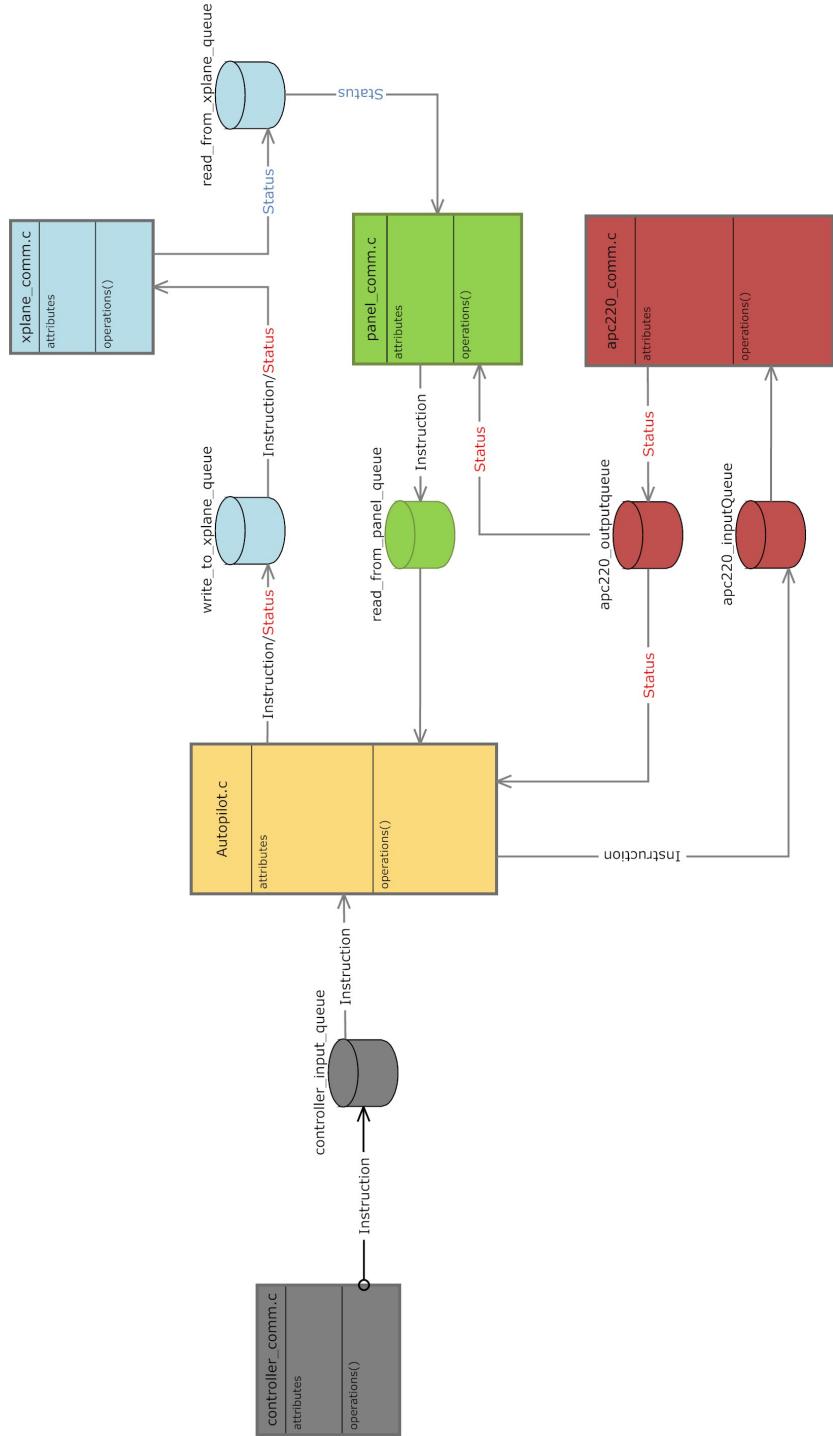


Figure 3.8: Flight control module implementation.

The flight control module's primary mission is to interconnect all four interfaces. Each interface corresponds to a hardware device or software unit.

The relation between the external devices and the interfaces is shown in the next figure.

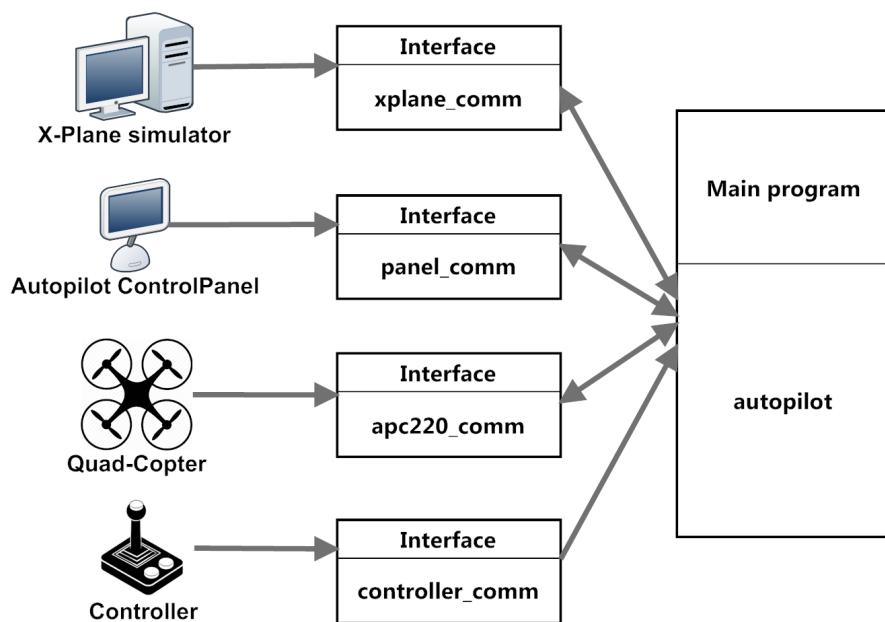


Figure 3.9: Flight control module interfaces.

### 3.3.2.1 Interfaces priorities protocol

Each interface is created as a standalone program, capable of providing full functionality to the connected device thanks to a test program included on every interface program source folder.

These interfaces must also follow several priority protocols in order to avoid possible conflicts between communications and flight operations. The possible scenarios are:

- Quad-rotor (apc220.comm), X-Plane flight simulator (xplane.comm) and manual controller (controller.comm) active: In this case all status messages will be generated by the apc220.comm interface and read by xplane.comm interface emulating the actual flight condition of the real-world quad-rotor helicopter in the flight simulator. All instruction messages will be generated by the controller.comm interface and sent to the apc220.comm interface.
- Quad-rotor (apc220.comm), X-Plane flight simulator (xplane.comm) and autopilot control panel (panel.comm) are active: This is only possible after the manual controller or controller.comm interface become disabled, as it holds a higher priority over the panel.comm interface. In this case, the apc220.comm sends status messages to be read by the xplane.comm and the panel.comm interfaces. All the instruction messages are generated by the panel.comm interface and sent to the apc220.comm interface.
- Quad-rotor (apc220.comm) active, X-Plane flight simulator (xplane.comm) inactive and manual controller (controller.comm) active: In this scenario the entire system acts as a remote controlled unit. The only function for the system “at this point” is to manually pilot the real-world quad-copter helicopter. In this case, the apc220.comm interface still generates status messages but are not read by any other interface. Again as in the first scenario, the controller.comm interface generates the instruction messages that are sent to the apc220.comm interface.
- Quad-rotor (apc220.comm) active, X-Plane flight simulator (xplane.comm) inactive and autopilot control panel (panel.comm) active: This scenario is similar to the previous one. The apc220.comm interface generates status messages and are read by the panel.comm interface. Again as in the second scenario, the panel.comm interface generates the instruction messages that are sent to the apc220.comm interface.

- Quad-rotor (apc220\_comm) inactive, X-Plane flight simulator (xplane\_comm) and manual controller (controller\_comm) active: In this new case scenario the xplane\_comm is now in charge of generating the status messages in absence of the real-world quad-copter helicopter. These messages are not read by any other interface. The instruction messages are created by the controller\_interface and sent to the xplane\_comm interface. This interaction is very similar to playing a video-game.
- Quad-rotor (apc220\_comm) inactive, X-Plane flight simulator (xplane\_comm) and autopilot control panel (panel\_comm) active: In this case scenario, the xplane\_comm is in charge of generating the status messages in absence of the real-world quad-copter helicopter. These messages are read by the panel\_comm interface. The instruction messages are also created by the panel\_interface and sent to the xplane\_interface.
- There are two final situations that are not relevant. This is with the absence of X-plane and the absence of the real-word quad-rotor. In this case, either one or both of the controllers (manual and autopilot) can be active, but nothing will happen, making this scenario useless.

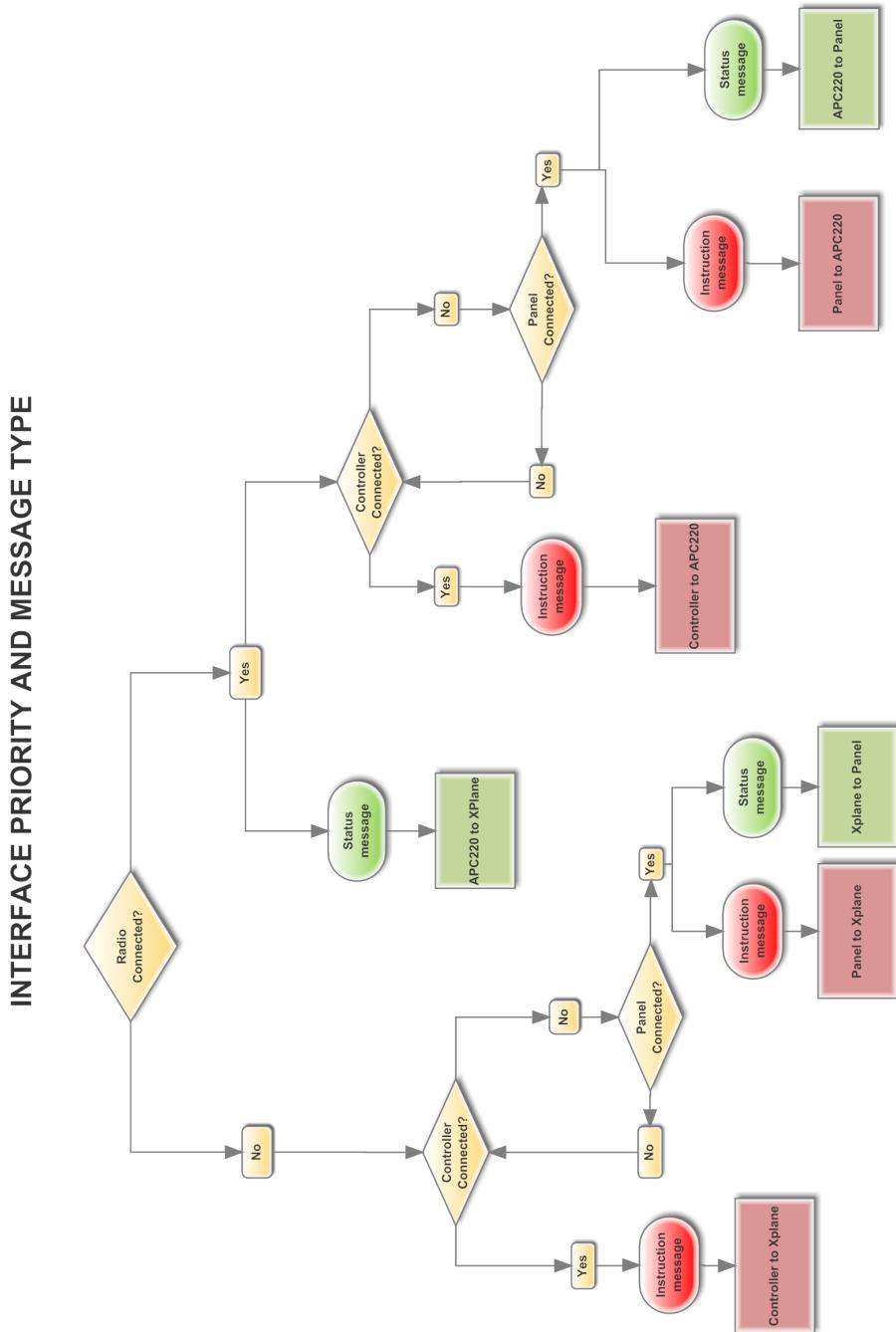


Figure 3.10: Interface priority and message type.

### 3.3.2.2 Flight control module file system

The flight control project is divided into four standalone programs and a fifth program that executes and interconnects the other four. This is a global view of the flight control module program file system.

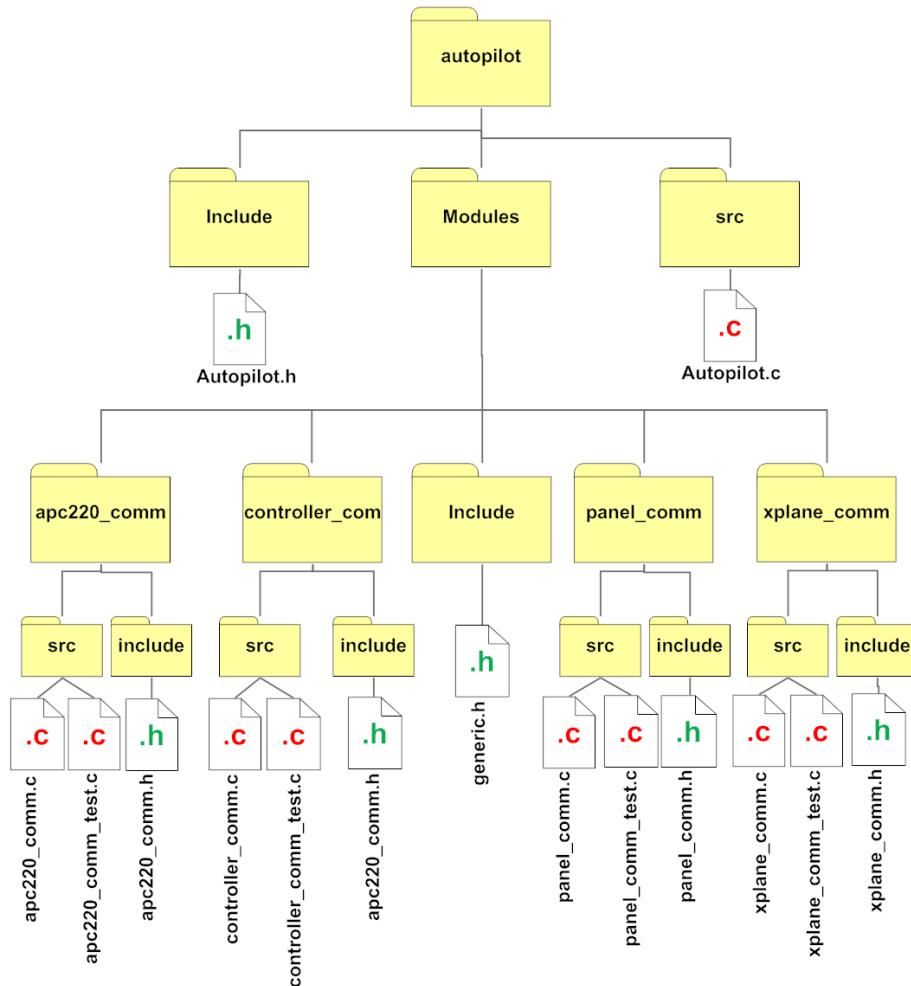


Figure 3.11: Flight control program file system.

### 3.3.2.3 Real time tasks and queues definitions

Before initiating an in depth explanation of the modules and their functions, a brief explanation of real time tasks and real time queues is important to achieve a better understanding of the system's behavior.

- Real Time Task

There are two necessary steps to have a task executed: **Task Create** and **Task Start**. Creating a task does not automatically start the task. Creating a task reserves the address of a task descriptor that Xenomai will use to store the task-related data. Starting a task involves scheduling it for the first time. The “create-start” mechanism could be compared to a “ready-go!” procedure.

- Real Time Queue

Contrary to real time tasks, queues only have to be created in order to start functioning. The “create queue” function creates a message queue object that allows multiple tasks to exchange data through the use of variable-sized messages. A message queue is created empty. Message queues can be local to the kernel space or shared between kernel and user-space.

It is now time to initiate an in depth study of each module or interface to better understand how the system works as a unit. The first module is the real-world quad-rotor interface (apc220\_comm). This unit is in charge of communicating the quad-rotor and the embedded system via radio by reading and writing data. It is a standalone program capable of performing these functions without the flight control module in operation. As seen in figure 16, the apc220\_comm program is composed of 2 source files and 2 header files.

### 3.3.2.4 Interface apc220\_comm

- apc220\_comm.c

This source file contains all the functions required to read and write to the serial port connected to the radio transmitter. It creates the needed tasks and queues to perform its operations.

- apc220\_comm\_test.c

This file creates a standalone test application that executes the functions in apc220\_comm.c.

- apc220\_comm.h

The header files contains the declaration of constants and function definitions.

- generic.h

The generic.h header file is, as its name suggests, an integral part used by all other modules that compose the system.

#### **apc220\_comm.c functions:**

- int apc220\_comm\_init()

Initialization function. It creates three real-time tasks and one real time queue.

- Task: apc220\_connect\_task
- Task: apc220\_read\_task
- Task: apc220\_write\_task
- Queue: acp220\_inputQueue

After creating the real time objects, it only starts “apc220\_connect\_task” task.

- void apc220\_connect\_task\_func(void \*arg)

This task tries to open a file descriptor of a serial port in a loop. If this operation is successful and connection is established, it configures the serial parameters. It then starts the remaining tasks:

- Task: apc220\_read\_task
  - Task: apc220\_write\_task
- void apc220\_read\_task\_func(void \*arg)

This task function is very basic. It reads the incoming buffer from the file descriptor created by the “apc220\_connect\_task\_func” task and passes it without any treatment directly to the queue previously created by the “apc220\_comm\_init()” function. It reads the incoming data from the quadcopter and makes this information available to the entire system.

- void apc220\_write\_task\_func(void \*arg)

As opposed to the “apc220\_read\_task\_func” task, this task reads the information available in a real time queue created by autopilot.c file. The information here corresponds to motor values. This information is then sent (writes) to the same file descriptor created on “apc220\_comm\_init()” function.

### 3.3.2.5 Interface controller\_comm

The next module is the controller\_comm interface. This unit is responsible for reading the actions taking place on the manual controller by the user and translating these movements into integer values that indicate motor speed values with a range from 0 to 100. The controller initially has a range of values between -32767 to 32767 on each axis which are not valid motor speed values.

The controller\_comm interface is also a standalone program capable of performing these functions without the flight control module in operation. As seen in figure 16, the controller\_comm program is composed of 2 source files and 2 header files.

- controller\_comm.c

This source file contains all the functions required to read from a serial port connected manual controller. It creates the needed tasks, queues and functions to perform its operations.

- controller\_comm\_test.c

This file creates a standalone test application that executes the functions in controller\_comm.c.

- controller\_comm.h

The header file contains the declaration of constants and function definitions.

- generic.h The generic.h header file is, as its name suggests, an integral part used by all other modules that compose the system.

#### **controller\_comm.c functions:**

- int controller\_comm\_init() Initialization function. It creates one real time task and one real time queue.
  - Task: controller\_read\_task

- Queue: controller\_inputQueue

After creating the real time objects, it starts “controller\_read\_task” task.

- void controller\_read\_task\_func(void \*arg) This task tries to open a file descriptor of a serial port in a loop. If this operation is successful and connection is established, it configures the serial parameters. When the connection with the manual controller has been established, it reads the input buffer from the file descriptor associated to the serial port. The information read belongs to the value that all four axis are providing. These values range from -32767 to 32767 and have to be translated into a 0 to 100 value range. To do so, this task calls the function responsible for the translation: “xplane\_controller\_motors,” sending the four received values as parameters.
- int xplane\_controller\_motors(int X,int Y,int Z,int R) This function receives the four controller values corresponding to X, Y, Z and R axis. The function creates flight maneuvers by translating the incoming values into motor values. Depending on what axis is changing values at the time, it will generate a different maneuver. The most important value that has to be taken into consideration is the *Center Null Zone “0”*. In order to make the flight control easier for the pilot, this value has to be changed into a recommended *hover* value. Finally when the new motor speed values have been created, this function writes these new values into the real-time queue “controller\_inputQueue”, making the values available for the rest of the flight control module.

### 3.3.2.6 Interface xplane\_comm

The next module is the xplane\_comm interface. This unit is responsible for reading and writing data to and from the X-Plane simulator. During the reading process, it receives byte format values via UDP and translates them into float numbers. The writing process performs the opposite operation, it transforms float

numbers into byte values and send them via UDP to the flight simulator. The xplane\_comm interface is also a standalone program capable of performing these functions without the flight control module in operation. As seen in figure 16, the xplane\_comm program is composed of 2 source files and 2 header files.

- **xplane\_comm.c**

This source file contains all the functions required to read incoming X-Plane data. It transforms the data received into valid values for the other components and for the “writing to X-Plane” process. It also accepts data from the system and transforms it into X-Plane valid values before sending the data to the flight simulator. To do this, it creates the required tasks, queues and functions.

- **xplane\_comm\_test.c**

This file creates a standalone test application that executes the functions in xplane\_comm.c.

- **xplane\_comm.h**

The header file contains the declaration of constants and function definitions.

- **generic.h**

The generic.h header file is, as its name suggests, an integral part used by all other modules that compose the system.

### **xplane\_comm.c functions:**

- **int xplane\_comm\_init()**

Initialization function. It creates two real time tasks and one real time queue.

- Task: xplane\_read\_task

- Task: xplane\_write\_task
- Queue: read\_from\_xplane\_queue

After creating the real time objects, it starts the created tasks “xplane\\_read\\_task” and “xplane\\_write\\_task”.

- void xplane\_write\_task\_func(void \*arg)

This task tries to open a file descriptor for UDP socket in a loop. If this operation is successful and connection is established, it configures the UDP socket parameters. Once completed, the task reads the values available on the output queue created by autopilot.c “write\_to\_xplane\_queue”. The information available is a set of float numbers. The position of the numbers in this set corresponds to a certain flight value. This set is defined as a “struct” type.

|                |                   |                           |
|----------------|-------------------|---------------------------|
| <i>float1</i>  | $\leftrightarrow$ | <i>Xaxisaccelerometer</i> |
| <i>float2</i>  | $\leftrightarrow$ | <i>Yaxisaccelerometer</i> |
| <i>float3</i>  | $\leftrightarrow$ | <i>Zaxisaccelerometer</i> |
| <i>float4</i>  | $\leftrightarrow$ | <i>Roll</i>               |
| <i>float5</i>  | $\leftrightarrow$ | <i>Pitch</i>              |
| <i>float6</i>  | $\leftrightarrow$ | <i>Yaw</i>                |
| <i>float7</i>  | $\leftrightarrow$ | <i>Motor1</i>             |
| <i>float8</i>  | $\leftrightarrow$ | <i>Motor2</i>             |
| <i>float9</i>  | $\leftrightarrow$ | <i>Motor3</i>             |
| <i>float10</i> | $\leftrightarrow$ | <i>Motor4</i>             |

Each float number equals to four byte values. The byte values are set in order and an index number is added at the beginning of the output string buffer. The index value tells X-Plane what data the incoming value corresponds to. The float values are transformed into 4 byte values using the **Union**

**struct**<sup>1</sup> C function.

The resulting output string buffer has the following format:

|              | header | Index    | Data                                                                |
|--------------|--------|----------|---------------------------------------------------------------------|
| Definition   | 5bytes | 4bytes   | 32bytes, 8 groups of 4 bytes                                        |
| Byte values  | “DATA” | 0 0 0 16 | 0,0,0,0-0,0,0,<br>0,0,0,0-0,0,0,<br>0,0,0,0-0,0,0,<br>0,0,0,0-0,0,0 |
| Float values |        | 16       | 0,0,0,0,0,0,0,0                                                     |

Table 3.1: X-Plane byte values format.

The following is an example of a normal float value transformed into a byte value. Additionally, for X-Plane the value -999 is interpreted as a value that should not be modified. In other words, when it receives this value it does not act on it.

$$\begin{aligned} 1 &\leftrightarrow 0, 0, 128, 63 \\ -999 &\leftrightarrow 0, 192, 121, 196 \end{aligned}$$

After the desired conversion is completed and the byte values buffer has been established, the task sends the buffer through the socket, reaching X-plane as a UDP datagram.

- void xplane\_read\_task\_func(void \*arg)

This task, as the xplane\_read\_task did, tries to open a file descriptor for UDP socket in a loop. If this operation is successful and connection is established, it configures the UDP socket parameters. When the UDP connection is created, it reads the X-Plane input. The incoming datagrams

---

<sup>1</sup>A union is a special data type available in C that enables you to store different data types in the same memory location. You can define a union with many members, but only one member can contain a value at any given time. Unions provide an efficient way of using the same memory location for multi-purpose.

are composed of a buffer of byte values. The task sends the byte values to the “byte\_to\_float\_func” which translates the values into float numbers. With the new converted float values, the task writes these values onto the “read\_from\_xplane\_queue” queue.

- `int byte_to_float_func(uint8_t xplaneByteInputValues[XPLANE_BASIC_INDEXES_USED][UDP_DATA_TOTAL_SIZE+1])`

This is the function in charge of transforming byte values into float values. The function receives a matrix of X-plane byte values. Each row corresponds to an index from the X-plane UDP data selection. It uses an algorithm that runs through the matrix with two nested “for” functions. As it gathers 4 byte values it transforms each value into a float number. It finally saves the float values in another matrix. This is a dynamic function. It can translate as many X-Plane indexes as necessary.

### 3.3.2.7 Interface panel\_comm

The next and final module is the panel\_comm interface. This interface is responsible for creating a TCP connection and receiving data from the Autopilot control panel, which was previously detailed. The data received indicates an automated flight maneuver procedure. This interface is also responsible for creating these flight maneuvers by analyzing the status messages that have been previously sent to the corresponding queues by the xplane\_comm interface or the apc220\_comm interface and depending on the flight status it creates the proper motor instructions which are sent into the system, making them available to the other interfaces.

The panel\_comm interface is also a standalone program capable of performing these functions without the flight control module in operation. As seen in figure 16, the panel\_comm program is composed of 2 source files and 2 header files.

- `panel_comm.c`

This source file contains all the functions required to read the incoming

Autopilot control panel commands through a TCP protocol connection. Then, it generates motor values which triggers the correct flight maneuver. To do this, it creates one task for each command and an infinite loop task, which acts as a TCP server constantly listening for new input messages.

- `panel_comm-test.c`

This file creates a standalone test application that executes the functions in `panel_comm.c`.

- `panel_comm.h`

The header files contains the declaration of constants and function definitions.

- `generic.h`

The `generic.h` header file is, as its name suggests, an integral part used by all other modules that compose the system.

#### **panel\_comm.c functions:**

- `int panel_comm_init()`

Initialization function. It creates one real time tasks and one real time queue.

- Task: `panel_tcp_test_incoming_task`
- Queue: `read_from_panel_queue`

After creating the real time objects, it starts the real time task created. “`panel_tcp_test_incoming_task`”.

- `void panel_tcp_test_incoming_task_func(void *arg)`

This task creates a TCP socket and it remains permanently listening to this socket for incoming data from the autopilot control panel. The data sent by the control panel and received by the `panel_comm` interface is a

set of previously defined integers. Each integer received corresponds to a different flight command. Also with each integer received, a new task is created and started. To avoid problems with overlapping instructions, this task prior to starting a demanded task, terminates and deletes any other task that could be generating an old flight instruction. This is done with the “delete\_running\_tasks” function. This task creates and starts eleven tasks, as ordered.

- Task: takeoff\_task
- Task: land\_task
- Task: hover\_task
- Task: climb\_task
- Task: descend\_task
- Task: yaw\_left\_task
- Task: yaw\_right\_task
- Task: roll\_left\_task
- Task: roll\_right\_task
- Task: forward\_task
- Task: reverse\_task
- Queue: read\_from\_panel\_queue

This task can also stop all other tasks executed if the “stop” command is received from the autopilot control panel. To do this, it calls the “delete\_running\_tasks function”, which terminates all other running tasks.

- void takeoff\_task\_func(void \*arg)

This function reads the contents of the queues created by the xplane-comm interface “read\_from\_xplane\_queue” or the apc220\_comm interface

“acp220\_inputQueue”, depending on the priority status of the flight control module. The values read from the queue correspond to the flight status of the simulated or the real aircraft (roll and pitch, altimeter). Taking these values into consideration, it gives enough power to the motors to create lift-off by assigning the correct float values for each motor and then sending these values to the “read\_from\_panel\_queue” in order to make these values available to the other components of the system. The same amount of power is applied to all four motors to perform a correct lift-off. During the lift-off process, the altimeters are evaluated and when a minimum safe altitude is reached, it initiates a hover stand.

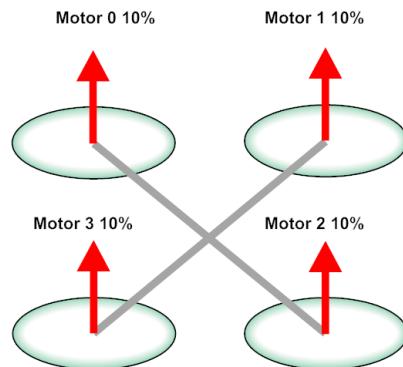


Figure 3.12: Automated takeoff procedure.

- void land\_task\_func(void \*arg)

This function reads the contents of the queues created by the xplane-comm interface “read\_from\_xplane\_queue” or the acp220\_comm interface “acp220\_inputQueue”, depending on the priority status of the flight control module. The values read from the queue correspond to the flight status of the simulated or the real aircraft (roll and pitch, altimeter). The task reduces the speed of the motor gradually depending on the altitude readings. As it approaches the ground, it creates a small lift increasing the power to the

motors to prevent hard landings. Thanks to the fall inertia, increasing the power will not create a climb procedure, it will only reduce the descent speed. The same amount of power is applied to all four motors to perform a correct descend and land procedure. During the landing process, the altimeter is evaluated and when it is on the ground, the motors are shutdown.

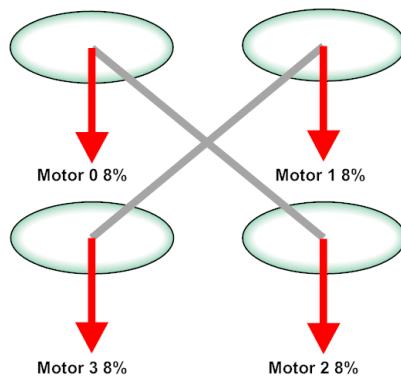


Figure 3.13: Automated landing procedure.

- `void hover_task_func(void *arg)`

This function reads the contents of the queues created by the xplane-comm interface “`read_from_xplane_queue`” or the apc220\_comm interface “`acp220_inputQueue`”, depending on the priority status of the flight control module. The values read from the queue correspond to the flight status of the simulated or the real aircraft (roll and pitch, altimeter). When the hover button is pressed, the task reads the current altitude and sets it as a threshold. Then, it gives the sufficient power to the motors depending on a valid error range. If it is too high, motors reduce speed, if it is too low, motors increase speed. This way, it creates an oscillation around the selected altitude creating a hovering situation.

- void climb\_task\_func(void \*arg)

Similar to the takeoff task, this task applies power to the engines, but with more intensity than the takeoff procedure. This task does not create a service ceiling for the aircraft, for which it is up to the pilot to know when to abort the climb maneuver. Contrary to the takeoff procedure, in order for this task to work, the quad-rotor must already be at altitude and not on the ground. This is done by evaluating the altitude from the corresponding queues.

- void descend\_task\_func(void \*arg)

Similar to the landing task, this task reduces the engine power, but on a higher degree than the landing procedure. This task doesn't control the altitude reading, for which it is up to the pilot to know when to abort the descend maneuver. In order to descend, this task creates the same low power settings to all four motors.

- void yaw\_left\_task\_func(void \*arg)

This task creates a yaw movement to the left. It rotates the quad-copter helicopter around its Z axis. It has to evaluate from the corresponding queues, the roll and yaw values from the simulated or real-world quad-copter to prevent the drone from going into steep roll or high or low pitch situations. In order to complete this maneuver, the task applies a slightly higher power to opposing motors, in this case the motors with odd numbers (1 and 3).

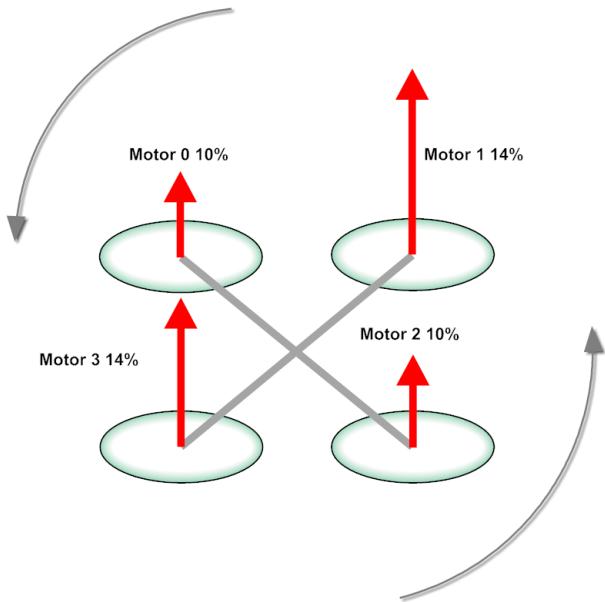


Figure 3.14: Automated yaw left maneuver.

- void yaw\_right\_task\_func(void \*arg)

This task creates a yaw movement to the right. It rotates the quad-copter helicopter around its Z axis. It has to evaluate from the corresponding queues, the roll and yaw values from the simulated or real-world quad-copter to prevent the drone from going into steep roll or high or low pitch situations. In order to complete this maneuver, the task applies a slightly higher power to opposing motors, in this case the motors with even numbers (0 and 2).

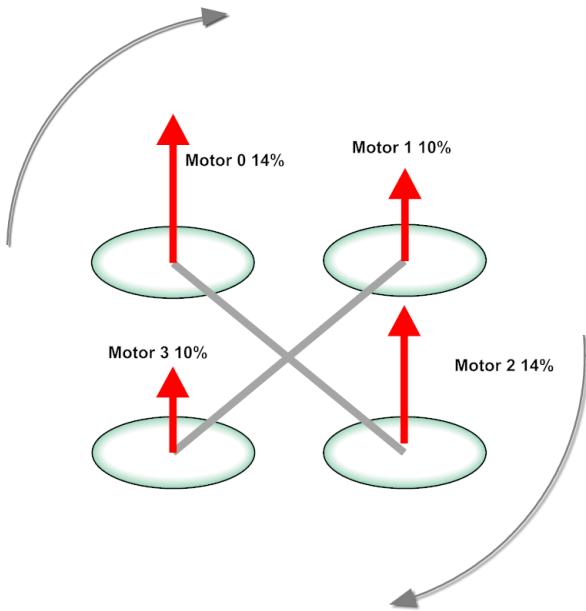


Figure 3.15: Automated yaw right maneuver.

- void roll\_left\_task\_func(void \*arg)

This task creates a roll movement to the left. It rotates the quad-copter helicopter around its longitudinal or X axis. It has to evaluate from the corresponding queues, the roll and yaw values from the simulated or real-world quad-copter to prevent the drone from going into a steep roll or high or low pitch situations. In order to complete this maneuver, the task applies a slightly higher power to inline motors, in this case the motors number (1 and 2).

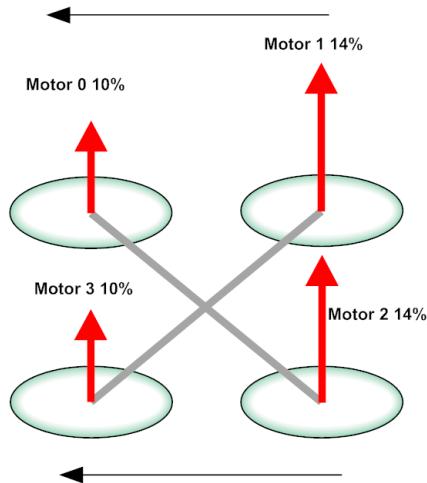


Figure 3.16: Automated roll left maneuver.

- void roll\_right\_task\_func(void \*arg)

This task creates a roll movement to the left. It rotates the quad-copter helicopter around its longitudinal or X axis. It has to evaluate from the corresponding queues, the roll and yaw values from the simulated or real-world quad-copter to prevent the drone from going into a steep roll or high or low pitch situations. In order to complete this maneuver, the task applies a slightly higher power to inline motors, in this case the motors number (0 and 3).

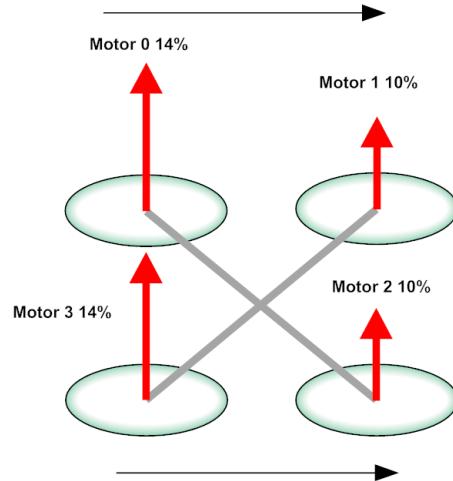


Figure 3.17: Automated roll right maneuver.

- void forward.task\_func(void \*arg)

This task creates a pitch movement. It pitches the quad-copter helicopter around its lateral or Y axis. It has to evaluate from the corresponding queues, the roll and yaw values from the simulated or real-world quad-copter to prevent the drone from going into a steep roll or high or low pitch situations. In order to complete this maneuver, the task applies a slightly higher power to inline motors, in this case the motors number (2 and 3).

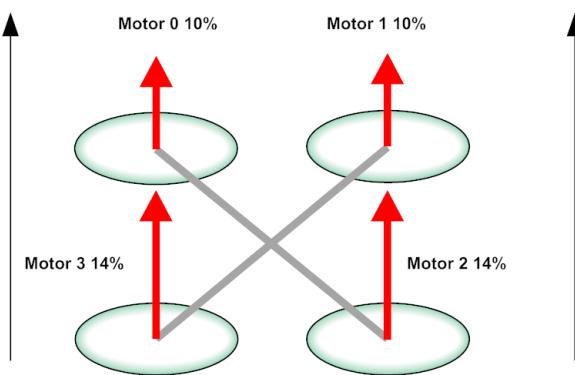


Figure 3.18: Automated forward maneuver.

- void reverse\_task\_func(void \*arg)

This task creates a pitch movement. It pitches the quad-copter helicopter around its lateral or Y axis. It has to evaluate from the corresponding queues, the roll and yaw values from the simulated or real-world quad-copter to prevent the drone from going into a steep roll or high or low pitch situations. In order to complete this maneuver, the task applies a slightly higher power to inline motors, in this case the motors number (0 and 1).

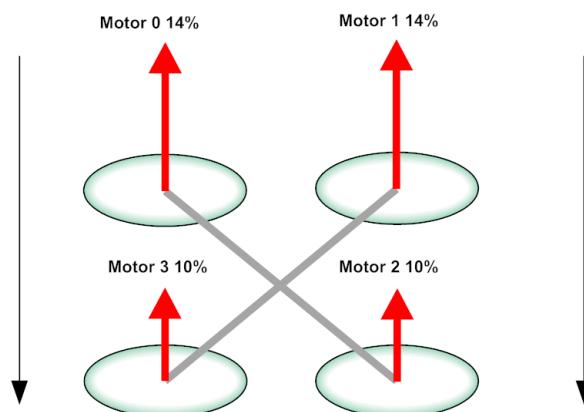


Figure 3.19: Automated reverse maneuver.

- void delete\_running\_tasks()

This function is called by the init\_panel() function every time a new task is going to start or when a stop message has been received from the autopilot control panel. It stops execution of any running maneuver task and then deletes it.

### 3.3.2.8 Flight control module; autopilot program

This is the final and most important unit of the system. Its function is to create the necessary intermediate message queues and to initialize all the other interfaces. It is also responsible for imposing the priority protocol of the embedded flight

control module and interconnecting all the interfaces through messages. As seen in figure 16, the panel\_comm program is composed of 1 source files and 2 header files.

- **autopilot.c**

This source file contains all the functions required to intercommunicate all of the other interfaces between each other. It is also responsible for managing incoming data by adopting the proper format for the requesting interface. At the same time, it is responsible for the priority designations for the system. This is the nucleus of the system.

- **autopilot.h**

The header files contains the declaration of constants and function definitions.

- **generic.h**

The generic.h header file is, as its name suggests, an integral part used by all other modules that compose the system.

### **autopilot.c functions:**

- **int panel\_comm\_init()**

Initialization function. It creates one real time tasks and two real time queue. It also calls all the initialization functions from all four interfaces.

- Task: read\_input\_queues\_task
- Queue: acp220\_outputQueue
- Queue: write\_to\_xplane\_queue

After creating the real time objects, it starts the real time task created tasks “read\_input\_queues\_task”.

- **void read\_input\_queues\_task\_func(void \*arg)**

This task connects with the queues created by the other interfaces following

the priority protocols. The basic function is to read from a queue the desired value, process the data and write the new information onto one of the created queues “acp220\_outputQueue” or “write\_to\_xplane\_queue”.

### 3.3.3 System compilation

The system is composed of 4 standalone independent programs. Each of these programs are compiled with their own scripting and Makefile files. Each Makefile is designed to create each module as a library. The headers and library files are installed in the operating system’s file system.

The following files are created:

- Header files location: /usr/local/include/pfc
  - apc220\_comm.h
  - controller\_comm.h
  - panel\_comm.h
  - xplane\_comm.h
  - generic.h
- Library files location: /usr/local/lib/pfc
  - libapc220\_commd.so
  - libcontroller\_commd.so
  - libpanel\_commd.so
  - libxplane\_commd.so

## 3.4 System testing and deployment

This project is created on an embedded system. This offers a wide range of possibilities where the system can be deployed. It was created so it could be

used as an independent unit, connected remotely to the real-world quad-copter helicopter or as an on-board system embedded on the quad-copter.

The credit card size Raspberry Pi makes these scenarios very easy to accomplish. On a real-world scenario the objective is to enable the human controller to pilot the quad-rotor remotely with the help of the flight simulator emulation. For example: The human controller can pilot the drone without visual contact with the quad-rotor through a computer. This is achieved because the project is designed to emulate on X-Plane flight simulator the exact same flight conditions as the real-world quad-copter.

The quad-copter can alter its flight pattern due to weather conditions, but the sensors readings (accelerometer, gyroscope, etc) are sent to the flight simulator which recreates the exact movement suffered by the quad-copter. For example: If wind gusts reach the real-world drone, the g-forces that are generated on the drone are sent to X-Plane which, at the same time, will suffer the same consequences as the real quad-copter.

### 3.5 Project schedule (Gantt chart)

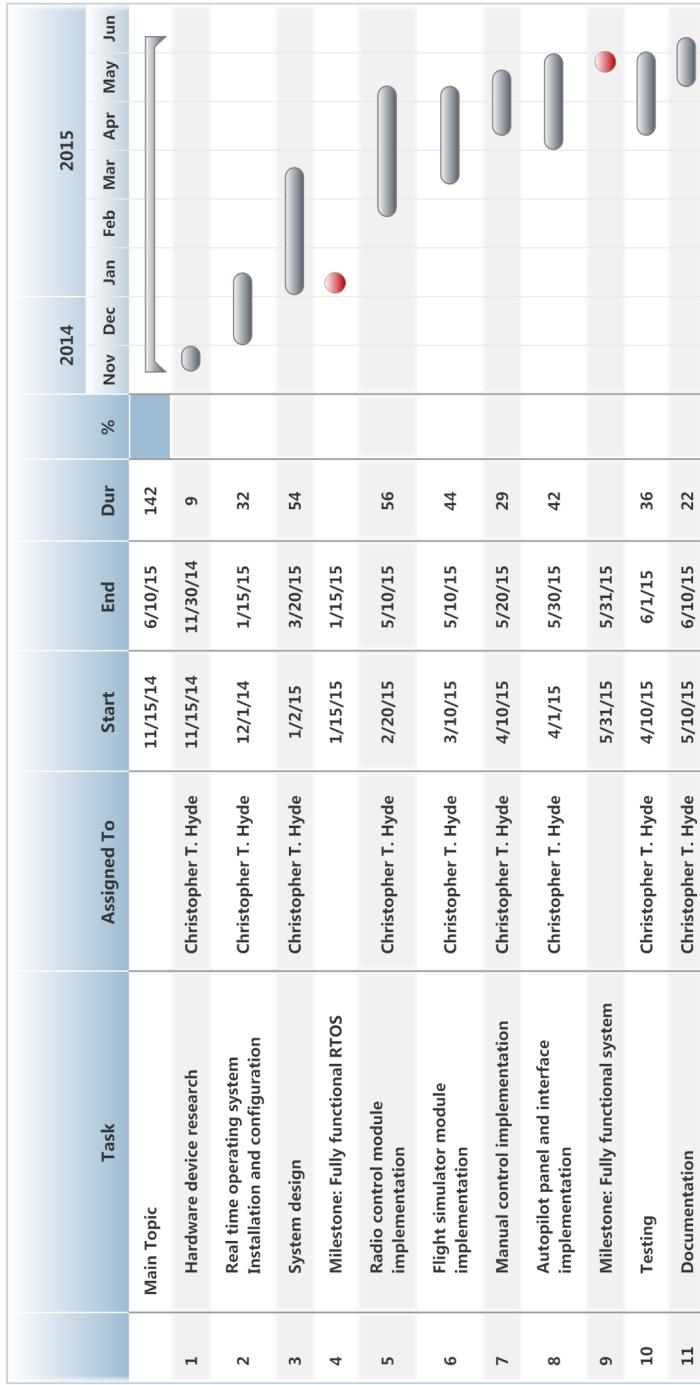


Figure 3.20: Gantt chart.

## 3.6 Project estimate costs

The costs are divided into two groups.

1. Hardware costs.
2. Workforce costs.

### 3.6.1 Hardware costs

| Device                                        | Quantity | Unit cost |
|-----------------------------------------------|----------|-----------|
| Raspberry 1 model B+                          | 1        | 35€       |
| Micro SD 8gb                                  | 1        | 8€        |
| Apc220 radio antennas                         | 2        | 40€       |
| PlayStation®3 DUALSHOCK®3 wireless controller | 1        | 25€       |
| Wifi dongle                                   | 1        | 9€        |
|                                               | Total:   | 117€      |

### 3.6.2 Workforce costs

The project must be composed of a team of a minimum of 5 members in order to obtain the best performance and produce a high quality software.

- Software developer 1  $\Leftrightarrow$  Apc220 interface.
- Software developer 2  $\Leftrightarrow$  Xplane interface.
- Software developer 3  $\Leftrightarrow$  Panel interface.
- Software developer 4  $\Leftrightarrow$  Controller interface.
- Software developer 5 (Project manager)  $\Leftrightarrow$  Modules integration.

|                      | Workforce costs |              |          |
|----------------------|-----------------|--------------|----------|
|                      | Hours           | Hourly costs | Total    |
| Software developer 1 | 448             | 25           | 11,200   |
| Software developer 2 | 352             | 25           | 8,800    |
| Software developer 3 | 232             | 25           | 5,800    |
| Software developer 4 | 336             | 25           | 8,400    |
| Software developer 5 | 1,136           | 35           | 39,760   |
| Total                | 2,504           |              | 73,960 € |

Figure 3.21: Workforce costs.

---

## CHAPTER 4

### USER GUIDE

P

LEASE follow these next steps to execute the system and all its components:

1. Connect the Raspberry Pi to 5V power supply.
2. Wait for system start up (approximately 1 minute).
3. Make sure IP address is configured on the Raspberry Pi. If the system is initiated in a new network, the IP needs to be configured. To configure the new network, shutdown the Raspberry PI, remove the micro SD card from the raspberry PI and insert it in a Linux OS to acces Raspbian's file system. Follow the next steps:

- `$ sudo vi /etc/network/interfaces`
- Add the following to `/etc/network/interfaces`:

```
allow-hotplug wlan0
iface wlan0 inet dhcp
wpa-conf /etc/wpa_supplicant/wpa_supplicant.conf
iface default inet dhcp

• $ sudo vi /etc/wpa_supplicant/wpa_supplicant.conf

• Add the following to /etc/wpa_supplicant/wpa_supplicant.conf
update_config=1
network={
ssid="YOURSSID"
psk="YOURPASSWORD"
Protocol type can be: RSN (for WP2) and WPA (for WPA1)
proto=WPA
Key management type can be: WPA-PSK or
#WPA-EAP (Pre-Shared or Enterprise)
key_mgmt=WPA-PSK}
```

Unmount and remove the SD card from the computer. Insert it back into the Raspberry PI and start the system again.

4. If connection to network was successful, access the Raspberry Pi via a terminal with ssh with a remote computer.
5. Enter user name: raspberry
6. Enter password: pi

7. Connect via USB cable the PlayStation®3 DUALSHOCK®3 wireless controller.
8. Make sure the lights on the controller are permanently blinking.
9. ArduCopter is a model simulation of a quadcopter for Xplane. Download from:  
<https://code.google.com/p/ardupilotdev/wiki/ArduCopter>  
and install on X-Plane.
10. Initiate Xplane and load the quad-copter.
11. Configure Xplane to send/recieve proper data with the Data window.
12. Start the Autopilot control panel executable on a remote computer.
13. If the program is downloaded with git, the file system can be saved anywhere in the user space file system. Each module has a folder named *scripts*. This folder contains a script called *build* that compiles each module and creates the libraries. To execute these files, type in the terminal the following command on each module:

```
$.build
```

This creates a binary file in *bin* folder in each module. Each binary file can be executed using the command:

```
$ sudo ./apc220_commd
```

Once all modules are compiled, the next step is to compile the autopilot. To do this, go back to the main program's folder and type

```
$ sudo make OS=XENOMAI
```

. Finally to execute the program, execute

```
$ sudo ./autopilotd
```

from the bin folder.

To use the existing Raspberry Pi project, execute the flight control under */workspace/autopilot/bin* and execute the program with

```
$ sudo ./autopilotd
```

14. System should be now ready to fly. Enjoy!

## 4.1 Pilots guide

A bit of skills are required to perform a good flight.

Be reminded that if the autopilot has been engaged prior to manual flight, the autopilot must send a stop message before the manual controller is engaged. In order words, prior to manual flight, press “stop” button on autopilot control panel. If this step is not completed, the quad-copter will have an odd behavior during flight. The reason for this is that the autopilot is sending the last selected command at the same time as the manual controller is sending its commands.

The following are a few steps to learn how to pilot the quad-copter.

- Enable controller.

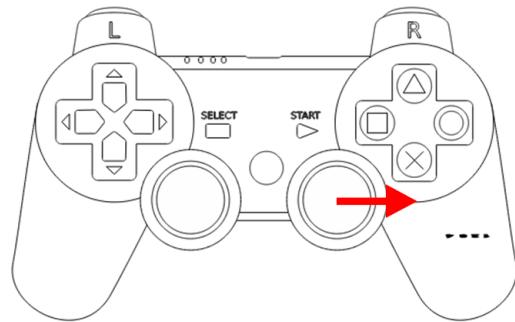


Figure 4.1: Manual controller enable.

- Disable controller.

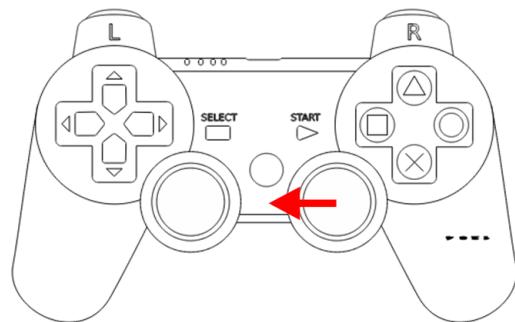


Figure 4.2: Manual controller disable.

- Climb maneuver.

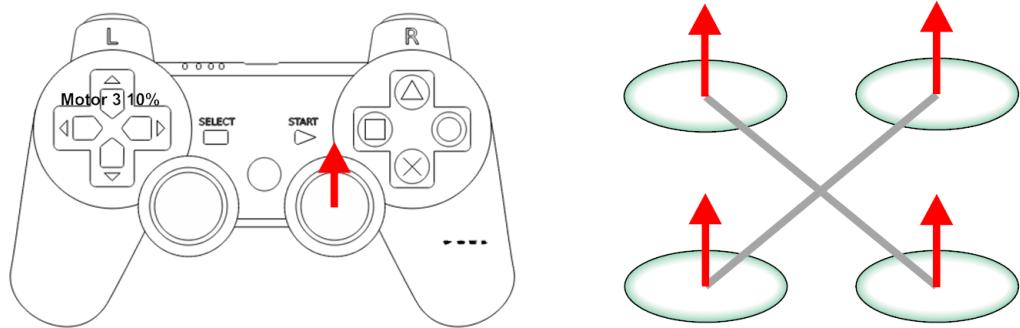


Figure 4.3: Manual controller climb maneuver.

- Descend maneuver.

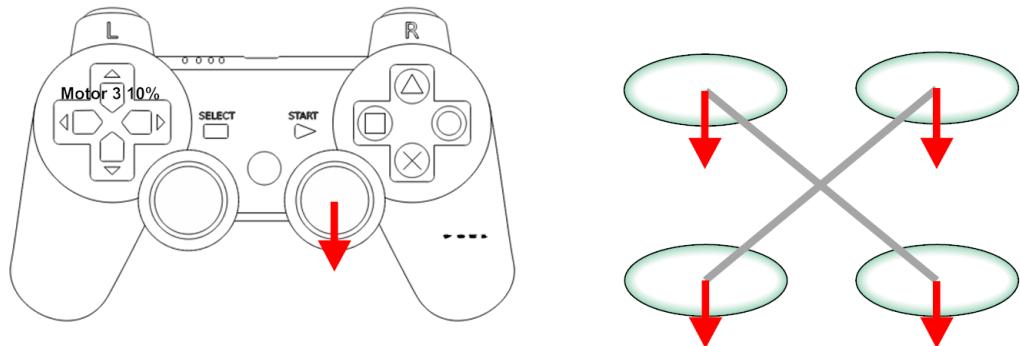


Figure 4.4: Manual controller descend maneuver.

- Forward maneuver (pitch down).



Figure 4.5: Manual controller forward maneuver (pitch down).

- Reverse maneuver (pitch up).

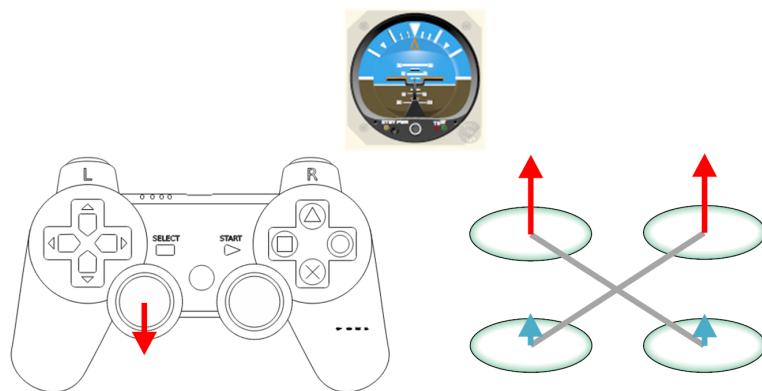


Figure 4.6: Manual controller reverse maneuver (pitch up).

- Yaw left maneuver.

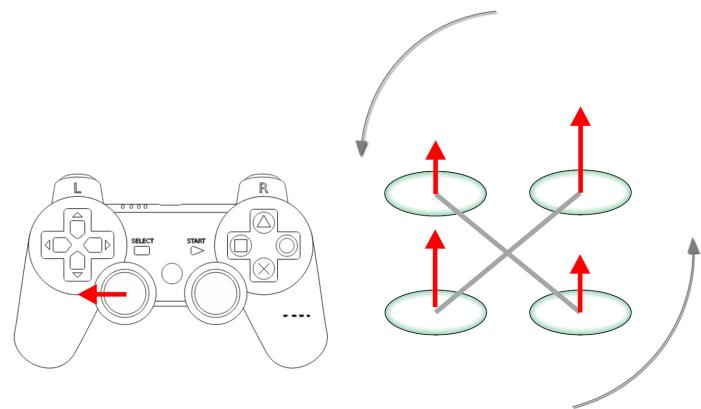


Figure 4.7: Manual controller yaw left maneuver.

- Yaw right maneuver.

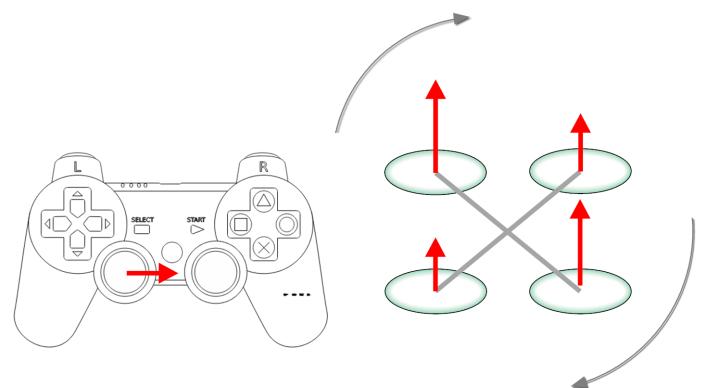


Figure 4.8: Manual controller yaw right maneuver.

- Takeoff.
  1. Enable controller.
  2. Apply power by gently sliding forward the right side joystick.
- Climb.
  1. Use same process as with takeoff.
- Hover.
  1. Release both joysticks to enter hover mode.
  2. During hover, slight attitude corrections maybe necessary.
- Forward.
  1. Gently apply power by sliding forward he left side joystick.
  2. This is a dangerous maneuver, keep pitch angle under control.
- Reverse.
  1. Gently apply power by sliding backwards he left side joystick.
  2. This is a dangerous maneuver, keep pitch angle under control.
- Yaw left.
  1. Gently apply power by sliding to the left the left side joystick.
  2. This is a dangerous maneuver, keep pitch and roll angles under control.
- Yaw right.
  1. Gently apply power by sliding to the left the left side joystick.
  2. This is a dangerous maneuver, keep pitch and roll angles under control.

- Descend.
  1. Gently release power by sliding back the right side joystick.
  2. This is a dangerous maneuver, keep pitch and roll angles under control.
  3. A fast descend may lead to vibrations on the quad-copter and loss of control.
- Land.
  1. Use same procedure as descend.
  2. When approaching ground, return to hover mode and then continue to slightly reduce power until touchdown.

---

## CHAPTER 5

### PROPOSED IMPROVEMENTS

THANKS to the design of the project, it is easy to improve the operability and functions of the system. Each improvement requires a new module and new communications with the system.

Following the systems design, each new component must also be a standalone program. In other words, it must be able to do all of its required functions without being integrated into the system.

The following is a list of new possible modules that can be added to the system.

- Live camera feed.

Installing a camera on the quad-copter gives it a huge amount of functionality. It enables the pilot to fly the drone without the need of visual contact with the UAV. The pilot only needs to look at a monitor or computer screen to see what the drone's camera is filming. An easy way to accomplish this is by using the TCP protocol on the autopilot control panel. In this panel, a new view window can be created where the live feed would show.

- Attitude indicator.

This module is probably the easiest new module to integrate. It can also be integrated in the autopilot control panel. To achieve this, the TCP communications between the embedded system and the autopilot control panel can be used. The embedded system can send the roll, pitch and altitude indicator values via TCP to the autopilot control panel. There are many open source visual attitude indicators already created that can be used for Qt.

- Assisted flight.

This is another very useful feature that does not involve the creation of a new module. This functionality can be added directly to the controller interface. Its mission would be to maintain control of high pitch angles and steep rolls induced by the pilot, consequence of poor piloting abilities.

- New hardware.

Although the Raspberry Pi model 1B+ has been found to best fit this project, there are many more boards that can fulfill all the required tasks providing more resources but at a higher price.

- On board.

The objective is to embed the flight control module onto the quad-rotor helicopter. This creates a more comfortable flight environment. On the ground, the pilot will only need a computer to connect and fly the drone.

## 5.1 Remarks

This project must be deployed together with a real-world quad-copter to enable the complete functionality of the system. Radio communications is the only basic quad-copter requirement needed to communicate with this project.

Both sides must first establish the data format and communication timing used to complete data exchange successfully. Additionally, they must agree on the units of measure for the values provided by the quad-copter sensors. Furthermore, there must be an agreement on the amount and type of sensors used on the quad-copter to cross-reference them with the values provided by the flight simulator X-Plane. By doing this, a correct flight maneuver is guaranteed on every flight mode scenario.



---

## REFERENCES

- [1] [Online] Xenomai RTOS. Xenomai — Real-time framework for Linux : Main web-page for Xenomai developers. (Versions, installation, API, references, know-hows, etc)  
URL: [www.xenomai.org](http://www.xenomai.org)  
(Accessed: Feb 2015).
- [2] [Online] NuclearProjects. X-Plane communication projects:  
TurbineJesse@gmail.com personal web-page with several shared projects, including Xplane communication examples.  
URL: [www.nuclearprojects.com](http://www.nuclearprojects.com)  
(Accessed: May 2015).
- [3] [Online] Joystick API Documentation:  
C API and library for joystick programming.  
URL: [www.kernel.org/doc/Documentation/input/joystickapi.txt](http://www.kernel.org/doc/Documentation/input/joystickapi.txt)  
(Accessed: May 2015).

- [4] [Online] Raspberry Pi Developers forum:  
URL: [www.raspberrypi.org/forums](http://www.raspberrypi.org/forums)  
(Accessed: Dec 2014).
- [5] [Online] Raspberry Pi Wiki:  
URL: [www.elinux.org/RPi\\_Hub](http://www.elinux.org/RPi_Hub)  
(Accessed: Jan 2015).
- [6] [Online] Raspberry Pi Kernel compilation tutorial:  
URL: [www.elinux.org/Raspberry\\_Pi\\_Kernel\\_Compilation](http://www.elinux.org/Raspberry_Pi_Kernel_Compilation)  
(Accessed: Dec 2014).
- [7] C Programming Language, Prentice Hall Software  
Brian W. Kernighan, Dennis Ritchie, Apr 1998.
- [8] Real-Time Systems and Programming Languages: Ada 95, Real-Time Java and Real-Time POSIX, International Computer Science Series  
Alan Burns, Andy Wellings, Aug 2001.

