Politecnico di Torino

Master of Science degree in Mechatronic Engineering

Master Thesis

PX4 autopilot customization for non-standard gimbal and UWB peripherals

Supervisor:
Marcello Chiaberge

Candidate:
Francesco Malacarne
s260199

Thesis advisors:
Dott. Ing. Gianluca Dara
Dott. Ing. Simone Silvestro

Academic year 2019-2020

Contents

List of Figures

Abstract

Due to their extreme versatility, autonomous drones have been gaining a lot of interest over the last decades. Applications are mainly intended to reduce injuries, saving lives and optimizing already existing processes, covering a wide range of scenarios, such as smart agriculture, emergency situations and safety inspections.

PIC4SeR, PoliTO Interdepartmental Centre for Service Robotics, embraced this topic a few years ago starting to develop different UAVs, from ultralight models lighter than 250 grams to much heavier drones, in order to exploit technologies oriented to service robotics, smart cities, precision agriculture and search-rescue operations, which are the main topics of interest of the center. This thesis was born as one of the building blocks of a wider project about swarm flight, indoor navigation and robots cooperation, that will take place in the near future merging several technologies coming from different works carried by researchers, PhD students and master thesis works.

As most of the standard solutions implemented in autonomous drone navigation rely on GNSS positioning, indoor navigation represents one of the main challenges to be managed. Although computer vision is getting more and more accurate, allowing a possible solution to this problem, it always needs a lot of computational power, requiring the addition of companion PCs, increasing both the system complexity and the UAV's weight. Nonetheless, an emerging technology used to solve indoor navigation problems leaning on very low hardware resources components, is the Ultra Wide Band technology, an anchor-tag radio-frequency based positioning system. Considering that no drivers for the integration of UWB modules in UAV autopilots have been developed in literature so far, the goal of this thesis is the deep understanding of the RTOS PX4 autopilot firmware, and the implementation of custom modules, drivers and commands, with particular focus on UWB nodes.

Introduction

1.1 Objective of the thesis

The goal of this thesis is to study and customize the open-source autopilot PX4, integrating functionalities and non-standard hardware components. The main focus will be on two big topics: the integration of a non-standard lightweight camera gimbal on a drone mounting Pixracer flight controller, and the integration of a UWB module to be used for future works about indoor positioning and swarm navigation. The stabilization of the camera pitch is the progression of a previous work developed at PIC4SeR by Simone Silvestro, which was aimed at building an ultralight drone (less than 250 grams) to perform most of the standard operations a heavier drone could do, but with fewer law constraints, due to its weight. The problem with such a model was that the chosen flight controller (Pixracer), was not built for mounting a camera pitch stabilizer, despite having a pin potentially available for this purpose. Therefore, since most of the operations require a camera recording specific scenes, we decided not only to add the possibility of changing the camera orientation using the radio controller but also to introduce a stabilization module to get smoother videos.
On the other side, dealing with the second topic, the standard approach for introducing UWB sensors is to do so using companion PCs, which are general-purpose PCs connected to the drone, much more powerful than the flight controller used to run the autopilot, allowing the UAV to perform more sophisticated tasks (vision-based tasks, object tracking and many others). With this approach, the flight controller is not directly connected to the UWB module, but it is leveraging data processed by an external unit (companion PC) interfaced with the drone with a specific communication protocol. Although being a common and versatile solution, companion PCs cannot be physically mounted on drones that must be lighter than 250 grams, due to their weight. Therefore, the goal of this work is to bypass the companion PC and read the necessary UWB measures directly from the autopilot leveraging a custom driver.

1.2 Organization of the thesis

The work of this thesis is organized in seven chapters, here briefly summarized for a better understanding.

Chapter 1: a short introduction about the problem to be solved and the organization of the thesis.

Chapter 2: an overview about the state of the art of drone autopilots focusing on standard procedures used to introduce UWB sensors in UAVs.

Chapter 3: PX4 overview highlighting the main parts that will be involved in this work.

Chapter 4: an overview about the main PX4 competitor (Ardupilot) plus a comparison between the two firmware.

Chapter 5: description of the two customization applied to the firmware:

- Introduction of a module able to stabilize the camera pitch on Pixracer.
- Development of a serial driver to integrate UWB modules.

Chapter 6: detailed description of the testing phase highlighting obtained results and problems to be solved.

Chapter 7: an overview about the obtained results with some hints about further development under study at PIC4SeR.

State of Art

State of Art was the first challenge of this thesis. In the first place it was necessary to understand how autonomous drones are made, since I have never had the chance of working with such a system, then it was useful to validate the purpose of my work. In particular, I had to figure out whether it was possible to introduce a single-axis camera gimbal and a UWB sensor without a companion PC.

In the first part of this chapter I will briefly introduce the main components of an autonomous drone, focusing more on the software parts rather than on the hardware ones, as they have been deeply introduced by Simone Silvestro's work. Then, in the second part, I will show the most relevant papers I found during my investigation, highlighting the main challenges I will have to face at during the work.

2.1 System description

Any autonomous vehicle project is essentially composed by the following parts:

- **Hardware**: combination of sensors, controllers and output devices allowing the drone to sense the external environment and decide what to do according to the current scenario. It essentially consists in all the physical components composing the drone.
- **Firmware**: the code running on the controller which guarantees reliability, stability and provides features leveraging inputs from sensors, elaborating them through a processing unit and sending outputs to other peripherals such as ESC's, servos, gimbals etc. The firmware allows to integrate hardware components.
- **Software**: it is the interface to the controller often called Ground Control Station (GCS). Software can run both on PC's and/or mobile devices. A GCS allows users to set-up, configure, test, and finely tune the vehicle setup, acting on the firmware in a simple and straight-forward fashion. Advanced packages or add-ons allow autonomous mission planning, operation, and post-mission analysis [3]. This is generally used by the operator both to program the mission and to analyze log files.

The cooperation of these 3 parts allows the drone to perform specific operations autonomously, given an initial setup provided by the operator. Working on the firmware, different hardware components can be integrated, allowing the drone either to perform new tasks or to enhance already existing ones.

Autonomous drones are gaining a lot of interest both in private companies and in universities. This lead to a subdivision of the market in two different categories of systems: open-source and private. The main interest of this thesis will be on open-source systems, with focus on open-source firmware.

2.2 Open-source firmware

Autonomous drones are gaining a lot of interest both in private companies and in universities. This lead to a subdivision of the market in two different categories of systems: open-source and private. The main interest of this thesis will be on open-source systems, with focus on open-source firmware.
Open-source firmware can be divided in 2 different categories based on the automation level they can provide: fully autonomous firmware are those not requiring human intervention at all (or just partially, according to the flight mode that is chosen), whereas non-autonomous firmware are those requiring human control for any operation (no level of automation). This work will be focused on the first category, as the final goal is to achieve a fully autonomous drone able to work in specific conditions. There are many different open-source firmware available on the market: ArduPilot, PX4, Paparazzi, FlexiPilot, Airrails, SmartAP, Armazila, Autoquad and SLUGS, but only the first two can rely on a stable community and complete documentation. Both ArduPilot and PX4 are valid choices as far as the available functionalities are concerned; wide range of hardware compatibility, good software stability, similar architecture, ROS compatibility and strong community are only a flavour of their entire capabilities, nonetheless, some differences are present.
Before choosing one of the two software it is important remarking that both alternatives will lead to similar and optimal performance. The choice made for the PIC4SeR project was PX4, essentially due to the BSD license and the user-friendliness.

2.3 Recent works from other Universities

Investigations were mainly focused on UWB integration rather than camera stabilization, as the latter one is a more practical implementation deeply discussed on the community forum rather than in academical researches. In fact, there is an already existing (and well documented) 3-axis gimbal implementation available for Pixhawk boards, therefore the extension to Pixracer was only a matter of understanding the original module and writing a new one, based on the already existing one.
On the other side, as far as the UWB technology is concerned, the goal was to understand the standard approach used to introduce such modules in autonomous systems, in order to compare it with our aim, trying to understand whether it was even possible to realize it.

One of the best paper able to summarize the standard approach is the one written by Francesco Betti Sorbelli and Cristina M. Pinotti, researchers from University of Perugia, Italy, named “Ground Localization with a Drone and UWB Antennas: Experiments on the Field” [38], that is available on IEEE periodic. This work was aimed at evaluating the precision of the Drone Range-Free (DRF) localization algorithm, comparing it with

a custom algorithm, leveraging a drone mounting the same UWB sensors used in our research center. My interest was not about their results concerning the localization algorithm, but on the used setup. They used the 3DR Solo drone running PX4 connecting the UWB module to a Raspberry Pi interfaced to the drone, therefore the flight controller was not receiving data directly from the UWB module, it was receiving messages from the companion PC. A similar approach was used by Janis Tiemann, Andrew Ramsey and Christian Wietfeld, researchers from Dortmund University, in their work named "Enhanced UAV Indoor Navigation through SLAM-Augmented UWB Localization" [43], where they implemented an indoor navigation system for drones based on SLAM and UWB technology, using a Parrot Bebop 2 connected to a companion PC. It is plenty of work relying on such a setup, because it is strongly versatile allowing to both create complex projects (vision-based, neural network oriented etc.) and to abstract the UWB modules from the flight controller (guaranteeing interchangeability between different drones). Nonetheless, it is not compatible with ultralight drones due to the companion PC's weight, which is generally higher than 250g itself, making the entire system exceeding the law limitations that play a relevant role in drone operations. This is why Michael Strohmeier, Thomas Walter, Julian Rothe, and Sergio Montenegro researches from University of Wuerzburg, Germany, in their work "Ultra-Wideband Based Pose Estimation for Small Unmanned Aerial Vehicles" [40], available on IEEE, did not use a companion PC, but they directly connected the UWB module to the custom drone, realizing a final assembly lighter than 200 grams. Even though they implemented the UWB module with the operating system Real-time Onboard Dependable Operating System (RODOS), their setup increased our chances of realizing an ultralight drone running PX4 implementing UWB technology.

PX4 overview

3.1 PX4 main features

"PX4 is an open-source flight control software for drones and other unmanned vehicles. The project provides a flexible set of tools for drone developers to share technologies to create tailored solutions for drone applications" [28].
Its main properties are:

- **Modular Architecture**: PX4 is strongly modular and extensible, both in terms of hardware, allowing the integration of many different sensors, and software, allowing to directly interact with the firmware to implement any desired feature.
- **Open-source**: PX4 development is carried out all around the world by a very active community. The flight stack is not intended to fulfill the needs of single labs or companies, but has been designed as a general toolkit to be used in both industries and universities.
- **Configurability**: PX4 provides optimised APIs and SDKs for developers working with integrations and customizations.
- **Autonomy Stack**: PX4 is designed to be deeply coupled with embedded computer, in particular concerning vision for autonomous capabilities.

3.2 PX4 architectural overview

PX4 is made of two main layers: the **flight stack** that is an estimation and flight control layer, and the **middleware** that is a general robotics layer providing hardware integration and internal/external communications. PX4 main feature is that all the airframes share a single codebase (including other robotic systems like rovers, submarines, boats etc.) [32].

3.2.1 Flight stack

According to the developer guide, "the flight stack is a collection of guidance, navigation and control algorithms for autonomous drones. It includes controllers for fixed wing, multirotor and VTOL airframes as well as estimators for attitude and position" [32]. In this thesis the main focus will be on multirotor, nevertheless, everything can be easily extended to different airframes.

Figure 3.1 shows a general overview of the flight stack building blocks. It contains the full pipeline from sensors, RC input and autonomous flight control (Navigator), down to the motor or servo control (Actuators) [32].

- An **estimator** takes one or more sensor inputs, combines them according to specific algorithms, and computes a vehicle state (for instance the attitude based on IMU sensor data).
- A **controller** is a component that takes a setpoint (i.e. the target) and a measurement (if available) or estimated state (process variable) as input. Its aim is to adjust the value of the process variable so that it matches the target setpoint. The output is the correction needed to reach the setpoint. Making an example, the position controller has position setpoints acting as inputs, the current estimated position as process variable, whereas the output is a combination of attitude and thrust setpoints moving the vehicle towards the desired position.
- "A **mixer** takes force commands (e.g. turn right) and translates them into individual motor commands, while ensuring that some limits are not exceeded. This translation is specific for a vehicle type and depends on various factors, such as the motor arrangements with respect to the center of gravity, or the vehicle's rotational inertia" [32].

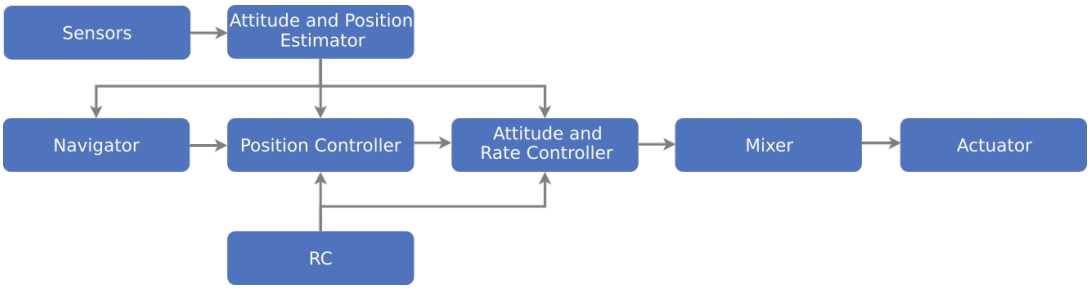

Figure 3.1: Flight stack diagram. Image source: PX4 developer guide.

Multicopter position controller

The multicopter position controller diagram is shown below. The controller is made of two loops: a P loop for position error and a PID loop for velocity error. The controller output is a thrust vector split in two components: thrust direction (namely rotation matrix for multicopter orientation) and thrust scalar (i.e. multicopter thrust itself). The controller does not need Euler angles for its routine, they are generated to be used (control purposes) and understood by developers (logging).
Some remark about such a control schematic based on the developer guide [33]:

- Estimated states and variable come from EKF2.
- The control schematic represents a standard cascaded position-velocity loop.
- The position loop is only involved in specific situations: when holding position or when the desired velocity along an axis is null. This is why the outer loop may be bypassed depending on the used mode (whenever the position should not be held).

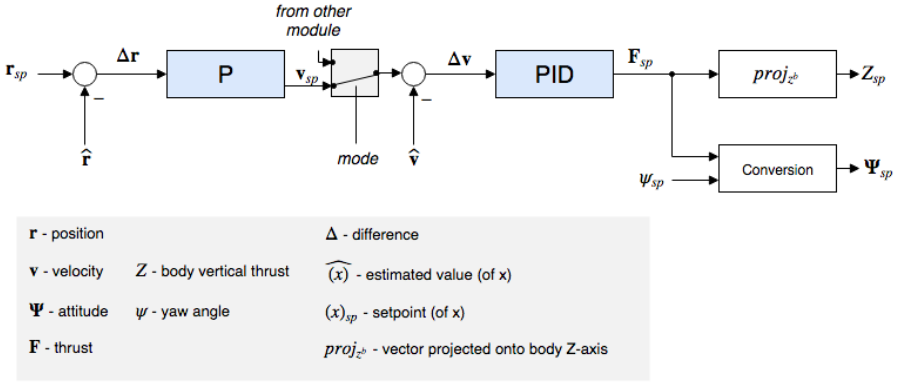

Figure 3.2: `mc_position_controller` control diagram. Image source: PX4 developer guide

- The integrator within the inner loop controller (velocity) includes an anti-reset windup, using a clamping method.

Multicopter attitude controller

The multicopter attitude controller [7] diagram is very similar to the position one, although implementing the multicopter attitude and rate controller. It takes attitude setpoints (`vehicle_attitude_setpoint`) or rate setpoints (in acro mode via `manual_control_setpoint` topic) as inputs and outputs actuator control messages [33]. The controller has two loops: a P loop for angular error and a PID loop for angular rate error.
Publication documenting the implemented Quaternion Attitude Control can be found at the following link:
`https://www.research-collection.ethz.ch/bitstream/handle/20.500.11850/154099/eth-7387-01.pdf`

3.2.2 Middleware

"The middleware consists primarily of device drivers for embedded sensors, communication with the external world (companion computer, GCS, etc.) and the message bus. The Firmware is made of modules/programs communicating each other through a publish-subscribe message bus named **uORB**" [32]. While the flight stack is necessary to let the drone fly in a reliable manner, the middleware is that part of the firmware allowing to add functionalities, integrating sensors or changing the behaviour of the already existing ones.
Dealing with a publish-subscribe scheme means that:

- **The system is reactive** therefore it is asynchronous and will update instantly whenever new data are available.
- All operations and **communications are fully parallelized**.
- A system component can employ data from anywhere in a **thread-safe manner**.

Moreover, the middleware incorporates a simulation layer allowing PX4 to run on a desktop OS and control a modelled vehicle in a simulated environment.

Since the main goal of this work is to integrate sensors and actuators, the involved part of the firmware is the middleware. On the contrary, future applications such as indoor navigation, will likely require some changes into the flight stack.

3.2.3 Runtime environment

PX4 runs on many different operating systems providing a POSIX-API (NuttX, macOS, Linux or QuRT). For our application NuttX will be the used RTOS.
The inter-module communication relies on uORB and it is based on shared memory. A single address space is used to run the entire PX4 middleware, in fact memory is shared between all modules. Nevertheless, the system is designed such that with minimal changes it would be possible to execute each module in a private address space [32].
There are 2 different ways to execute a module:

- **Task**: the module runs on its own task with its own stack and process priority (this is the most used case). The main advantage in using such approach is that it requires less RAM, even though the task is not allowed to sleep or poll on messages.
- **Work queue**: the module runs on a shared task, meaning that it does not own a proprietary stack. Multiple tasks run on the same stack with single priority per work queue. Work queues are essentially used for periodic tasks, such as sensor drivers or the land detector [32].

3.3 NuttX

NuttX is the primary RTOS for running PX4 on a flight-control board. It is open-source (BSD license), light-weight, efficient and very stable [2]. NuttX, as all RTOSs, is a collection of various features packed as a library. It is executed in 2 conditions only: when the application is asking a piece of the NuttX library code, or when an interrupt occurs [26].
Its most relevant features are:

- Fully preemptible.
- FIFO, round-robin, and “sporadic” scheduling.
- Realtime, deterministic, with support for priority inheritance.
- System logging.
- 30+ supported platforms.
- 22 supported file systems.

Before delving into NuttX, it is necessary to clarify the difference between processes/tasks and threads. “A thread of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the

operating system" [42]. The differentiation and usage of threads and processes changes among operating systems, but in most cases a thread is a process component. From the NuttX guide it is possible to see how a process is defined: it is a *protected environment* that hosts one or more threads. By environment it is generally meant the set of resources set aside by the OS (essentially memory) [26].

"All RTOSs support the notion of 'tasks' rather than processes. A task is the RTOS's moral equivalent of a process" [26]. Why tasks are so important? Because modules (the firmware parts we can actually modify and create) are executed as tasks.

Each task is represented by a particular data structure called task control block or TCB, which is unique. Nevertheless, all tasks share a single address space [26] [27].
Moreover, each task/thread has a fixed-size stack, and there is a periodic task which checks whether all stacks have enough free space left (based on stack coloring) [32]. The stack is that portion of the memory used to manage the automatic allocation of memory; stack may contain the CPU registers during context switch or other useful data.

3.3.1 Task states

Why do we need to define task states? Because only one task at a time can be executed, all the other must wait the CPU to be available. The mechanism that allows to stop a task execution and start/restore another one is called context switch. Context switch implements 6 basic operations: it sets the current running task as ready (all resources available but the CPU), it saves the context (progress made by the task i.e. the CPU registers), and it appends this task to the queue list. Then it moves the task on top of the queue to the running one, and it restore its content, so that the task can start again from the very same place it was interrupted.

A task state can be indicated both by the `task_state` field of the TCB and by a series of task lists. Although not being always necessary, most of these lists are prioritized.[26] [27].
Whenever a new task arrives, it starts in a non-running, uninitialized state:

```
volatile dq_queue_t g_inactivetasks;
```

This is the list of all tasks that have been initialized, but not activated yet. An important remark is that this is the only list that is not prioritized.

Whenever the task is initialized, it is moved from the non-running list to a ready-to-run list, that is the list containing all the tasks only needing to be given to the CPU to run. There are two lists representing ready-to-run threads and several lists representing blocked threads. The ready-to-run threads are the following ones:

```
volatile dq_queue_t g_readytorun;
```

This is the list of all tasks being actually ready to run. The head of this list is the currently running task, whereas the tail of this list is always the idle task because whenever all the other tasks have been completed, the CPU is given to the idle task (lowest priority) to do nothing until higher priority task arrive.

```
volatile dq_queue_t g_pendingtasks;
```

This is the list of all tasks theoretically being ready-to-run, but cannot be placed in the

g_readytorun list because a particular situation takes place: they have a higher priority than the currently running task, hence they should be given to the CPU, but the running task, who arrived first, disabled pre-emption. Basically, the lower priority task arrived before the higher priority task, but it disabled the pre-emption mechanism, so that when the higher priority task arrived it could not be executed by the CPU. This is a condition generally used to protect critical sections and it generates the priority inversion problem (lower priority task blocks a higher priority one) [27].
These tasks will remain in this holding list until pre-emption is again enabled (or the currently running task voluntarily relieves the CPU for any reason).

Tasks in the g_readytorun list may become blocked while needing data coming from other tasks or peripherals. In this case, they will be moved to one of the blocked lists. When the blocked task is again ready-to-run, it will be moved back to either the g_readytorun or to the g_pendingtasks lists, depending on the situation [27].
Here the blocked task lists:

```
volatile dq_queue_t g_waitingforsemaphore;
```

This is the list of all tasks that are blocked waiting for a semaphore.

```
volatile dq_queue_t g_waitingforsignal;
```

This is the list of all tasks that are blocked waiting for a signal (signal support must be enabled to have this list).

```
volatile dq_queue_t g_waitingformqnotempty;
```

This is the list of all tasks that are blocked waiting for a message queue to become non-empty (message queue support must be enabled to have this list).

```
volatile dq_queue_t g_waitingformqnotfull;
```

This is the list of all tasks that are blocked waiting for a message queue to become non-full (message queue support must be enabled to have this list).

```
volatile dq_queue_t g_waitingforfill;
```

This is the list of all tasks that are blocking waiting for a page fill (only if on-demand paging is selected) [26] [27].

3.3.2 Scheduling policy

In order to be a real-time OS, the OS must support SCHED_FIFO that is, strict priority scheduling: the highest priority thread runs (i.e. given to the CPU). The highest priority thread is always associated with the TCB on the head of the g_readytorun list.
As it is possible to read from the official NuttX website, "NuttX supports one additional real-time scheduling policy: SCHED_RR. The RR stands for roundrobin and this is sometimes called Round-Robin scheduling. In this case, NuttX supports timeslicing: If a task with SCHED_RR scheduling policy is running, then when each timeslice elapses, it will give up the CPU to the next task that is at the same priority. Note: (1) If there is only one task at this priority, SCHED_RR and SCHED_FIFO are the same, and (2) SCHED_FIFO tasks are never pre-empted in this way" [26] [25].

Task ID

The TCB is not the only structure representing a task, in fact there is a special number univocally identifying each task named task ID which must be considered dealing with RTOSs. TCB and task ID are functionally equivalent: given a task ID, the RTOS can find the associated TCB, given a TCB, the RTOS can find the related task ID. Nonetheless, only the task ID is exposed at the application interfaces [26] [25].

3.3.3 Virtual File System

NuttX implements a Virtual Files System (VFS) that can be used to communicate with a number of different entities through the standard open(), close(), read(), write(), etc. interfaces. Like other VFSs, the NuttX VFS will support file system mount points, files, directories, device drivers, etc. [26] [25].

3.3.4 NuttX initialization

NuttX initialization sequence can be broken down in three simple phases:

1. The hardware-specific power-on reset initialization.
2. NuttX RTOS initialization.
3. Application initialization.

This initialization sequence is actually quite simple because the system runs in single-thread mode until the application starts. Essentially, the initialization sequence is just straight-line function calls. Right before starting the application, the system changes into multi-threaded mode and things complicate a lot [26] [25].

Power-on reset initialization

Software begins execution whenever the processor is reset. Reset condition takes place in three situations generally: power-on (standard booting), pressing reset button, or due to watchdog timer expiration. Anyhow, the software executing when the processor is reset is specific to the used CPU architecture and is not a standard part of NuttX [26] [25]. However, the high-level operations that the architecture-specific reset should perform are [26]:

1. Putting the processor in its operational state. This may include operations like setting CPU modes; initializing co-processors, etc.
2. Setting up clocking so that the software and peripherals operate as expected.
3. Setting up the C stack pointer (and other processor registers).
4. Initializing memory.
5. Starting NuttX.

NuttX RTOS initialization

When the low-level, architecture-specific initialization is completed, NuttX is started by calling the function `os_start()`. This function is located in the file nuttx/sched/os_start.c. The operations performed by such a function are summarized below. Many of these features can be disabled from the NuttX configuration file; in those cases the operations are not performed [26] [25]:

1. Initialize some NuttX global data structures.
2. Initialize the TCB for the IDLE (i.e, the thread that the initialization is performed on).
3. kmm_initialize(); initialize the memory manager (in most configurations kmm_initialize() is an alias for the common mm_initialize()).
4. irq_initialize(); initialize the interrupt handler subsystem. This initializes only data structures; CPU interrupts are still disabled.
5. wd_initialize(); initialize the NuttX watchdog timer facility.
6. clock_initialize(); initialize the system clock.
7. timer_initialize(); initialize the POSIX timer facilities.
8. sig_initialize(); initialize the POSIX signal facilities.
9. sem_initialize(); initialize the POSIX semaphore facilities.
10. mq_initialize(); initialize the POSIX message queue facilities.
11. pthread_initialize(); initialize the POSIX pthread facilities.
12. fs_initialize(); initialize file system facilities.
13. net_initialize(); initialize networking facilities.

    So far, all of the performed steps have been software initializations only; no hardware was involved. Actually, all of these steps simply prepared the environment so that features like interrupts and threads can properly work.
14. up_initialize(); the processor-specific details for running the operating system will be managed here. Such operations as setting up interrupt service routines depend on the used processor and hardware platform.
15. lib_initialize(); initialize the C libraries. This is done at the end because libraries may depend on the previous steps.
16. sched_setupidlefiles(); this is the logic that opens /dev/console and creates stdin, stdout, and stderr for the IDLE thread. All tasks subsequently created by the IDLE thread will inherit these file descriptors.
17. os_bringup(); create the initial tasks.

18. Finally enter the IDLE loop. After completing the initialization, the IDLE thread becomes the thread executing only when there is nothing else to do in the system (from where, the name IDLE thread).

As previously mentioned, the IDLE thread is that specific thread that executes only when nothing else can be done. It has the lowest priority in the system (priority 0) and it is the only thread being able to have such a priority. IDLE thread can never be blocked (otherwise, what would run then?). As a consequence, the IDLE thread is always in the `g_readytorun` list and, in particular, being this list a prioritized one, and having the IDLE thread lowest priority, it will always be the tail of the `g_readytorun` list.
The IDLE thread appears as an infinite loop (doing nothing). Nonetheless, it never saturates CPU because having the lowest priority, whenever new task arrive (any task), it will always pre-empt the IDLE thread [26] [25].

3.4 uORB

uORB is the core of all the communications between the drone internal components (modules). It consists in an asynchronous publish() / subscribe() messaging API used for inter-thread/inter-process communication automatically started early on bootup (`uorb start` command) as many applications rely on it [46].
The list of the built-in topics (more than 100) can be found at `https://github.com/PX4/Firmware/tree/master/msg/`, or within the directory `Firmware/msg` on the development machine where PX4 firmware is downloaded. An example of message file (`mount_orientation.msg`) is shown in figure 3.3; it contains two elements: the timestamp in microseconds and the camera attitude expressed with the Euler angle convention.

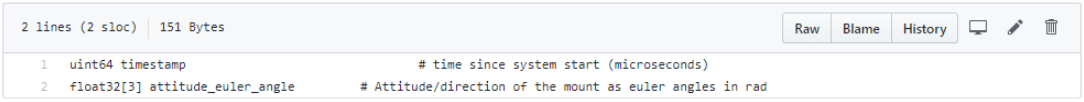

Figure 3.3: Message file example (`mount_orientation.msg`). Image source: GitHub repository.

Although having a lot of built-in topics, it may be needed to add new ones. To add a new topic, it is required to create a new .msg file in the `msg/` directory and add the file name to the `msg/CMakeLists.txt` list. From this, the needed C/C++ code is automatically generated while making the code. Having a look at the existing msg files may be useful for checking supported types. A message can also be used nested in other messages [46]. "To each generated C/C++ struct, a field uint64_t timestamp will be added. This is used for the logger, so make sure to fill it in when publishing the message" [46]. To use the topic in the code, it is enough to include the header:

```
1 #include <uORB/uORB.h>
2 #include <uORB/topics/topic_name.h>
```

Figure 3.15 represents the connection between modules (threads/processes) and topics. Modules are represented in grey rounded boxes, topics in coloured squared containers. Modules can either subscribe or publish to topics: dashed lines represent a module publishing on a topic, solid lines subscribing to it.

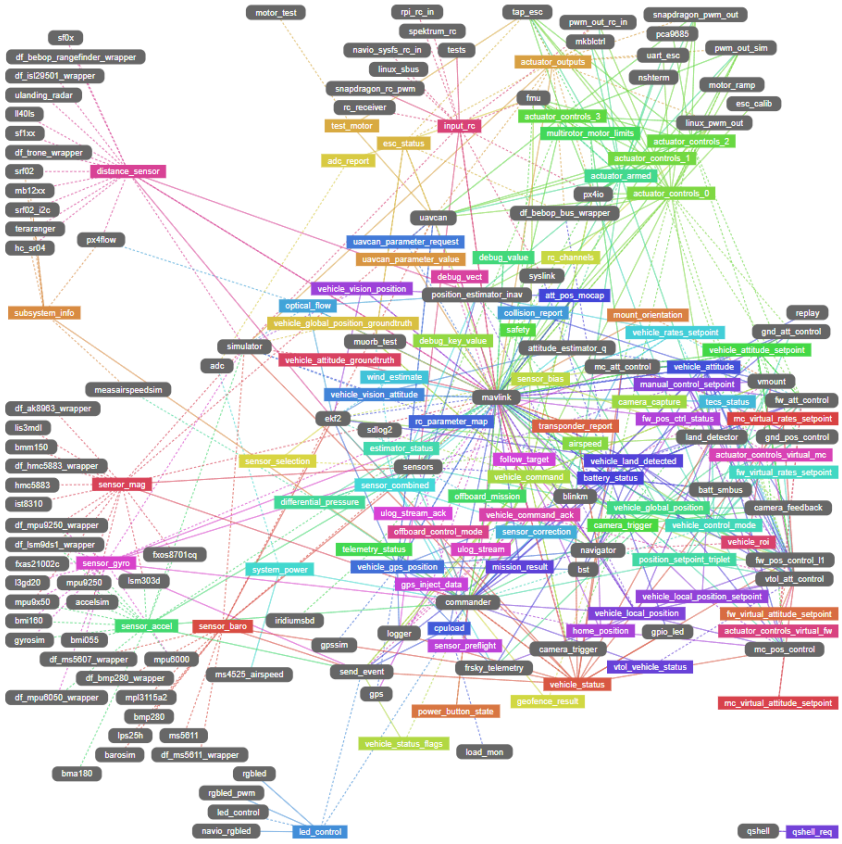

Figure 3.4: uORB schematic. Image source: PX4 developer guide.

How to add a module communicating with others?

1. The first operation is to create the module. For this simple tutorial I'm going to use the built-in module template available on directory `Firmware/src/templates/module` or `https://dev.px4.io/master/en/apps/module_template.html`. Any module is composed by 3 fundamental components: the .cpp file, implementing the actual code, the .h file containing all the class declarations and libraries, and lastly the CMakeLists.txt file needed to build the code.

2. The second operation is to place the module folder, composed by the 3 above mentioned file, inside the correct directory. The directory depends on the kind of module that have been created (driver, module, example, system commands etc.), nevertheless, the starting point is always the same: `Firmware/src`. For instance, if a driver have been created, it should be placed in `Firmware/src/drivers`.

3. The third operation is to let the system know the new module is available. In order to do it, go into `Firmware/boards/px4` and add to the target platform default.cmake file the name of the module within the associated list (in our example under the drivers list).

4. Finally, test the module (`make px4_sitl gazebo` if you are working in SITL). Typing `help` inside the simulation shell you should be able to find the name of the new module. For testing it, just use the command `module start`.

actuator_outputs

The `actuator_outputs` topic contains the PWM values that are passed to the different ports on the autopilot board (e.g. Pixracer). It is generally a value between 1000 and 2000, whereas 900 is the default value for disarmed condition. The number of PWM values depend on the specific autopilot board, for instance, using Pixracer only 6 signals are passed (4 motors plus 2 other pins). Unfortunately, it is not possible to directly write the PWM value on a specific pin without interfering with the others; the only solution is to pass through `actuator_controls` topic.

Which modules publish on `actuator_outputs` topic? Essentially `fmu, uavcan, px4io`. The most important one for our purposes is `fmu`, because it is used to convert values coming from `actuator_controls` into suitable PWM outputs.

Which modules subscribe to `actuator_outputs` topic? `Mavlink, simulator`.

actuator_controls

The `actuator_controls` topic contains the control values of all the possible control groups. It is more than just a topic, because it contains all the "sub-topics" regarding the different control groups (a detailed overview about the control groups can be found in section 3.8.3). For instance, `actuator_controls_0` contains the control values of the control group 0 (roll, pitch, yaw and thrust) [19].

- `actuator_controls_0`: roll, pitch, yaw and thrust (multicopter).
- `actuator_controls_1`: roll, pitch, yaw and thrust (VTOL).
- `actuator_controls_2`: roll, pitch, yaw and shutter (gimbal). This topic can be used only in the presence of the AUX ports, i.e. no with Pixracer autopilot.
- `actuator_controls_3`: manual passthrough. Useful values are 6th and 7th (number 5 and 6 respectively), since they allow to exploit the original mixer files and tune PWM output on pin 5 and 6.
- `actuator_controls_virtual_fw`
- `actuator_controls_virtual_ms`

Which modules publish on `actuator_controls` topic? Essentially `mc_att_controller`.

Which modules subscribe to `actuator_controls` topic? `fmu, uavcan, px4io, commander, mavlink, sensors`. `fmu` is the module connecting control inputs with PWM outputs.

input_rc

The `input_rc` topic contains information about the radio commands passed by the user, plus additional flags such as lost signal or failsafe mode. This topic is a sort of informative topic. The module actually responsible for this update is `rc_update`.

Which modules publish on `input_rc` topic? Essentially `fmu, px4io, simulator, rc_receiver`. The most important one is the receiver driver.

Which modules subscribe to `input_rc` topic? `Mavlink, sensors`.

sensor_combined

The `sensor_combined` topic is a sort of hub for sensor values. It contains two main elements:

- `gyro_rad`: average angular rate measured in the XYZ body frame in rad/s over the last gyro sampling period (the sampling period is defined within this topic, variable `gyro_integral_dt` measured in us) [35].

- `accelerometer_m_s2`: average value acceleration measured in the XYZ body frame in m/s/s over the last accelerometer sampling period (the sampling period is defined within this topic, variable `accelerometer_integral_dt` measured in us) [35].

The `sensor_combined` topic receives data from the module `sensors` and it sends them to `navigator, commander, frsky_telemetry and mavlink` [35].

sensors module

The sensors module assumes a key role to the entire system. It takes low-level output from drivers, turns them into a more valuable form (filtering), and publishes them, letting the other modules to take benefit of the clean measurements.
The provided functionality includes:

- Read the output from the sensor drivers (`sensor_gyro, sensor_accel, sensor_baro, sensor_mag, differential_pressure`). If there are multiple of the same type, do voting and failover handling. Then apply the board rotation and temperature calibration (if enabled). And finally publish the data; one of the topics is `sensor_combined`, used by many parts of the system [36].

- Make sure the sensor drivers get the updated calibration parameters (scale and offset) when the parameters change or on startup. The sensor drivers use the ioctl interface for parameter updates. For this to work properly, the sensor drivers must already be running when '`sensors`' is started [36].

- Do preflight sensor consistency checks and publish the `sensor_preflight` topic [36].

vehicle_attitude

The `vehicle_attitude` topic contains two important parameters:

- `q`: quaternion rotation from XYZ body frame to NED earth frame.

- `delta_q_reset`: quaternion variation from the last reset.

Which modules publish on `vehicle_attitude` topic? Only `ekf2`, because it provides filtered data based on the measured ones.

Which modules subscribe to `vehicle_attitude` topic? `Mavlink` and most of the controllers.

vehicle_gps_position

The `vehicle_gps_position` topic contains all the information coming from gps sensor, such as velocity with respect to ground, accuracy etc.

Which modules publish on `vehicle_gps_position` topic? Only `gps`, because it implements driver functionality.

Which modules subscribe to `vehicle_gps_position` topic? `Mavlink, EKF2, commander, navigator`.

vehicle_control_mode

The `vehicle_control_mode` topic contains a list of boolean values used to define the kind of control mode that is implemented (complete list available at `https://docs.px4.io/master/en/flight_modes/`).

Which modules publish on `vehicle_control_mode` topic? Only `gps`, because it implements driver functionality.

Which modules subscribe to `vehicle_control_mode` topic? `Mavlink, ekf2, commander, navigator`.

vehicle_status

The `vehicle_status` topic contains information about the drone status, in particular about the arming condition, vehicle type, navigation state etc. As it is a sort of information hub, a lot of modules are subscribed to it. The most relevant ones are: `mavlink, commander, navigator`.

vehicle_command

The `vehicle_command` topic contains a huge list of commands (each identified with a specific number) to be given to the drone in order to perform specific operations. A complete list of all the available commands can be found at: `https://mavlink.io/en/messages/common.html`. As it contains such a number of commands, a lot of modules are subscribed to this topic. The most relevant ones are: `mavlink, fmu, px4io, uavcan, camera_trigger, commander, navigator`.

mount_orientation

The `mount_orientation` topic contains information about the mount:

- `attitude_euler_angle`: mount attitude/direction written as Euler angles (rad).

`mount_orientation` receives data from vmount module and publish them on `mavlink`. This topic is only used in the presence of AUX pins, i.e. no Pixracer autopilot.

camera_trigger

The `camera_trigger` topic contains information about camera functionalities:

- `seq`: image sequence number.

- `feedback`: boolean value for checking capture was right.

`camera_trigger` receives data from `camera_trigger` module and publish them on `mavlink` and `camera_feedback`.

3.5 MAVLink

While uORB is responsible for the communication between drone internal components, MAVLink is the messaging protocol that has been designed for the entire drone ecosystem, made of the drone and the ground control station. In particular, "PX4 uses MAVLink to communicate with QGroundControl, and as the integration mechanism for connecting to drone components outside of the flight controller: companion computers, MAVLink enabled cameras etc" [17].
MAVLink protocol defines both the message structure and the serialization criterion at the application layer (it basically defines how information should be passed through the network). These messages are then forwarded to the lower layers (i.e., transport layer, physical layer) to be transmitted to the network. One of the biggest advantages using MAVLink is that due to its lightweight structure it supports different types of transport layers and mediums. In fact, it can be transmitted both through serial telemetry low bandwidth channels, operating in the MHz range, namely 433 MHz, 868 MHz or 915 MHz, or through WiFi and Ethernet (TCP/IP networks). Using sub-GHz frequencies guarantees to reach wide communication ranges to handle the unmanned system [16].
On the other hand, using WiFi or Ethernet interface, MAVLink messages are streamed through IP networks. Both the TCP and UDP connection protocols can be used, depending on the required reliability of the application [16].

First MAVLink version (1.0) was released by Lorenz Meier in 2009, whereas a second version with important improvements was released in 2017 (MAVLink 2.0). In the following section the second version will be elaborated as it is the recommended one.

3.5.1 MAVLink 2.0 frame

As shown in figure 3.5, 11 important fields are present in a MAVLink frame:

1. **STX** is the symbol representing MAVLink start of frame. In MAVLink 1.0 STX is `0XFE`, whereas in MAVLink 2.0 STX is `0xFD`.

2. **LEN** represents the message length in bytes. It is encoded in 1 Byte.

3. **INC FLAGS** are incompatibility flags: flags affecting the message structure. The flags indicate whether the packet contains some features that must be considered when parsing the packet. This flag is not present in MAVLink 1.0.

4. **CMP FLAGS** are compatibility flags. It indicates flags that can be ignored if not understood and it does not prevent the parser from processing the message even if the flag cannot be interpreted. This flag is not present in MAVLink 1.0.

5. **SEQ** denotes the sequence number of the message. It is encoded into 1 Byte and takes values from 0 to 255. Once it reaches 255, the sequence number is reset again to 0 and incremented in each generated message.

6. **SYS ID** represents the System ID. Every unmanned system should have its System ID, in particular, if they are managed by one ground station. The System ID 255 is typically allocated for ground stations.

7. **COMP ID** is the component ID, and it identifies the component of the system that is sending the message. There are 27 hardware types (i.e., components) in MAVLink 1.0.

8. **MSGID** is the message ID: it refers to the type of the message embedded in the payload. In MAVLink 2.0 it is encoded with 24 bits instead of 8, like in the first version.

9. **PAYLOAD** is the actual message. It can contain up to 255 bytes.

10. **CHECKSUM** contains two bytes dedicated for checking the message correctness (1 byte for CKA and 1 byte for CKB). [16].

11. **SIGNATURE**: finally, MAVLink 2.0 uses an optional Signature field of 13 bytes to ensure that the link is tamper-proof. This features significantly improve security aspects of the MAVLink 1.0 as it allows the authentication of the message and verifies that it originates from a trusted source.

The minimum message length using MAVLink 1.0 is 8 bytes (packets without the payload). On the other hand, the maximum length of a MAVLink 1.0 message is 263 bytes (full payload). As far as MAVLink 2.0 is concerned, the minimum length is 11 bytes, whereas the maximum one is 279 bytes [16].

Figure 3.5: MAVLink 2.0 frame. Image source: MAVLink developer guide.

The over-the-wire format for a MAVLink v2 packet looks like the one in figure 3.6.

```
uint8_t magic;              ///< protocol magic marker
uint8_t len;                ///< Length of payload
uint8_t incompat_flags;     ///< flags that must be understood
uint8_t compat_flags;       ///< flags that can be ignored if not understood
uint8_t seq;                ///< Sequence of packet
uint8_t sysid;              ///< ID of message sender system/aircraft
uint8_t compid;             ///< ID of the message sender component
uint8_t msgid 0:7;          ///< first 8 bits of the ID of the message
uint8_t msgid 8:15;         ///< middle 8 bits of the ID of the message
uint8_t msgid 16:23;        ///< last 8 bits of the ID of the message
uint8_t payload[max 255];   ///< A maximum of 255 payload bytes
uint16_t checksum;          ///< X.25 CRC

uint8_t signature[13];      ///< Signature which allows ensuring that the link is tamper-proof (optional)
```

Figure 3.6: MAVLink 2.0 packet structure. Image source: MAVLink developer guide.

3.5.2 QGroundControl

"QGroundControl is the ground control station (GCS) providing full flight control and vehicle setup for PX4 or ArduPilot powered vehicles. It provides easy and straightforward usage for beginners, while still delivering high end feature support for experienced users" [34]. Key features are:

- Full setup/configuration of ArduPilot and PX4 Pro powered vehicles.
- Flight support for vehicles running PX4 and ArduPilot (or any other autopilot that communicates using the MAVLink protocol).
- Mission planning for autonomous flight.
- Flight map display showing vehicle position, flight track, waypoints and vehicle instruments.
- Video streaming with instrument display overlays.
- Support for managing multiple vehicles.
- QGC runs on Windows, OS X, Linux platforms, iOS and Android devices.

In practice it allows to configure on the go the entire drone. On the left side of figure 3.7 it is possible to see all the different categories that can be modified.

- **Airframe**: it allows to customize the airframe of the specific vehicle to be used. Changing the airframe will change the booted files (vehicle configuration and mixers).
- **Radio**: it allows to both calibrate the RC and feed the drone with commands directly from the GCS.
- **Sensors**: it allows to calibrate all sensors. It is required before flying.
- **Flight modes**: it allows to select the flight mode to be used.
- **Power**: it provides power details.
- **Tuning**: it allows to tune some parameters changing on the fly the drone responsiveness (PID tuning).
- **Camera**: it allows to tune camera settings.
- **Parameters**: it allows to configure any parameter just by typing its name and choosing the desired value.

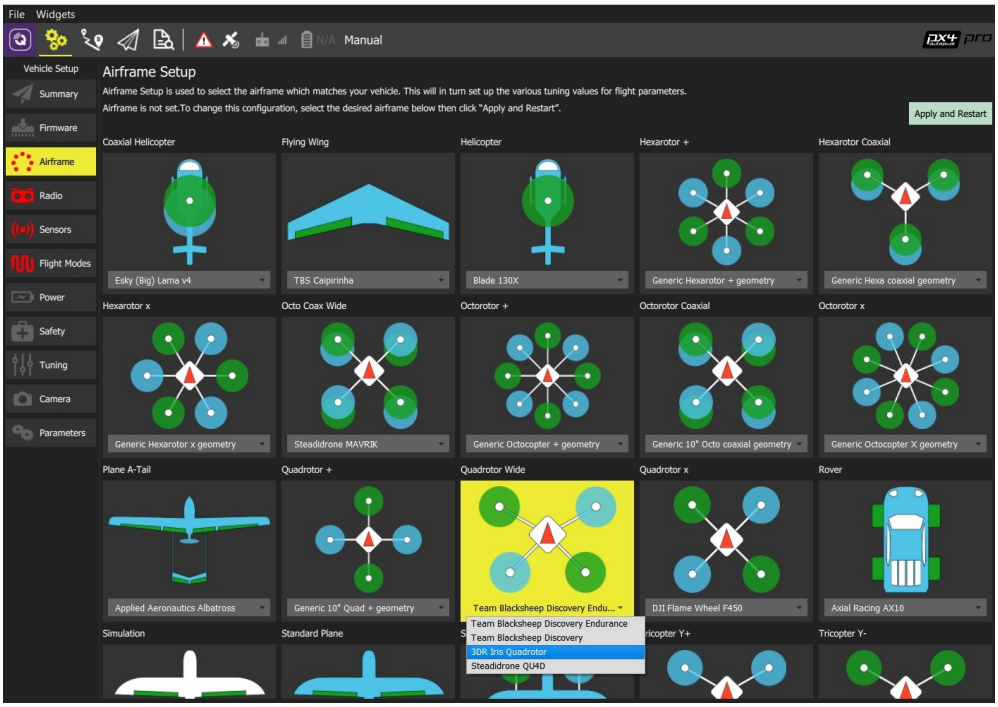

Figure 3.7: PX4 airframes available in QGroundControl.

3.6 Entire software architecture

The goal of this part of the thesis is to delve into the software architecture in order to be able to customize the firmware adding functionalities. Figure 3.10 provides a precise overview of the main PX4 building blocks. As previously mentioned, PX4 architecture can be divided in 2 different parts: the middleware and the flight stack. Middleware blocks are illustrated in the top part of the diagram, while the flight stack components are shown in the lower section. An elaboration of the most relevant parts is now shown.

3.6.1 Storage

Database

Dataman is the module exploited to provide permanent storage for the entire system through a simple database accessible by means of a C API [24]. Multiple backends are supported:

- a file (on the SD card)
- FLASH (if the board supports it)
- FRAM
- RAM (obviously not persistent)

It is used to store structured data of different types, such as mission waypoints, mission state and geofence polygons. Each structure has a specific type and a fixed maximum amount of storage items, so that fast random access is possible [24].

Missions and plans are physically stored within the drone internal memory. In particular, plans are stored in JSON files containing basic information about mission goals and step to be performed (Home position, Rally Points, Geo-fence, Waypoints and mission commands) [37]. A detailed guide about .plan files can be found at: `https://dev.qgroundcontrol.com/en/file_formats/plan.html`. Moreover, SD card allows to store captured images/video to be elaborated afterwards.

Parameters

Configuration parameters are fundamental since they allow to customize drone behaviour. Each module composing the firmware is equipped with a `params.c` file containing all the parameters that can be modified within that specific module. QGroundControl allows to set these parameters in a user-friendly way, without browsing into all the directories to find the correct param file, just typing the name of the parameter to be modified.

Logging

Every flight is registered and saved as a log file on the microSD mounted on the Pixracer. The PX4 log file came in the .ulg file format, a proprietary one that can be managed in different ways:

- pyulog: a set of python scripts made available directly from PX4 Github repository permits (after correct installation and configuration) to convert .ulg files into more common and manageable ones. This tool is particularly useful for topics log; I wrote a short guide on how to log a topic on the SD and how to visualize the .ulg files converting them into csv.
- Online PX4 Flight Review: this online tool permits to upload directly the .ulg flight log and after couple seconds visualize various Graphs representing almost all the parameters necessary to a good analysis of the flight and drone attitude behaviour [37].

The module responsible for logging is logger [24].

3.6.2 External connectivity

MAVLink

MAVLink, as explained in section 3.5, is the communication protocol used to guarantee the communication between the drone and the GCS.

FastRTPS

"The PX4-FastRTPS Bridge adds a Real Time Publish Subscribe (RTPS) interface to PX4, enabling the exchange of uORB messages between PX4 components and (offboard) Fast RTPS applications (including those built using the ROS2/ROS frameworks)" [10].
RTPS should be considered when real-time or time-critical information need to be exchanged between the flight controller and the off-board components (perception computer) in a reliable manner. One of the best RTPS applications is when off-board applications (i.e. running on the perception computer) need to become a peer of software components running on PX4 (i.e. internal modules) by sending and receiving uORB topics. It is essentially used to allow a companion application to be subscribed to a uORB topic in order to use those data.
Computer vision is one of the standard cases where RTPS can be the best solution to handle real-time data flow [10].

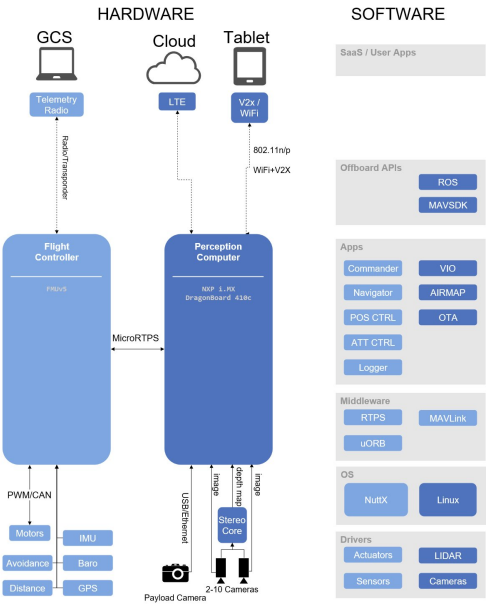

Figure 3.8: Most generic architecture including Fast RTPS. Image source: PX4 developer guide.

It is important to remark how Fast RTPS was not designed to be a MAVLink replacement. MAVLink remains the most suitable communication protocol for sharing information with GCSs, gimbals, cameras, and other off-board components. Nevertheless, FastRTPS could be used to handle particular peripherals needing to interact with PX4 in specific ways with real-time constraints [10].
For our application no perception computer was needed, due to the law limit for ultralight drones fixed at 250g. All the image processing and other complex operations are performed after the mission, analysing data stored inside the onboard SD card.

3.6.3 Drivers

Drivers allow to program physical components, like camera, gimbal, GPS and all the other peripherals. They are fundamental for a correct navigation and mission.

Camera control

Camera control handles the camera trigger (no gimbal). Most useful parameters are:

- **TRIG_PINS**: selects which FMU pin is used. "The PWM interface takes two pins per camera, while relay triggers on every pin individually. Example: Value 56 would trigger on pins 5 and 6. For GPIO mode Pin 6 will be triggered followed by 5. With a value of 65 pin 5 will be triggered followed by 6. Pins may be non contiguous. I.E. 16 or 61. In GPIO mode the delay pin to pin is smaller than 0.2 uS" [12]. Reboot required.

- **TRIG_INTERFACE**: selects the trigger interface; allowed values are:
  - 1: GPIO
  - 2: Seagull MAP2 (over PWM)
  - 3: MAVLink (forward via `MAV_CMD_IMAGE_START_CAPTURE`)
  - 4: Generic PWM (IR trigger, servo)
- **TRIG_POLARITY**: set trigger polarity, 0 means Active low, 1 Active high. This parameter cannot be changed into QGC [12].
- **TRIG_ACT_TIME**: Camera trigger activation time. It sets the time the trigger needs to pulled high or low [12].
- **TRIG_MODE** : Camera trigger mode [12]. Allowed values are:
  - 0: Disable
  - 1: Time based, on command
  - 2: Time based, always on
  - 3: Distance based, always on
  - 4: Distance based, on command (survey mode)

Gimbal/Mount

Whenever willing to control a gimbal (or any payload) connected to the vehicle, it is necessary to specify how PX4 can interact with it.
"PX4 contains a generic mount/gimbal control driver with different input and output methods. The input defines how you control the gimbal: via RC or via MAVLink commands (for example in missions or surveys). The output defines how the gimbal is connected: some support MAVLink commands, others use PWM (described as AUX output in the following). Any input method can be selected to drive any output. Both have to be configured via parameters" [13].
The most important parameters are:

- **MNT_MODE_IN**: mount input mode. It selects how gimbal inputs are given [12] allowed values are:
  - -1: DISABLED
  - 0: AUTO
  - 1: RC
  - 2: MAVLINK_ROI
  - 3: MAVLINK_DO_MOUNT
- **MNT_MODE_OUT**: mount output mode. It defines how the gimbal is moved [12]; allowed values are:
  - 0: AUX
  - 1: MAVLINK

More information can be found at `https://dev.px4.io/v1.9.0/en/advanced/gimbal_control.html`

GPS

The GPS driver is responsible for the position publication via uORB read from the GPS sensor. Depending on the sensor vendor, it may support different protocols; however, by default it autonomously selects the most suitable one. It is possible to specify a secondary GPS device through the -e parameter. Information coming from this secondary sensor will not be used to handle the drone position, they will be only logged to be analysed in post-production checking whether the main GPS was right or not [22].
GPS values are directly used for the position and attitude estimation, without passing trough the sensor hub.

Airspeed, telemetry and distance sensors

Airspeed is correlated with the wind speed estimation, necessary for stability and control purposes. Before being processed, it needs to be corrected, that is why it is sent to the sensor hub. Equivalent Airspeed (EAS) data can be used to reduce drift when GPS is lost by setting `EKF2_ARSP_THR` to a positive value [23].

Telemetry is used to communicate the drone measurement to the GCS and for logging completeness.

Distance sensors are generally laser or ultrasonic, a light or sound impulse is sent from the transmitting apparatus to the outside of the drone, in the desired direction, and the Time Of Flight of the signal bouncing back on the eventual barrier surface determines the distance from the object to be avoided. Also Laser Imaging Detection and Ranging (LIDAR) sensors may be used. LIDAR is a planar laser distance sensor; a prisma rotating 360° takes the distance of objects on the LIDAR plane on N points during the complete rotation, where the number of samples is determined by manufacturer and then by the software. Distance sensor output are directly used by the estimators [37].

RC input

RC input is a fundamental module responsible for reading the input pins and driving the output. As far as the Pixracer is concerned, the MAIN channels are used, since the AUX ones are not present.
It listens on the `actuator_controls` topics, does the mixing and writes the PWM outputs on `actuator_outputs` [22].
"The module is configured via `mode_*` commands. This defines which of the first N pins the driver should occupy. By using `mode_pwm4` for example, pins 5 and 6 can be used by the camera trigger driver or by a PWM rangefinder driver" [22].

IMU

An inertial measurement unit (IMU) is a device that integrates multi-axes, accelerometers, gyroscopes, and other sensors to provide estimation of an objects orientation in space.

Measurements of acceleration, angular rate, and attitude are typical data outputs [12]. Pixracer comes with the following sensors:

- Gyro/Accelerometer: Invensense MPU9250 Accel / Gyro / Mag (4 KHz)
- Gyro/Accelerometer: Invensense ICM-20608 Accel / Gyro (4 KHz)
- Barometer: MS5611
- Compass: Honeywell HMC5983 magnetometer with temperature compensation

IMU output are necessary for the position and attitude estimation as well as the autonomous flight, position controller and attitude and rate controller. Nevertheless, raw data must be adapted before being used for the estimation.

3.6.4 Flight control

Sensors Hub

The sensors module assumes a key role to the entire system. It takes low-level output from drivers, turns them into a more valuable form (filtering), and publishes them, letting the other modules to take benefit of the clean measurements [24].
The provided functionality includes:

- Read sensor drivers output (sensor_gyro, etc.). If there are multiple of the same type, do voting and failover handling. Then apply the board rotation and temperature calibration (if enabled). And finally publish the data; one of the topics is sensor_combined, used by many parts of the system [24].
- Do RC channel mapping: read the raw input channels (input_rc), then apply the calibration, map the RC channels to the configured channels and mode switches, low-pass filter, and then publish as rc_channels and manual_control_setpoint [24].
- Read the output from the ADC driver (via ioctl interface) and publish battery_status [24].
- Make sure the sensor drivers get the updated calibration parameters (scale and offset) when the parameters change or on startup. The sensor drivers use the ioctl interface for parameter updates. For this to work properly, the sensor drivers must already be running when sensors is started [24].
- Do preflight sensor consistency checks and publish the sensor_preflight topic [24].

Position and attitude estimators

EKF2 and LPE are the two main attitude and position estimators involved within the PX4 architecture. The Estimation and Control Library (ECL) relies on Extended Kalman Filter algorithm to process sensor measurements in order to provide an estimation of these states [9]:

- Quaternion defining the rotation from North, East, Down local earth frame to X,Y,Z body frame

- Velocity at the IMU - North,East,Down (m/s)
- Position at the IMU - North,East,Down (m)
- IMU delta angle bias estimates - X,Y,Z (rad)
- IMU delta velocity bias estimates - X,Y,Z(m/s)
- Earth Magnetic field components - North,East,Down (gauss)
- Vehicle body frame magnetic field bias - X,Y,Z (gauss)
- Wind velocity - North,East (m/s)

EKF2 is one of the most difficult concept dealing with PX4, this is why I remind to the official website for more information `https://docs.px4.io/v1.9.0/en/advanced_config/tuning_the_ecl_ekf.html`

An important consideration about the modules involved in each category is summarized in figure 3.9

	Fixed Wing	Multi Copter	VTOL
Navigator	navigator	navigator	navigator
Estimator	ekf_att_pos_estimator	attitude_estimator_q position_estimator_inav	attitude_estimator_q position_estimator_inav
Controller	fw_att_control fw_pow_control_l1	mc_att_control mc_pos_control	vtol_att_control mc_att_control mc_pos_control fw_att_control fw_pow_control_l1

Figure 3.9: Modules associated with each airframe. Image source: how to customize pixhawk in your own project.

State machine

Commander module is used to change the drone state, for instance from armed to disarmed, or to perform basic operations like taking off or land. It is able to receive input both from the sensor hub (RC input) and MAVLink. Data coming from the commander module and position and attitude estimators are then used as inputs for the autonomous flight, position controller and attitude/rate controller [24].

Autonomous flight, position controller, attitude and rate controller

The navigator module, based on the commander input, the mission plan (SD card) and the estimated states, creates position setpoints. Such a setpoints are the inputs for the position controller (as well as the RC input and the estimated states). The position controller, as explained in section 3.2.1, creates attitude setpoints, that are used by the attitude and rate controller to create the actuator controls [21].
Actuator controls are then converted into suitable values to be applied to the actuators (PWM/UART/CAN).

In figure 3.10 grey lines represents raw data, brown lines stand for estimated data and light-blue lines shows RC input.

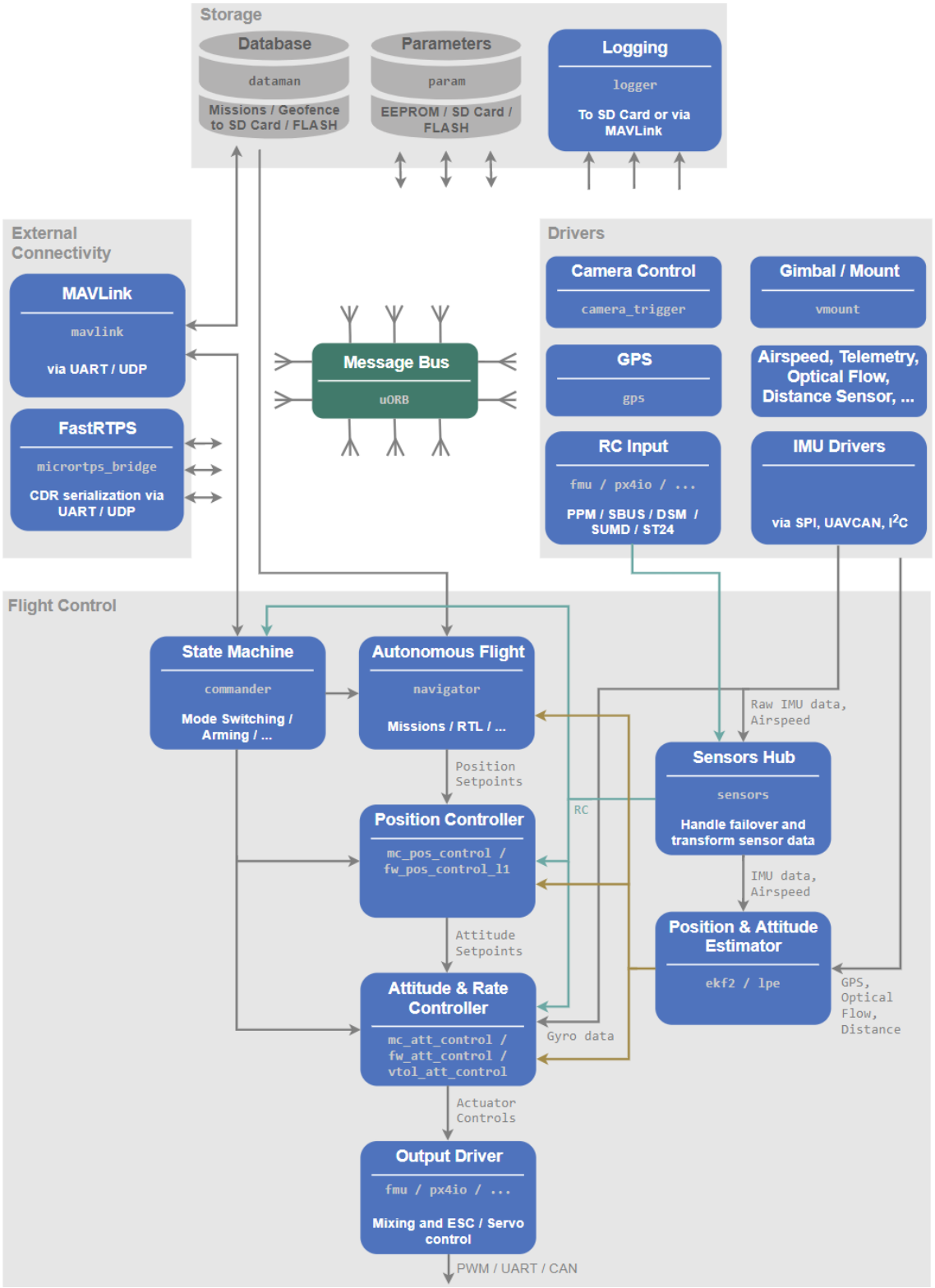

Figure 3.10: PX4 software architecture. Image source: PX4 developer guide.

3.6.5 Flight modes

"Flight Modes define how the autopilot responds to user input and controls vehicle movement. They are essentially grouped into three categories: manual, assisted or autonomous modes, depending on the level of control the autopilot should produce. The pilot transitions between flight modes may be performed using switches on the remote control or with a ground control station" [11].

Manual modes

"Manual" modes are those where the RC control can be used to entirely control the drone by the user: any stick movement is associated to a specific vehicle movement. Nevertheless, changing the manual mode it is possible to change the level of response assigned to specific movements. For instance, skilled fliers shall use modes providing direct passthrough of the stick positions to actuators, while beginners should choose modes guaranteeing less responsiveness to any stick changes [11]. As far as the manual modes is concerned, 3 subsections can be defined:

- **MANUAL/STABILIZED**: throttle is passed directly to the output mixer, meaning that the pilot should always tune the throttle to keep the drone on flight. Pilot's inputs are passed as roll and pitch angle commands and a yaw rate command. In this mode the autopilot controls the attitude, regulating the roll and pitch angles to zero when the RC sticks are centred. Nonetheless, the autopilot can not control the vehicle position, therefore drone can move due to wind [11].

- **ACRO**: as in the manual mode, the throttle is directly passed to the output mixer. Conversely with respect to the previous mode, pilot's inputs are introduced as roll, pitch, and yaw rate commands to the autopilot, while in the previous case roll and pitch were introduced as angle commands (not rate). Angular rates can be controlled by the autopilot, but not the attitude. As a consequence, if the RC sticks are centred, the vehicle will not level-out. This allows the multirotor to perform acrobatic moves such as a complete inversion [11].

- **RATTITUDE**: as in the previous modes, throttle is directly passed to the output mixer. Pilot's inputs are introduced as roll, pitch, and yaw rate commands to the autopilot (like in the ACRO mode) only if they are greater than a specific threshold, namely if the RC sticks are a certain distance away from the centre position. If not, inputs are introduced as roll and pitch angle commands and a yaw rate command as in the manual mode. Such a mode is a hybrid between the two previous ones: up to a certain point it behaves like in the STABILIZED mode, whereas overcome a certain threshold it turns in ACRO mode [11].

Assisted modes

"Assisted" modes are an improvement with respect to manual modes offering "automatic" assistance in specific situations, like holding position/direction against wind. One of the advantages of assisted modes is the capability of restoring controlled flight [11].

- **ALTCTL** (altitude control): roll, pitch and yaw inputs are analogous to the STABILIZED mode. Throttle inputs indicate going up or down at a predetermined maximum speed. Centred Throttle holds altitude steady. The autopilot only controls altitude, hence wind can change the vehicle position [11].

- **POSCTL** (position control): roll controls left-right speed, pitch controls front-back speed with respect to ground and yaw controls the yaw rate similarly to MANUAL mode. Throttle handles climbing/descending rate as in ALTCTL mode. As a consequence vehicle position is held steady by the autopilot against any wind disturbances [11].

Automatic modes

"Auto" modes are the smartest ones, since the controller requires little to no user input [11].

- **AUTO LOITER** (loiter): the multirotor hovers maintaining the current position and altitude [11].

- **AUTO RTL** (return to land): the multirotor returns following a straight line either on the current altitude or at a pre-defined one [11].

- **AUTO MISSION**: the aircraft performs the programmed mission sent by the GCS. If no mission is received, aircraft will LOITER at current position, unless battery drops; in that case, depending on the failsafe mode, it will either return to the home position or simply land wherever it is [11].

3.7 Firmware structure

Once explained all the firmware features, a deep analysis of its components is required. The most relevant parts of the tree structure are proposed in the following tree-like structure:

```
Firmware
├── cmake
├── msg
│   └── templates
│       ├── px4
│       └── uorb
├── ROMFS
│   ├── px4fmu_common
│   │   ├── init.d
│   │   ├── logging
│   │   └── mixers
│   └── px4fmu_test
│       ├── init.d
│       ├── mixers
│       └── unit_test_data
└── src
    ├── drivers
    ├── examples
    ├── modules
    │   ├── attitude_estimator_ekf
    │   ├── attitude_estimator_q
    │   ├── bottle_drop
    │   ├── commander
    │   ├── controllib_test
```

```
25      ├── dataman
26      ├── ekf2
27      ├── ekf2_replay
28      ├── ekf_att_pos_estimator
29      ├── fw_att_control
30      ├── fw_pos_control_l1
31      ├── gpio_led
32      ├── land_detector
33      ├── local_position_estimator
34      ├── mavlink
35      ├── mc_att_control
36      ├── mc_att_control_multiplatform
37      ├── mc_pos_control
38      ├── mc_pos_control_multiplatform
39      ├── muorb
40      ├── navigator
41      ├── param
42      ├── position_estimator_inav
43      ├── px4iofirmware
44      ├── sdlog2
45      ├── segway
46      ├── sensors
47      ├── simulator
48      ├── systemlib
49      ├── uavcan
50      ├── unit_test
51      ├── uORB
52      └── vtol_att_control
53  └── systemcmds
54      ├── bl_update
55      ├── config
56      ├── dumpfile
57      ├── esc_calib
58      ├── i2c
59      ├── mixer
60      ├── motor_test
61      ├── mtd
62      ├── nshterm
63      ├── param
64      ├── perf
65      ├── pwm
66      ├── reboot
67      ├── reflect
68      ├── tests
69      ├── top
70      ├── topic_listener
71      ├── usb_connected
72      └── ver
```

Main components are summarized in the following table [29]:

cmake	make files
msg	uORB msg template, the uORB msg headers are generated from this folder
ROMFS	startup scripts, airframe configuration files and mixers
src/drivers	it contains all the drivers (pwm, gyro, gps etc.)
src/examples	some simple examples help you understand the code
src/modules	estimators, controllers, mavlink, vmount etc.

3.7.1 System commands (systemcdms)

config

It allows to configure a sensor driver (parameters like sampling and communication rate, range etc.). Allowed sensors are: gyro, accelerometers and magnetometer [24].

```
config <command> [arguments...]
 Commands:

 The <file:dev> argument is typically one of /dev/{gyro,accel,mag}i
   block         Block sensor topic publication
     <file:dev>  Sensor device file

   unblock       Unblock sensor topic publication
     <file:dev>  Sensor device file

   sampling      Set sensor sampling rate
     <file:dev> <rate> Sensor device file and sampling rate in Hz

   rate          Set sensor publication rate
     <file:dev> <rate> Sensor device file and publication rate in Hz

   range         Set sensor measurement range
     <file:dev> <rate> Sensor device file and range

   check         Perform sensor self-test (and print info)
     <file:dev>  Sensor device file
```

Figure 3.11: Config arguments. Image source: PX4 developer guide.

Reference link: `https://dev.px4.io/v1.9.0/en/middleware/modules_command.html#config`

ESC calibration (esc_calib)

This is a tool for ESC calibration. Calibration procedure (running the command will guide you through it):

- Remove props, power off the ESC's
- Stop attitude controllers: mc_att_control stop, fw_att_control stop
- Make sure safety is off
- Run the command

Reference link: `https://dev.px4.io/v1.9.0/en/middleware/modules_command.html#esccalib`

Topic listener (listener)

This is a tool to listen to uORB topics printing data to the console [24].
Reference link: `https://dev.px4.io/v1.9.0/en/middleware/modules_command.html#listener`

```
esc_calib [arguments...]
     [-d <val>]  Select PWM output device
                 values: <file:dev>, default: /dev/pwm_output0
     [-l <val>]  Low PWM value in us
                 default: 1000
     [-h <val>]  High PWM value in us
                 default: 2000
     [-c <val>]  select channels in the form: 1234 (1 digit per channel,
                 1=first)
     [-m <val>]  Select channels via bitmask (eg. 0xF, 3)
     [-a]        Select all channels
```

Figure 3.12: ESC calibration arguments. Image source: PX4 developer guide.

```
listener <command> [arguments...]
 Commands:
     <topic_name> uORB topic name
     [-i <val>]  Topic instance
                 default: 0
     [-n <val>]  Number of messages
                 default: 1
     [-r <val>]  Subscription rate (unlimited if 0)
                 default: 0
```

Figure 3.13: Listener arguments. Image source: PX4 developer guide.

Parameters configuration (param)

This is a command to access and manipulate parameters via shell or script. For instance, it is used in the startup script to set airframe-specific parameters. Such parameters are typically stored to SD card or FRAM; command `param select` can be used to change the storage location for sequential savings [24].

Parameter configuration can be easily performed by means of QGC just typing the name of the parameter to be set and setting the value to be assumed by the parameter.

Reference link: `https://dev.px4.io/v1.9.0/en/middleware/modules_command.html#param`

PWM (pwm)

This is the command used to configure PWM outputs for servo and ESC control.

The default device /dev/pwm_output0 are the Main channels, AUX channels are on /dev/pwm_output1 (-d parameter). Remember that in the Pixracer board there are no AUX ports [24].

Reference link: `https://dev.px4.io/v1.9.0/en/middleware/modules_command.html#pwm`

Running processes (top)

Monitor running processes and their CPU, stack usage, priority and state [24].

```
top [arguments...]
   once          print load only once
```

Figure 3.14: Top arguments. Image source: PX4 developer guide.

3.7.2 Communication

MAVLink

This is the module implementing the MAVLink protocol. It is able to communicate with the system through uORB: some messages are handled directly within the module (an example is the mission protocol), others need to be published via uORB (an example is the vehicle_command) [20].
"The implementation uses 2 threads, a sending and a receiving thread. The sender runs at a fixed rate and dynamically reduces the rates of the streams if the combined bandwidth is higher than the configured rate (-r) or the physical link becomes saturated. This can be checked with mavlink status, see if rate mult is less than 1" [20].
Reference link: `https://dev.px4.io/v1.9.0/en/middleware/modules_communication.html#mavlink`

uORB

The module implementing the uORB publication/subscription communication mechanism is uorb. Such a module is started when booting the system up, since most of the other modules need it to run properly. Since the communication is performed using shared memory, "no thread or work queue is needed, the module start only makes sure to initialize the shared global state. The implementation is asynchronous and lock-free, ie. a publisher does not wait for a subscriber and vice versa. This is achieved by having a separate buffer between a publisher and a subscriber" [20].
Messages are defined in the /msg directory. They are converted into C/C++ code at build-time.

Top instance can be used with uorb as well, and it is very useful to check the topic publication rate.

```
uorb top
```

```
uorb <command> [arguments...]
 Commands:
   start

   status        Print topic statistics

   top           Monitor topic publication rates
     [-a]        print all instead of only currently publishing topics
     [-1]        run only once, then exit
     [<filter1> [<filter2>]] topic(s) to match (implies -a)
```

Figure 3.15: uORB arguments. Image source: PX4 developer guide.

Reference link: `https://dev.px4.io/v1.9.0/en/middleware/modules_communication.html#uorb`

3.7.3 Controllers

The following controllers are all within the module directory and the only allowed arguments are: start, stop, status.

```
Commands:
  start

  stop

  status        print status info
```

Figure 3.16: Controller arguments. Image source: PX4 developer guide.

Fixed wing controllers

Fixed wing controllers are divided in two different categories [21]:

- fw_att_control: fixed wing attitude controller.
- fw_pos_control_l1: fixed wing position controller.

High level controller description can be found here: `https://dev.px4.io/v1.9.0/en/flight_stack/controller_diagrams.html`

Multicopter controllers

Multicopter controllers are divided in two different categories [21]:

- mc_att_control: multicopter attitude controller.
- mc_pos_control: multicopter position controller.

High level controller description was elaborated in section 3.2.1.

Navigator

Navigator is the module responsible for autonomous flight modes. Examples of such a flight modes are missions, takeoff and RTL. Moreover, navigator is in charge for geo-fence violation monitoring. "The different internal modes are implemented as separate classes that inherit from a common base class NavigatorMode. The member _navigation_mode contains the current active mode" [21].
The navigator output is a position setpoint triplets (position_setpoint_triplet_s), required as input by the position controller" [21].
Reference link: `https://dev.px4.io/v1.9.0/en/middleware/modules_controller.html#navigator`

```
navigator <command> [arguments...]
 Commands:
   start

   fencefile     load a geofence file from SD card, stored at etc/geofence.txt

   fake_traffic  publishes 3 fake transponder_report_s uORB messages

   stop

   status        print status info
```

Figure 3.17: Navigator arguments. Image source: PX4 developer guide.

3.7.4 Drivers

FMU driver (px4fmu)

This is the module driving the output based on the received inputs. As far as the boards without a separate IO chip (like Pixracer) is concerned, it leverages the main channels. On the other hand, boards with an IO chip (like Pixhawk), it exploits the AUX channels, while the px4io driver is used for main ones.

It listens on the actuator_controls topics (`Firmware/build/px4_fmu-v4_default/uORB/topics`), does the mixing (according to the mixer files in the micro SD) and writes the PWM outputs [22].

"The module is configured via mode_* commands. This defines which of the first N pins the driver should occupy. By using mode_pwm4 for example, pins 5 and 6 can be used by the camera trigger driver or by a PWM rangefinder driver. Alternatively, the fmu can be started in one of the capture modes, and then drivers can register a capture callback with ioctl calls.
By default the module runs on the work queue, to reduce RAM usage. It can also be run in its own thread, specified via start flag -t, to reduce latency. When running on the work queue, it schedules at a fixed frequency, and the pwm rate limits the update rate of the actuator_controls topics. In case of running in its own thread, the module polls on the actuator_controls topic. Additionally the pwm rate defines the lower-level IO timer rates" [22].
Reference link: `https://dev.px4.io/v1.9.0/en/middleware/modules_driver.html#fmu`

GPS

The GPS driver is responsible for the position publication via uORB read from the GPS sensor. Depending on the sensor vendor, it may support different protocols; however, by default it autonomously selects the most suitable one. It is possible to specify a secondary GPS device through the -e parameter. Information coming from this secondary sensor will not be used to handle the drone position, they will be only logged to be analysed in post-production checking whether the main GPS was right or not [22].
GPS values are directly used for the position and attitude estimation, without passing trough the sensor hub.

There is a thread for each device polling for data. The GPS protocol classes are implemented with callbacks so that they can be used in other projects as well (eg. QGroundControl uses them too).

```
gps <command> [arguments...]
 Commands:
   start
     [-d <val>]  GPS device
                 values: <file:dev>, default: /dev/ttyS3
     [-b <val>]  Baudrate (can also be p:<param_name>)
                 default: 0
     [-e <val>]  Optional secondary GPS device
                 values: <file:dev>
     [-g <val>]  Baudrate (secondary GPS, can also be p:<param_name>)
                 default: 0
     [-f]        Fake a GPS signal (useful for testing)
     [-s]        Enable publication of satellite info
     [-i <val>]  GPS interface
                 values: spi|uart, default: uart
     [-p <val>]  GPS Protocol (default=auto select)
                 values: ubx|mtk|ash|eml

   stop

   status        print status info

   reset         Reset GPS device
     cold|warm|hot Specify reset type
```

Figure 3.18: GPS arguments. Image source: PX4 developer guide.

Reference link: `https://dev.px4.io/v1.9.0/en/middleware/modules_driver.html#gps`

Simulation PWM signals (pwm_out_sim)

Driver for simulated PWM outputs. "Its only function is to take actuator_control uORB messages, mix them with any loaded mixer and output the result to the actuator_output uORB topic" [22].
It is used both in SITL and HITL.
Reference link: `https://dev.px4.io/v1.9.0/en/middleware/modules_driver.html#pwmoutsim`

Gimbal control

Mount (Gimbal) control driver. It maps several different input methods (eg. RC or MAVLink) to a configured output (eg. AUX channels or MAVLink). This makes sure that each input method can be used with each output method and new inputs/outputs can be added with minimal effort [22].

Boot process

When powering on the board, the bootloader will run first. Bootloader is like BIOS in a general purpose PC. And it's already in the board when you buy it. The bootloader will

```
vmount <command> [arguments...]
 Commands:
   start

   test          Test the output: set a fixed angle for one axis (vmount must
                 not be running)
     roll|pitch|yaw <angle> Specify an axis and an angle in degrees

   stop

   status        print status info
```

Figure 3.19: Vmount arguments. Image source: PX4 developer guide

launch the Nuttx Operating System. After some initialization of the hardware, memory, peripherals etc [41]. During the boot process the main performed steps are the following ones:

- Read the parameter file
- Start the sensor drivers (script rc.sensors)
- Set and load the mixer corresponding to the airframe parameter SYS_AUTOSTART, set the pwm channel (script rc.autostart, this file is generated after you build the code)
- Start the flight tasks corresponding to the airframe parameter SYS_AUTOSTART (script rc.fw_apps, rc.mc_apps, etc.) [41].

3.8 Mixing and actuators

3.8.1 Autopilot hardware architecture

Before delving into the mixing and actuator section, it is necessary to elaborate the autopilot hardware architecture. Figure 3.20 shows the generic structure of a Pixhawk flight controller underlying differences between the FMU and I/O boards [14]. The I/O ports are used to handle the drone motors, whereas the FMU ports (AUX) are used to handle payload and gimbal. Unfortunately, the Pixracer board does not have the I/O component and it is only equipped with 6 PWM output pins (MAIN ones).
Flight controllers without an I/O board have MAIN ports (the number depend on the specific board, for instance Pixracer has only 6 pins), but they do not have AUX ports. As a consequence, they should be involved in those airframes not relying on AUX ports. The typical scenario is related to standard multicopters and fully autonomous vehicles, as these applications only need MAIN ports for motor controls and other basic peripherals (such as camera and single-axis gimbal) [14]. It is clear how the ports number may be a strong limitation: assuming to use the Pixracer board (6 pins) for a hexacopter application, no camera or single-axis gimbal could be connected since no port is available (all used to handle the 6 motors). Why using Pixracer boards then? Because they are extremely powerful considering the poor weight.
Figure 3.21 shows standard outputs for a generic quadrocopter like the one we are using.

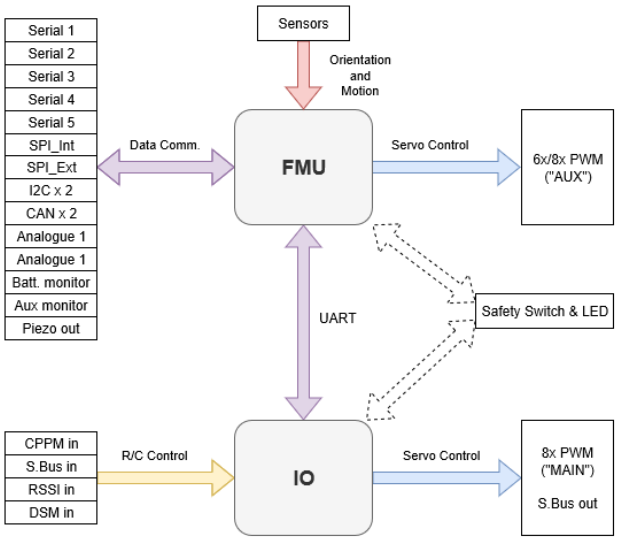

Figure 3.20: Generic autopilot hardware architecture. Image source: PX4 developer guide.

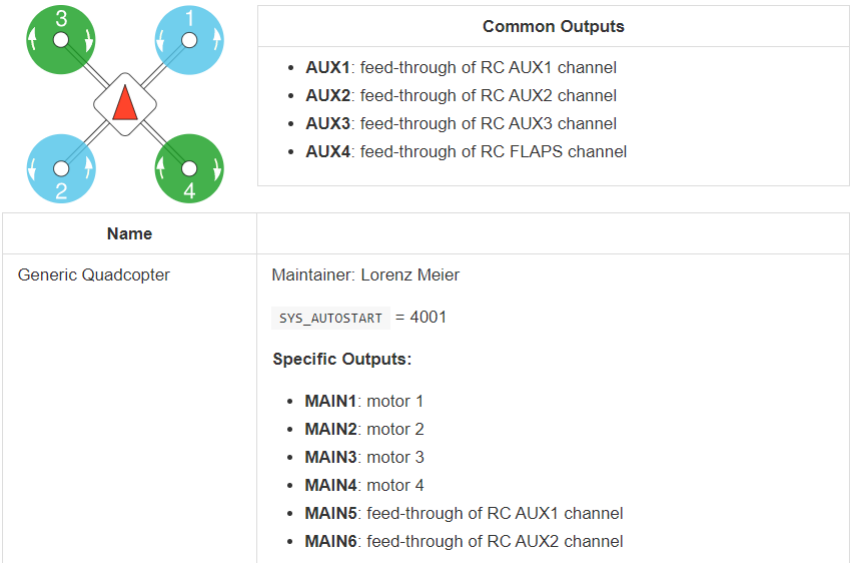

Figure 3.21: Generic quadrocopter outputs. Image source: PX4 developer guide.

3.8.2 Mixing and actuators

Mixing means translating high-level commands (i.e. torque commands such as yaw about 30 degrees), into actuator inputs.
Separating the mixer logic from the attitude controller strongly improves reusability, since the same attitude controller can be used to handle different airframes, where each airframe has a specific mixer file, namely a custom way to convert high-level inputs to actuator commands.

Control pipeline

The control pipeline can be summarized in this way: a particular controller (piece of software) sends a specific normalized force or torque query (scaled within the range -1 to +1) to the mixer, which then converts the input value into individual actuator commands.

Right after, the output driver (generally UART, UAVCAN or PWM) will scale it to the actuators specific units, for instance, considering a PWM driver, it may be a value of 1300 [19], which is sent to the actual actuator after being converted in a compatible signal.

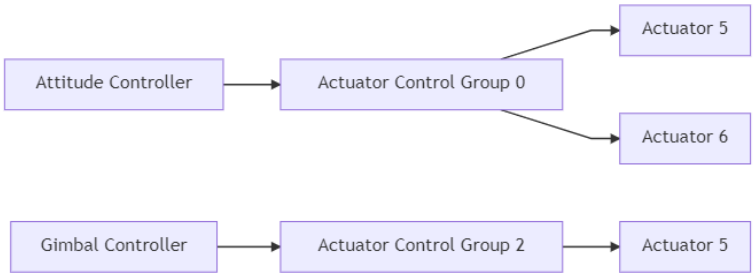

Figure 3.22: Control pipeline. Image source: PX4 developer guide.

3.8.3 Control Groups

PX4 uses control groups and output groups to manage inputs and outputs respectively. A control group is used for the flight controls (attitude), or payloads (gimbal for instance). On the other hand, an output group is a physical bus. Each of these groups has 8 normalized (from -1 to +1) command ports, that can be mapped and scaled through the mixer. A mixer defines how each of these 8 signals of the controls are connected to the 8 outputs [19].
Examples of control groups are the following ones:

Control Group #0 (Flight Control)

- 0: roll (-1..1)
- 1: pitch (-1..1)
- 2: yaw (-1..1)
- 3: throttle (0..1 normal range, -1..1 for variable pitch / thrust reversers)
- 4: flaps (-1..1)
- 5: spoilers (-1..1)
- 6: airbrakes (-1..1)
- 7: landing gear (-1..1)

Figure 3.23: Flight Control Group. Image source: PX4 developer guide.

Figure 3.24: Gimbal Control Group. Image source: PX4 developer guide.

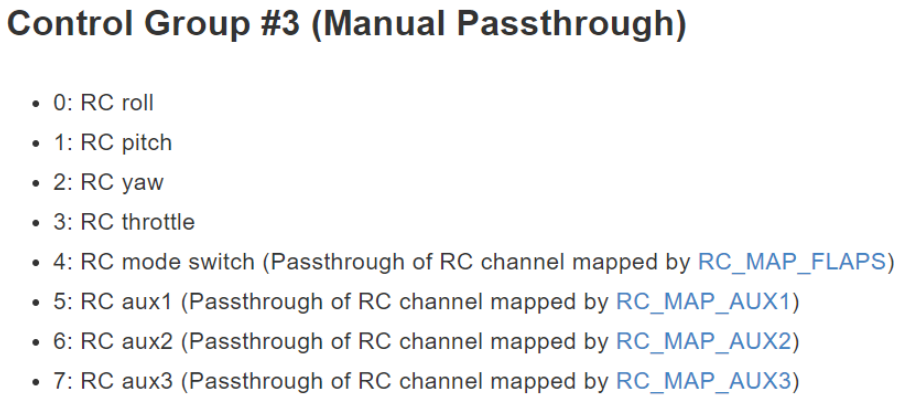

Figure 3.25: Manual passthrough Control Group. Image source: PX4 developer guide.

3.8.4 Output Group/Mapping

An output group is one physical bus (e.g. FMU PWM outputs, IO PWM outputs, UAVCAN etc.) that has N (usually 8, unless using Pixracer, which has 6) normalized (-1..+1) command ports mapped and scaled using mixer files.

The mixer file does not explicitly define the actual output group (physical bus) where the outputs are applied. Yet, its purpose (that may be to control MAIN or AUX outputs) is deduced from the mixer filename; the rc.interface module will perform the mapping to the correct physical bus.

This approach is needed because the physical bus used for MAIN outputs is not always the same; it depends on whether or not the flight controller has an IO Board or uses UAVCAN for motor control [19].

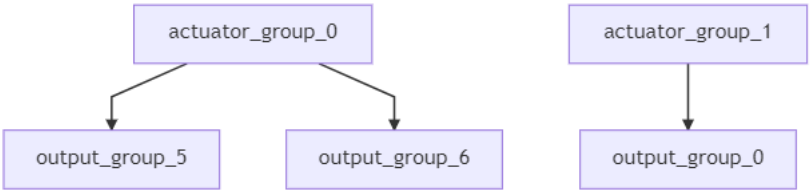

Figure 3.26: Input-Output relationships. Image source: PX4 developer guide.

In practice, the startup scripts only load mixers into a single device (output group). This is a configuration rather than technical limitation; you could load the main mixer into multiple drivers and have, for example, the same signal on both UAVCAN and the main pins [19].

At the end of the day, what should we do if we want to command a specific pin? We need to write a piece of code that publishes the right value on the right topic (control group), then, depending on the mixer file that is loaded, which could be either the default one, or a modified version (according to your needs), the firmware will convert the received control value to the corresponding output, suitable for the actuator connected to the pin. As an example, if someone needs to publish the maximum value of pin 5 (in PWM mode), he will first write a piece of code publishing the value 1 to the control group number 3 (in the 6th value of the array, according to the definition in figure 3.25). Then, the mixer file will be written so that 6th value of control group 3 will drive pin 5 on the main outputs. In such a way, any control value published on the 6th value of the control group 3 will be mapped with pin 5 on the board.

3.8.5 Mixer definition

Mixer files

A mixer file must be named XXXX.main.mix if it is responsible for the mixing of MAIN outputs or XXXX.aux.mix if it mixes AUX outputs. In our specific case, only XXXX.main.mix files will be used, because Pixracer board does not have them [19].

Mixer loading

The default set of mixer files (in Firmware) are defined in px4fmu_common/init.d/airframes/. These can be overridden by mixer files with the same name in the SD card directory /etc/mixers/ (SD card mixer files are loaded by preference).
PX4 loads mixer files named XXXX.main.mix onto the MAIN outputs and YYYY.aux.mix onto the AUX outputs, where the prefixes depend on the airframe and airframe configuration. Commonly the MAIN and AUX outputs correspond to MAIN and AUX PWM outputs, but these may be loaded into a UAVCAN (or other) bus when that is enabled [19].

How to find the prefix of the main mixer you have to use?

1. You need to know the airframe you are using: go on page `https://dev.px4.io/master/en/airframes/airframe_reference.html`. Remember that if you are using a board without I/O component, you cannot select an airframe with the gimbal, because the built-in airframe considers using the AUX pins that you do not have on your board.

2. Go on page `https://github.com/PX4/Firmware/tree/master/ROMFS/px4fmu_common/init.d/airframes` and open the airframe you chose.

3. Within the file, at the end, there will be a line similar to 'set MIXER quad_x'. The name you see after 'MIXER' is the prefix. In this case the name would be: quad_x.main.mix.

Loading a custom mixer

By default PX4 loads properly named mixer files from the SD card directory /etc/mixers/. To load a custom mixer, it is enough to give it the same name as a "normal" mixer file (that is going to be loaded by the chosen airframe) and put it in the etc/mixers directory on the flight controller's SD card [19].

Syntax

Mixer definitions are text files; lines beginning with a single capital letter followed by a colon are significant. All other lines are ignored, meaning that explanatory text can be freely mixed with the definitions, there is no need to add special characters at the beginning of the line [19].

A mixer begins with a line of the form:

```
<tag>:  <mixer arguments>
```

The tag defines the mixer type; 'M' for a simple summing mixer, 'R' for a multirotor mixer, etc. the mixer arguments specifies the kind of operations that should be done with the input value.

Null mixer

A null mixer receive no controls and produces a single actuator output whose value is always zero. Typically a null mixer is used as a placeholder in a collection of mixers in order to achieve a specific pattern of actuator outputs (there may be situations where, for whatever reason, an output group is skipped, like in the wing-wing configuration) [19]. The null mixer definition has the form:

```
Z:
```

Simple mixer

A simple mixer combines zero or more control inputs into a single actuator output. Inputs are scaled, and the mixing function sums the result before applying an output scaler [19]. A simple mixer definition begins with:

```
M: <control count>
O: <-ve scale> <+ve scale> <offset> <lower limit> <upper limit>
```

If `<control count>` is zero, the sum is effectively zero and the mixer will output a fixed value that is `<offset>` constrained by `<lower limit>` and `<upper limit>`.
The second line defines the output scaler with scaler parameters as discussed above. Whilst the calculations are performed as floating-point operations, the values stored in the definition file are scaled by a factor of 10000; as an example an offset of -0.5 is encoded as -5000 [19].
The definition continues with `<control count>` entries describing the control inputs and their scaling, using the form:

```
S: <group> <index> <-ve scale> <+ve scale> <offset> <lower limit> <upper limit>
```

The `S:` lines must be below the `O:` line.

The `<group>` value identifies the control group to be considered, and the `<index>` value represents the element to be considered within that group. When used to mix vehicle controls, mixer group zero is the vehicle attitude control group, and index values from zero to three are normally roll, pitch, yaw and thrust respectively [19].

Wing wing example

The example refers to the Wing Wing architecture. The original file name is wingwing.main.mix and can be found at `https://github.com/PX4/Firmware/blob/master/ROMFS/px4fmu_common/mixers/wingwing.main.mix`
The mixer file contains several blocks of code, each of which refers to one actuator or ESC. So if you have e.g. two servos and one ESC, the mixer file will contain three blocks of code [1].
According to the specifications, MAIN1 would be the left aileron, MAIN2 the right aileron, MAIN3 is empty (note the Z: zero mixer) and MAIN4 is throttle (to keep throttle on output 4 for common fixed wing configurations).
A mixer is encoded in normalized units from -10000 to 10000, corresponding to -1..+1. The first block (first servo) is shown and commented below [1] [19].

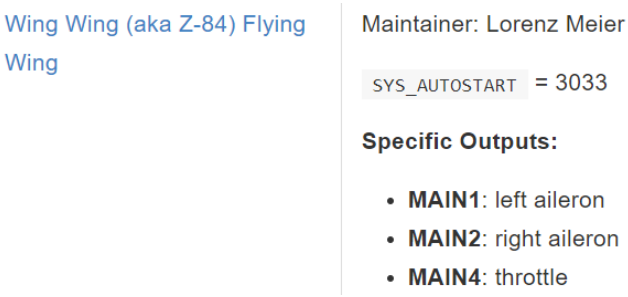

Figure 3.27: Wing Wing airframe specifications. Image source: PX4 developer guide.

```
M: 2
O: 10000 10000 0 -10000 10000
S: 0 0 -6000 -6000 0 -10000 10000
S: 0 1 6500 6500 0 -10000 10000
```

Where each number from left to right means:

- M: Indicates two scalers for two control inputs. It indicates the number of control inputs the mixer will receive. Having M = 2 means having two different S: declarations [1].
- O: Indicates the output scaling (*1 in negative, *1 in positive), offset (zero here), and output range (-1..+1 here). If you want to invert your PWM signal, the signs of the output scalings have to be changed (as it was done in the first S: inside the previous declaration) [1]:
  `O: -10000 -10000 0 -10000 10000`
- S: Indicates the first input scaler: it takes input from control group 0 (Flight Control) and the first input (roll). The first 0 represents the group number (in this case it is the flight control, i.e. group 0), the second 0 represents the component inside the group control (0 means roll according to figure 3.28) [1].

  It scales the roll control input * 0.6 and reverts the sign (because there is negative sign -0.6 becomes -6000 in scaled units). It applies no offset (0) and outputs to the full range (-1..+1) [1].
- S: Indicates the second input scaler: it takes input from control group 0 (Flight Control) and the second input (pitch, because the second component of the control group 0 is pitch). It scales the pitch control input * 0.65 (here there is no negative sign). It applies no offset (0) and outputs to the full range (-1..+1) [1].

The bottom line is that this mixer output would be SERVO = ((roll input * -0.6 + 0) * 1 + (pitch input * 0.65 + 0) * 1) * 1 + 0 [1].

More detail about mixer file examples at `https://dev.px4.io/master/en/airframes/adding_a_new_frame.html#mixer-file`

Multirotor mixer

As far as the multirotor mixers is concerned, things are a bit different from the simple example seen above, because roll, pitch and yaw cannot be controlled directly acting on a single motor.

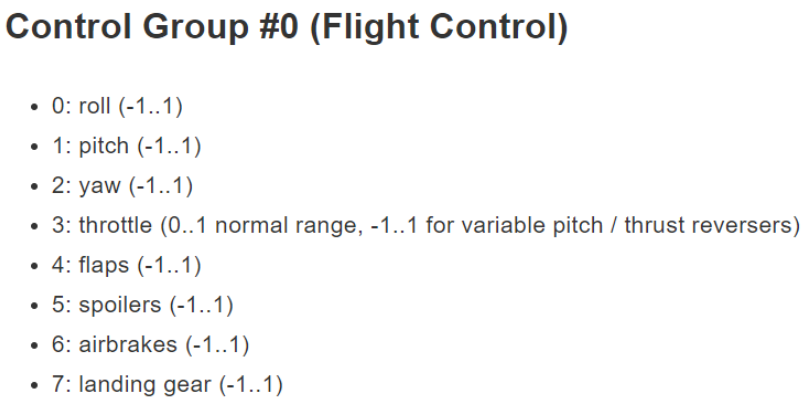

Figure 3.28: Flight Control Group. Image source: PX4 developer guide.

The multirotor mixer combines four control inputs (roll, pitch, yaw, thrust) within the range -1 to +1, except thrust that is between 0 and 1, into a bundle of actuator outputs intended to drive motor speeds [19].
The mixer definition is a single line of the form:

```
R: <geometry> <roll scale> <pitch scale> <yaw scale> <idlespeed>
```

Supported geometries are:

- 4x - quadrotor in X configuration
- 4+ - quadrotor in + configuration
- 6x - hexacopter in X configuration
- 6+ - hexacopter in + configuration
- 8x - octocopter in X configuration
- 8+ - octocopter in + configuration

Each of the roll, pitch and yaw scale values determine scaling of the roll, pitch and yaw controls relative to the thrust control. Whilst the calculations are performed as floating-point operations, the values stored in the definition file are scaled by a factor of 10000; i.e. a factor of 0.5 is encoded as 5000 [19].

Roll, pitch and yaw inputs are expected to range from -1.0 to 1.0, whilst the thrust input ranges from 0.0 to 1.0. Output for each actuator is in the range -1.0 to 1.0.
Idlespeed can range from 0.0 to 1.0. Idlespeed is relative to the maximum speed of motors and it is the speed at which the motors are commanded to rotate when all control inputs are zero [19].
Whenever an actuator saturates, all actuator values are rescaled so that the saturating component is limited to 1.0.

A real example is the one associated with the standard quadrocopter quad_x.main.mix:

```
R: 4x 10000 10000 10000 0

AUX1 Passthrough
M: 1
S: 3 5  10000  10000      0 -10000  10000

AUX2 Passthrough
M: 1
S: 3 6  10000  10000      0 -10000  10000

Failsafe outputs
The following outputs are set to their disarmed value
during normal operation and to their failsafe falue in case
of flight termination.
Z:
Z:
```

The first line defines the geometry (4x) and the scaling factor for the three allowed motions (roll, pitch and yaw), plus the idle speed (0), so no motor speed = 0 whenever there is no other input [19].

AUX1 Passthrough is defined with a single input (`M: 1`), therefore only one `S:` is defined. In this case `S: 3 5` points at control group 3 (manual passthrough, figure 5.1) element 5 (RC aux1, mapped by parameter RC_MAP_AUX1). If you want to associate a specific RC channel to aux1 you should change parameter RC_MAP_AUX1. What does this mean? That pin 5 will follow the RC command associated with the selected channel, for instance, choosing channel 12, pin 5 will follow RC movements in channel 12. Then the following numbers define the mapping between the input and the output. Both the first and the second ones are 10000, meaning that no sign inversion is needed (since it is a positive value), and that the slope is standard. The third value is the offset; being it 0, nothing shall be added. Last two values are -10000 and 10000, therefore the output range is maximum (1000 to 2000).

AUX2 Passthrough is defined with a single input (`M: 1`), therefore only one `S:` is defined. In this case `S: 3 6` points at control group 3 (manual passthrough, figure 5.1) element 6 (RC aux2, mapped by parameter RC_MAP_AUX2). If you want to associate a specific RC channel to aux2 you should change parameter RC_MAP_AUX2. What does this mean? That pin 6 will follow the RC command associated with the selected channel, for instance, choosing channel 10, pin 6 will follow RC movements in channel 10.

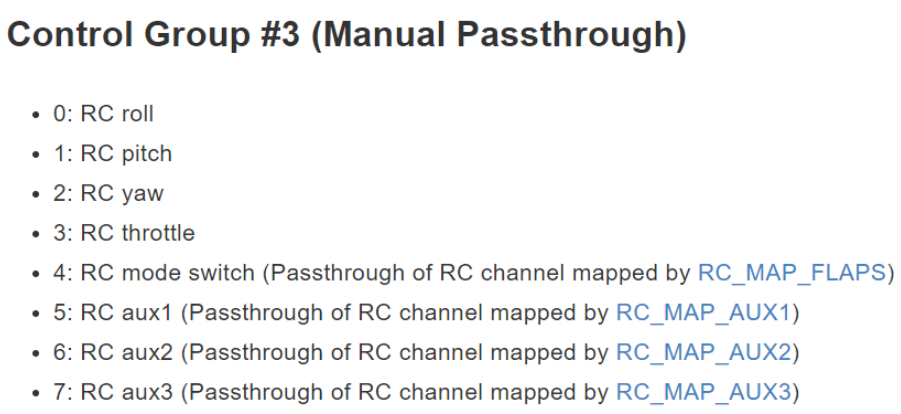

Figure 3.29: Manual Passthrough Control Group. Image source: PX4 developer guide.

How to link a new mixer to an existing airframe?

Before solving this problem it is necessary to understand how a mixer file is linked to an airframe, or better to its configuration file. PX4 uses canned airframe configurations as starting point for airframes. The configurations are defined in config files that are stored in the `ROMFS/px4fmu_common/init.d` folder. The config files reference mixer files that describe the physical configuration of the system, and which are stored in the `ROMFS/px4fmu_common/mixers` folder [1].
Mixers are specified within the configuration file in a line similar to this:

```
set MIXER quad_x
```

In this specific case, the linked mixer is `quad_x.main.mix` located in `ROMFS/px4fmu_common/mixers` folder.
Different options are possible now:

1. Assuming that you want to just modify the original mixer file `quad_x.main.mix`, you can create a copy of it inside the directory `/etc/mixers` and modify it. Then, in order to use it, you have to create a file named `config.txt` inside directory `/etc` containing the line

   ```
   set MIXER quad_x
   ```

   More information about this operation can be found at: `https://dev.px4.io/v1.9.0/en/concept/system_startup.html#starting-a-custom-mixer`

2. Assuming that you created a new airframe in the folder `ROMFS/px4fmu_common/init.d` named `4003_quad_x_v2` and a new mixer `quad_x_v2.main.mix`, you just have to place the correct name inside the configuration file:

   ```
   set MIXER quad_x_v2
   ```

   Then, setting the new airframe within QGC, the correct mixer file will be loaded. More information about adding new airframes at: `https://dev.px4.io/v1.9.0/en/airframes/adding_a_new_frame.html#mixer-file`

3.8.6 MAVLink commands

In previous sections inputs were given through RC, nevertheless, this is not the only possibility to accomplish tasks, in fact it is possible to write a text file containing a pre-defined mission to be completed by the drone. These text files are named plan files. Plan files are stored in JSON file format and contain mission items and (optional) geo-fence and rally-points components. An example plan file is shown and discussed below; it is a customization of the mission implemented by Simone Silvestro in its thesis [37].

```
{
  " fileType ": " Plan ",
  " geoFence ": {
    " circles ": [
    ],
    " polygons ": [
    ],
    " version ": 2
  },
  " groundStation ": " QGroundControl ",
  " mission ": {
    " cruiseSpeed ": 15,
    " firmwareType ": 12,
    " hoverSpeed ": 1,
    " items ": [
      {" autoContinue ": true ," command ": 530," doJumpId ": 1,
      " frame ": 2," params ": [0, 0, null , null , null ,
      null , null ]," type ": " SimpleItem "
      },
      {" autoContinue ": true ," command ": 2000," doJumpId ": 2,
      " frame ": 2," params ": [0, 0, 1, null , null , null,
      null ]," type ": " SimpleItem "
      },
      { " AMSLAltAboveTerrain ": null , " Altitude ": 5, "
      AltitudeMode ": 1, " autoContinue ": true , " command "
      : 22, " doJumpId ": 3, " frame ": 3, " params ": [ 15,
      0, 0, null , 45.061850921636996, 7.662915040275976, 5],
      " type ": " SimpleItem "
      },
      { " autoContinue ": true , " command ": 203, " doJumpId ":
      4, " frame ": 2, " params ": [ 0, 0, 0, 0, 1, 0, 0 ],
      " type ": " SimpleItem "
      },
      { " autoContinue ": true , " command ": 93, " doJumpId ":
      5, " frame ": 2, " params ": [ 20, 0, 0, 0, 0, 0, 0 ],
      " type ": " SimpleItem "
      },
      { " autoContinue ": true , " command ": 203, " doJumpId ":
      6, " frame ": 2, " params ": [ 0, 0, 0, 0, 1, 0, 0 ],
      " type ": " SimpleItem "
      },
      { " autoContinue ": true , " command ": 20, " doJumpId ":
      7, " frame ": 2, " params ": [ 0, 0, 0, 0, 0, 0, 0 ],
      " type ": " SimpleItem "
      }
    ],
    " plannedHomePosition ": [45.06184801299343, 7.6629065558
    804825, 247],
    " vehicleType ": 2,
    " version ": 2
  },
  " rallyPoints ": {
    " points ": [],
    " version ": 2
  },
  " version ": 1
}
```

Listing 3.1: Plan file example taken from Simone Silvestro's work.

Plan files are essentially divided in 3 different sections:

- **Geofence**: define the eventual boundaries in therms of circles and/or polygons. It is optional.
- **Mission**: here the real mission plan is defined, with SimpleItems (basically points or timing where commands have to be executed) or ComplexItems (such as Surveys, Corridor Scan, Structure scan, automatized by the ground station) [37] [30]. SimpleItems are divided into two main fields:
  - **Waypoints**: points of the path between the takeoff and the land used one by one as position setpoints by the vehicle running the mission.
  - **Commands**: actions to be executed during the mission, like maintaining hover for a certain amount of seconds or trigger a camera for example.

  At the end of the mission string is present the position in terms of latitude, longitude and heights in MAMSL. Mission must contain at least one item [30].

  Before defining the items, some parameters are initialized:

  - `cruiseSpeed`: the default forward speed for Fixed wing or VTOL vehicles (i.e. when moving between waypoints) in m/s.
  - `firmwareType`: it defines the autopilot type. A complete list can be found at `https://mavlink.io/en/messages/common.html#MAV_AUTOPILOT`. 12 is related to PX4.
  - `hoverSpeed`: the default forward speed for multi-rotor vehicles.
  - `vehicleType`: the vehicle type for which this mission was created. A complete list can be found at `https://mavlink.io/en/messages/common.html#MAV_TYPE`. 2 is related to quadrorotor [30].
- **Rally points**: optional safe points where vehicle can land in case of failsafe before the completion of the mission [37][30].

Mission items

Mission items are the core of the mission plan. The example 3.1 allows to describe the main components of a mission item:

- `type`: SimpleItem for a simple item, ComplexItem for a complex item.
- `autoContinue`: Autocontinue to next waypoint.
- `command`: This is the actual command that will be performed. The command list can be found at `https://mavlink.io/en/messages/common.html`
- `doJumpId`: The target id for the current mission item in DO_JUMP commands. These are auto-numbered starting from 1.
- `params`: Parameters needed by the specific command.

In example 3.1 the first command is 22, corresponding to quadrocopter takeoff. Then it is followed by command 203 that starts the camera recording. Then the vehicle is forced to hold position for 20 seconds with "command" = 93, to have a small footage of 20 second flight, and then again the camera control to stop the recording just before the Return to Land command send with the MAVLink message number 20 [37].

The following table contains a short connection with all the used commands:

22	Takeoff	`https://mavlink.io/en/messages/common.html#MAV_CMD_NAV_TAKEOFF`
203	Camera trigger	`https://mavlink.io/en/messages/common.html#MAV_CMD_DO_DIGICAM_CONTROL`
93	Hold position	`https://mavlink.io/en/messages/common.html#MAV_CMD_NAV_DELAY`
20	RTH	`https://mavlink.io/en/messages/common.html#MAV_CMD_NAV_RETURN_TO_LAUNCH`

3.9 Supported communication protocols

In order to analyse the communication protocols involved in PX4 it is necessary to understand the basic hardware architecture, as shown in figure 3.30.

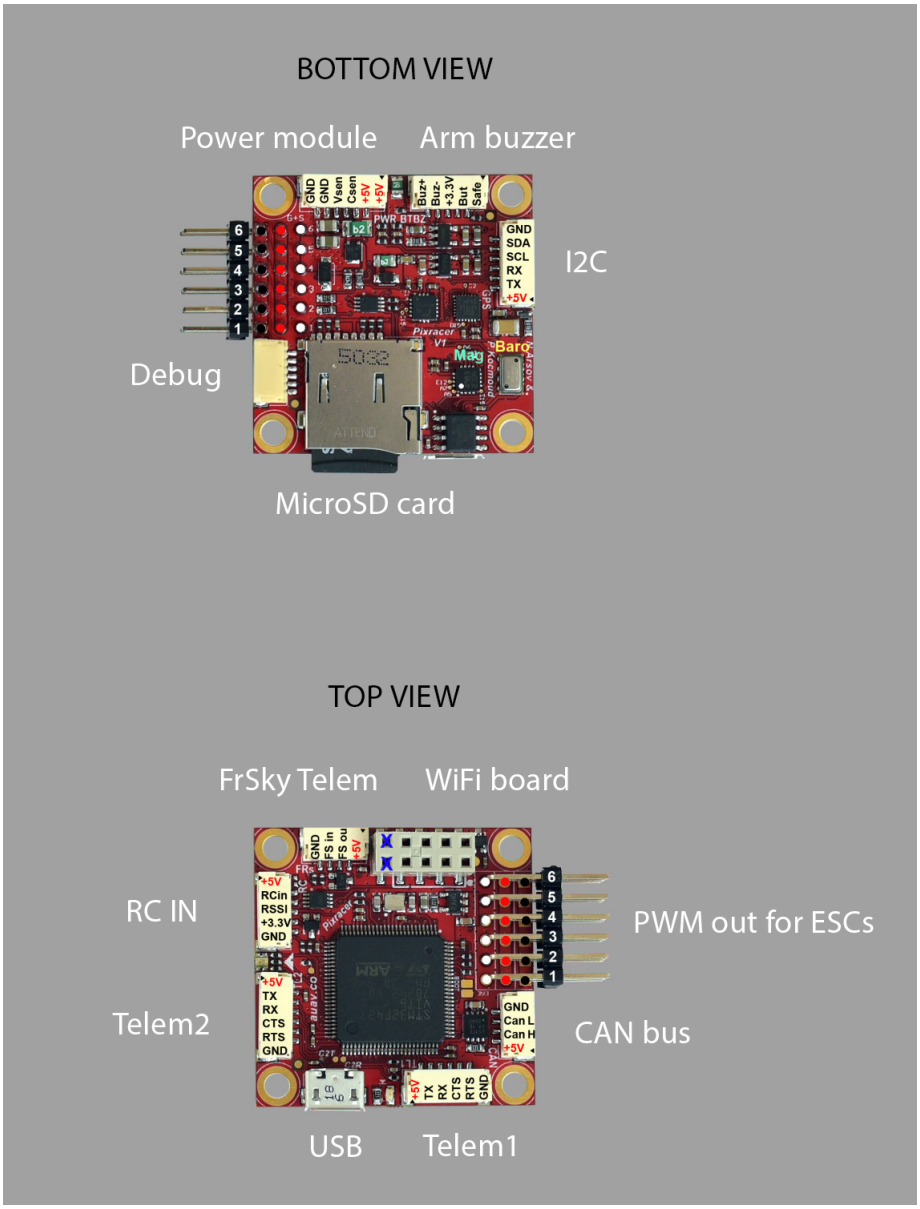

Figure 3.30: Top and bottom views of Pixracer board. Image source: PX4 user guide

Power module

The power module is connected to the battery and provides power to the entire board.

Arm buzzer

The arm buzzer is a safety switch that can be connected to a button to arm/disarm the drone physically.

I2C

I2C is a packet-switched serial communication protocol allowing multiple master devices to be connected to multiple slave devices using only 4 wires per connection (SDA, SCL, VCC and GND). It is mainly intended for connecting lower-speed peripheral ICs (generally within the range 100 kHz to 400 kHz) to processors and microcontrollers in short-distance or intra-board communication [15].
PX4 can use it for:

- Connecting off board components requiring higher data rates than provided by a direct serial UART; an example may be rangefinders.
- Connecting peripheral devices supporting I2C only.
- Allowing multiple devices to attach to a single bus (mainly to preserve the port number).

Important remark: IMUs (accelerometers/gyroscopes) should not be attached via I2C (typically the SPI bus is used). The bus is not fast enough even with a single device attached to allow vibration filtering (for instance), and the performance degrades further with every additional device on the bus [15].

Figure 3.31 shows the I2C communication schematic.

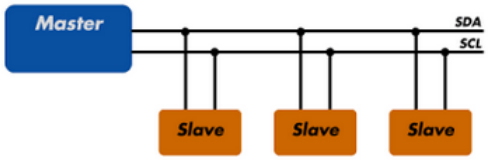

Figure 3.31: I2C communication schematic. Image source: ArduPilot guide

MicroSD

It allows to connect a MicroSD card necessary for logging purposes and loading custom mixer files.

Debug

The debug port is used for debugging purposes; it requires specific hardware for interacting with the firmware. More information at `https://dev.px4.io/v1.9.0/en/debug/system_console.html`

FrSky Telem

It allows the connection to the telemetry receiver.

WiFi board

Our platform embeds the ESP8266 component, which is a tiny WiFi module to create an access point on the drone to exchange data with companion computer.

PWM out

It sends PWM signals to be elaborated by the ESC. It has 6 ports, 4 are used by the rotors, 2 can be used for camera and gimbal.

UART

"A universal asynchronous receiver-transmitter (UART) is a computer hardware device for asynchronous serial communication in which the data format and transmission speeds are configurable" [44]. Being it an asynchronous communication, the data consistency is guaranteed specifying the correct baud rate (which must be the same for both the transmitter and the receiver). It generally requires 4 pins (GND, VCC, TX, RX). A practical example is the Arduino serial port used for debugging purposes.
Its main features are:

- Master-slave communication.
- Character-based protocol suitable for longer connections with respect to I2C and SPI.
- Intermediate speed: up to 1.5 Mbps.
- 4 pin required.

UAVCAN

"UAVCAN is an open lightweight protocol designed for reliable intravehicular communication in aerospace and robotic applications over CAN bus, Ethernet, and other robust transports. It is created to address the challenge of deterministic on-board data exchange between systems and components of next-generation intelligent vehicles: manned and unmanned aircraft, spacecraft, robots, and cars" [45]. Main features are:

- Multimaster bus.
- Long distance, even at maximum rate.
- 1 Mbps is the standard rate.
- 3 pin needed (CAN HI, CAN LO, GND).

More detail at `https://forum.uavcan.org/t/uavcan-v1-crash-course/778`.

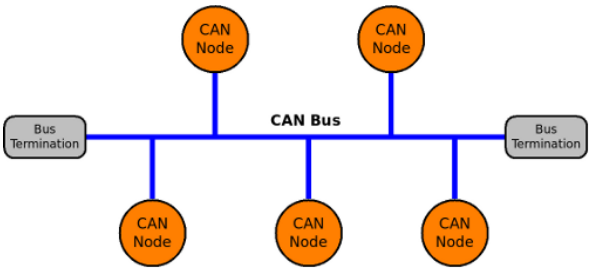

Figure 3.32: UAVCAN communication schematic. Image source: ArduPilot guide

PPM

Pulse-position modulation (PPM) is a form of signal modulation where M message bits are encoded by transmitting a single pulse in one of 2^M possible required time shifts. This is repeated every T seconds, such that the transmitted bit rate is M/T bits per second. It is primarily useful for optical communications systems, which tend to have little or no multipath interference [31].

SPI

The Serial Peripheral Interface (SPI) is a synchronous serial communication interface specification used for short-distance communication (10 cm max), primarily in embedded systems. Full duplex communication is used to connect SPI devices using a master-slave architecture with a single master and up to 4 slaves. Multiple slave-devices are supported through specific selection lines (SS: slave selection line). Moreover, a clock line SCLK is needed for synchronization purposes. The actual data exchange takes place over two different buses: MOSI (master output slave input - from master to slave) and MISO (master input slave output - from slave to master) [39]. SPI is much faster than I2C (20 MHz compared to 400 kHz), but it requires more wires and it has limitations on the maximum number of slaves. Figure 3.33 shows SPI schematic.

Figure 3.33: SPI communication schematic. Image source: ArduPilot guide

ArduPilot overview and comparison with PX4

4.1 ArduPilot main features

"ArduPilot is an open-source autopilot that enables the creation and use of trusted, autonomous, unmanned vehicle systems for the peaceful benefit of all" [3]. ArduPilot can be used to drive almost any vehicle and application. Being an open-source project, it is constantly improving relying on feedback from a large active community of users. Both the community and the commercial partners work with the development team adding functionalities and enhancing stability [3]. The three main features are:

- **Versatility**: although ArduPilot does not manufacture any hardware (conversly with respect to PX4), ArduPilot firmware works on a wide variety of different hardware to control unmanned vehicles of all types (PX4 boards included).
- **Real-time features**: coupled with ground control software, unmanned vehicles running ArduPilot can have advanced functionality including real-time communication with operators.
- **Strong community**: ArduPilot has a huge online community dedicated to helping users with questions, problems, and solutions [3].

4.2 ArduPilot architectural overview

The ArduPilot basic structure can be broken down into 5 main parts:

- Vehicle code
- Shared libraries
- Hardware abstraction layer (AP_HAL)
- Tools directories
- External support code (mavlink, dronekit)

4.2.1 Vehicle code

The vehicle directories are the top level directories that define the firmware for each vehicle type. Currently there are 4 vehicle types: Plane, Copter, APMrover2 and AntennaTracker. Although having different vehicle types, there are a lot of common elements between them. Our elaboration will be mainly about copter type [4], since this thesis is based on such a typology.
Along with the *.cpp files, each vehicle directory contains a make.inc file which lists library dependencies. The Makefiles read this to create the -I and -L flags for the build [4].

4.2.2 Libraries

The libraries are shared amongst the four vehicle types Copter, Plane, Rover and AntennaTracker. These libraries include sensor drivers, attitude and position estimation (aka EKF) and control code (i.e. PID controllers) [4].

4.2.3 Hardware abstraction layer

The AP_HAL layer (Hardware Abstraction Layer) is how ArduPilot is made portable to lots of different platforms. There is a top level AP_HAL in libraries/AP_HAL that defines the interface that the rest of the code has to specific board features, then there is a AP_HAL_XXX subdirectory for each board type, for example AP_HAL_AVR for AVR based boards, AP_HAL_PX4 for Pixhawk boards and AP_HAL_Linux for Linux based boards [4]. The goal is to abstract the low level details so that the top level can work irrespective of the used board.

4.2.4 Tools directories

The tools directories are miscellaneous support directories. For examples, tools/autotest provides the autotest infrastructure behind the `autotest.ardupilot.org` site and tools/Replay provides our log replay utility [4].

4.2.5 External support code

On some platforms we need external support code to provide additional features or board support [4]. Currently the external trees are:

- PX4 NuttX - the core NuttX RTOS used on Pixhawk boards
- PX4 Firmware - the base PX4 middleware and drivers used on Pixhawk boards
- UAVCAN - the uavcan CANBUS implementation used in ArduPilot
- MAVLink - the mavlink protocol and code generator

Figure 4.1 shows the ArduPilot software architecture. The GCS and the companion computer (if present) communicate with the drone by means of MAVLink. The abstraction layer abstracts hardware functionalities allowing the flight code to control peripherals by means of services. The RTOS suitable for our application would be

ChibiOS, since our board is a Pixhawk hardware, nevertheless, depending on the specific hardware, different solutions may be implemented.

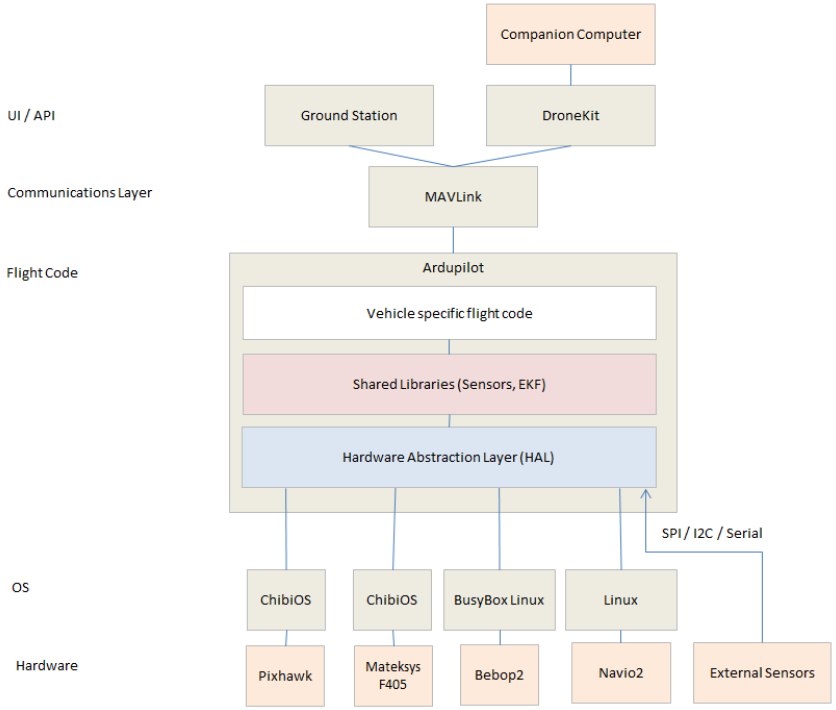

Figure 4.1: ArduPilot software architecture. Image source: ArduPilot website.

A more detailed schematic, containing all the involved scripts of the overall architecture is shown in figure 4.2. 'Background thread' groups together IMU, barometer and GPS measurements so that EKF can be used to correctly estimate current position and attitude. This part is very similar to the Sensor Hub seen in PX4. Estimated states are then used along with the flight mode for the navigation stack, the position control and the attitude control. Lastly, outputs are sent to the correct motors in order to apply the control strategy.

The overall structure is very similar to the one presented in figure 3.10.

Copter position and attitude controllers

Position controller is very similar to the one explained in PX4 overview, as noticeable in figure 4.3. Nevertheless, the proportional part used to transform the target position into a target velocity is implemented as a square root controller (instead of a linear one). The same consideration holds for the attitude controller.

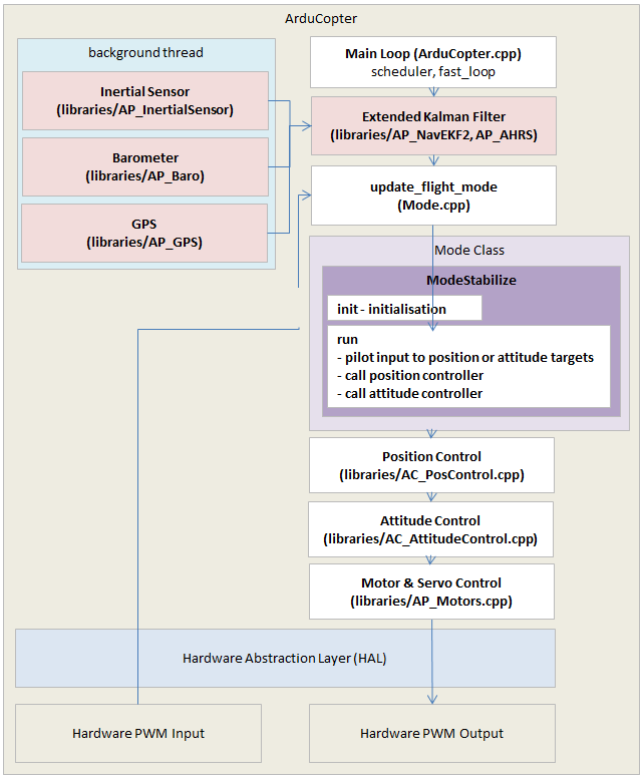

Figure 4.2: ArduPilot software architecture. Image source: ArduPilot website.

4.2.6 ChibiOS

ChibiOS is a complete development environment for embedded applications including RTOS, an HAL, peripheral drivers, support files and tools. ChibiOS is divided into different products, in particular 2 RTOS: RT and NIL [8].
RT is the high performance RTOS part of the ChibiOS embedded collection. It has been designed with the idea of creating a very feature-complete RTOS that could excel in performance and code size.
NIL has been created with the idea to bring RTOS functionalities to very small devices and yet to be a full-fledged RTOS.

RT is the version implemented by ArduPilot; its main features are:

- Extremely high performance RTOS.
- Fully static architecture.
- Very complete set of features.
- Strong debug support.
- Clean code base.
- Compatibility layer with CMSIS RTOS.
- Preemption.
- FIFO and Round Robin scheduler compatible.

Only 11 architectures are officially supported, 10 unofficially.

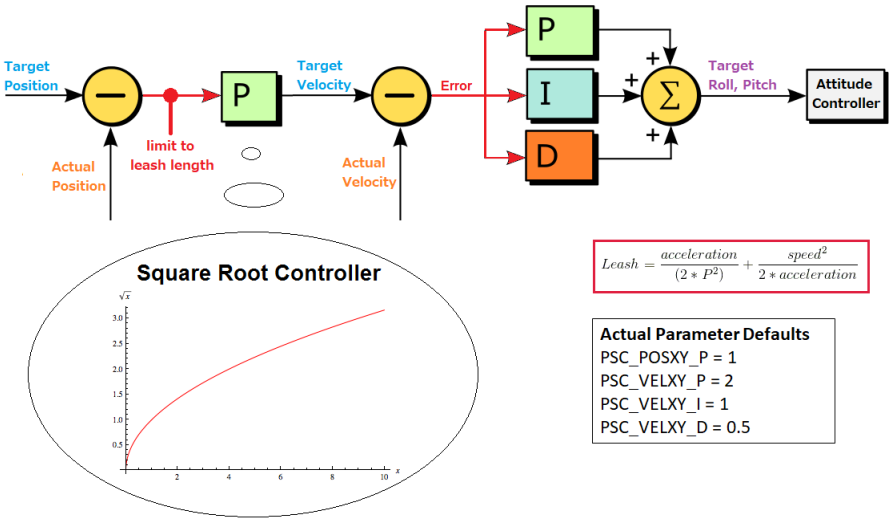

Figure 4.3: Copter position controller. Image source: ArduPilot website

4.2.7 ArduPilot threading

ArduPilot code is based on a setup() / loop() structure inherited from Arduino: the setup() function is executed only once whenever the system is booted, whereas the loop() part implements the actual functionalities and it is continuously executed. Although having such a structure similar to a single threaded system, in fact it is not. The threading approach strongly depends on the involved board. Some boards, like APM1 and APM2, do not support threads, therefore they are handled through timers and callbacks. On the contrary, some other boards such as PX4, supports a POSIX threading model with real-time features [6].

4.2.8 MAVLink

MAVLink is the communication protocol used in ArduPilot for the communication between the drone and the GCS. A more detailed overview can be found in section 3.5.

4.2.9 Mission Planner

Mission Planner is a ground control station for Plane, Copter and Rover, and it is the reference GCS for ArduPilot. Conversely to QgroundControl it is compatible with Windows only. Being Mission Planner a GCS, it can be used both as a configuration utility or as a dynamic control add-on for autonomous vehicles. A simple list of operations that can be performed using Mission Planner is shown below:

- Load the firmware on the flight controller board chosen to control the vehicle.
- Setup and tune vehicle parameters for optimal performance.
- Create plans for autonomous missions to be deployed on the vehicle.
- Download and analyse mission logs created by the flight controller.
- Perform HITL simulations.

- If using appropriate telemetry hardware it is possible to do the following operations:
  - Monitor the vehicle status during the operation.
  - Record telemetry logs containing much more details than the on-board autopilot logs.
  - Operate your vehicle in FPV (first person view)

4.2.10 Flight modes

Copter has 23 flight built-in flight modes, 10 of which are regularly used. Some modes support different levels/types of flight stabilization, some other a sophisticated autopilot and other smart features [5].
Flight modes can be controlled in different ways: using RC (through a transmitter switch), using mission commands, or commands from a ground station (GCS).

The following table summarizes all the ArduPilot flight modes [5].

Acro	Holds attitude, no self-level
Airmode	Actually not a mode, but a feature,see below
Alt Hold	Holds altitude and self-levels the roll and pitch
Auto	Executes pre-defined mission
AutoTune	Automated pitch and bank procedure to improve control loops
Brake	Brings copter to an immediate stop
Circle	Automatically circles a point in front of the vehicle
Drift	Like stabilize, but coordinates yaw with roll like a plane
Flip	Rises and completes an automated flip
FlowHold	Position control using Optical Flow
Follow	Follows another vehicle
Guided	Navigates to single points commanded by GCS
Heli_Autorotate	Used for emergencies in traditional helicopters. Helicopter only. Currently SI
Land	Reduces altitude to ground level, attempts to go straight down
Loiter	Holds altitude and position, uses GPS for movements
PosHold	Like loiter, but manual roll and pitch when sticks not centered
RTL	Retruns above takeoff location, may aslo include landing
Simple/Super Simple	An add-on to flight modes to use pilot's view instead of yaw orientation
SmartRTL	RTL, but traces path to get home
Sport	Alt-hold, but holds pitch and roll when sticks centered
Stabilize	Self-levels the roll and pitch axis
SysID	Special diagnostic/modeling mode
Throw	Holds position after a throwing takeoff
ZigZag	Useful for crop spraying

4.3 PX4 vs ArduPilot

A detailed comparison about the main aspects of the two firmware is now carried on.

4.3.1 Airframes

- PX4 firmware can be used to control 4 different vehicle frames/types: planes, multicopters, VTOL airframes and UGVs/rovers.
- Ardupilot firmware can be used to control 6 different vehicle frames/types: planes, copters, VTOL airframes, rovers, subs and antenna trackers.

As far as the subs is concerned, PX4 is currently working on the autopilot extension to this category, while antenna trackers are not intended to be adopted.
Although ArduPilot has more categories, it is worth to be said that within multicopter and VTOL airframes categories PX4 relies on a wider range of frames, in fact, considering multicopter category, PX4 has 18 frames, whereas Ardupilot only 7. Anyhow, new airframes can be added in both software.

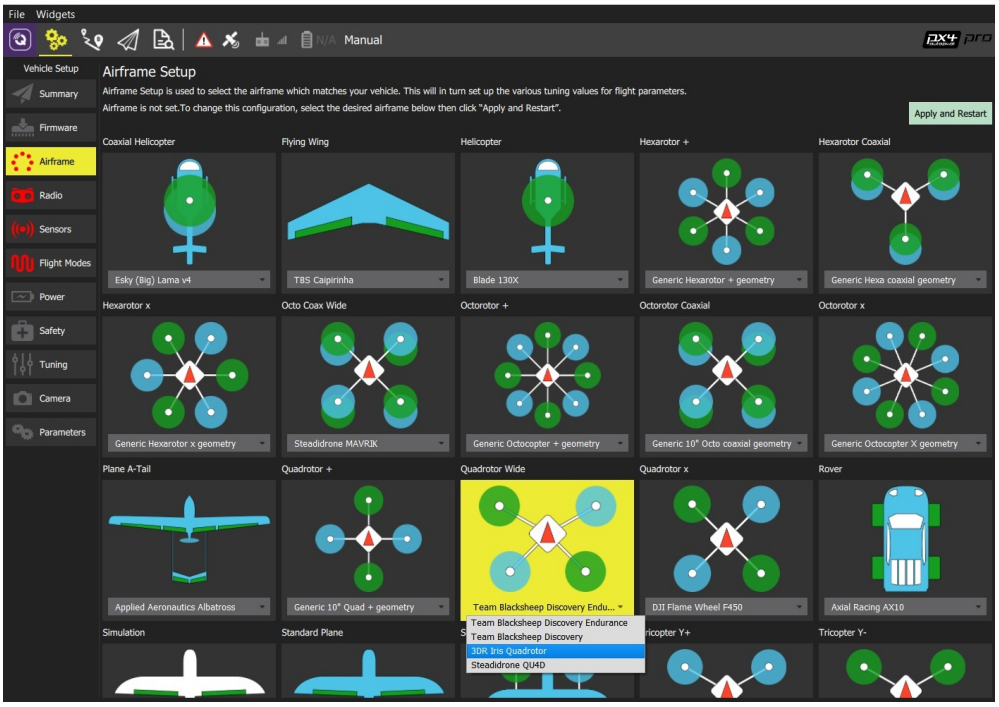

Figure 4.4: PX4 airframes available in QGroundControl.

More information about the supported frames can be found at the following links:
PX4: `https://dev.px4.io/v1.9.0/en/airframes/airframe_reference.html`
ArduPilot: `https://ardupilot.org/copter/docs/common-loading-firmware-onto-pixhawk.html`

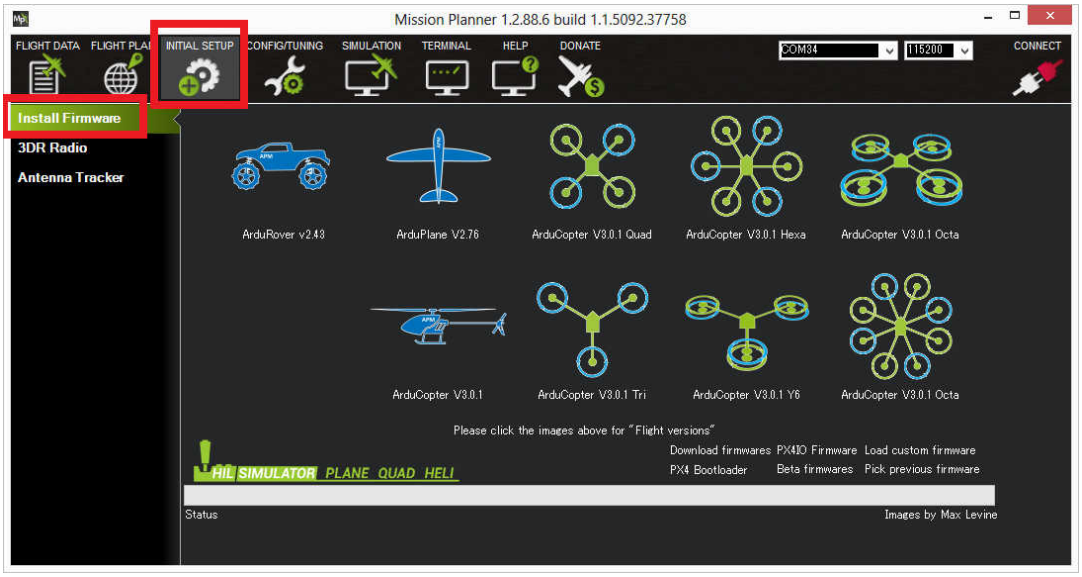

Figure 4.5: ArduPilot airframes available in Mission Planner.

4.3.2 Position and attitude controllers

Flight control stacks are very similar: IMU, GPS and other sensor's outputs are provided either directly to the navigation stack, or to the EKF module to estimate current position and attitude, and then to the navigation stack. Here position setpoints are created and sent to the position controller, which creates attitude setpoints to be sent to the attitude controller. Lastly, outputs are converted into suitable motor commands (PWM signals), in order to apply the correct control strategy.

Both the position and attitude controllers for the two autopilots rely on the same structure, which is a first P loop converting position/angle target into velocity/rate target, then a second PID loop creating the correct commands for the next block. Figures 3.2 and 4.3 show the two structures taken from the official websites.

Both ArduPilot and PX4 run at 400 Hz the position and attitude controllers if the same board is used. Obviously, using different boards with different processors, such a rate may change (no less than 100 Hz).

4.3.3 Control inputs

Both ArduPilot and PX4 use control inputs with the same values:

- Roll, pitch and yaw range: -1 to 1.
- Throttle range: 0 to 1.

Then, the conversion in PWM output values is performed differently:

- PX4 relies on mixer files. The final range is from 1000 to 2000.
- ArduPilot scales the control value to the range -4500 to 4500 by means of the hardware abstraction layer.

4.3.4 Flight modes

ArduPilot has 23 flight built-in flight modes, whereas PX4 only 13. Nevertheless, basic features such as automatic take off and landing, return to home etc. are provided in both cases. ArduPilot implements more difficult maneuver, such as the flip and ZigZag. In both firmware it is possible to add new flight modes easily.

More details can be found at the official webpage for both systems:
PX4: `https://docs.px4.io/v1.9.0/en/flight_modes/`
ArduPilot: `https://ardupilot.org/copter/docs/flight-modes.html`

4.3.5 RTOS

The RTOS used by ArduPilot is ChibiOS, whereas PX4 relies on NuttX. Both operating systems share the same key features, like preemption and Round Robin and FIFO schedulers. Nevertheless, NuttX supports a wider range of platforms, although some community members complained about NuttX not being fully real-time in the past, and a bit slower. It is worth to be said that ChibiOS is a relatively new OS, in fact before its introduction NuttX was used. Nowadays it is still possible to use NuttX with ArduPilot, although not recommended.

4.3.6 Ground Control Station

While PX4 is mainly used with QGroundControl, ArduPilot supports different ground control stations:

- Mission Planner: it is the most used GCS by ArduPilot users. Unfortunately, it is only Windows compatible but very user-friendly.
- QGroundControl: it is the same GCS seen in PX4, multi-platform (Windows, Mac OS X, Linux, Android and iOS), but slightly less intuitive.
- APM planner 2: it tries to merge the two good aspects of both QGroundControl and Mission Planner (multi-platform and user-friendliness), nevertheless it is not as known as the first two options.

The bottom line is that all GCS provide the same functionalities, what actually change is the GUI and the platform compatibility.

4.3.7 Simulation and testing

Both autopilots support and recommend Gazebo as simulator. Nevertheless, while PX4 performs both SITL and HITL on Gazebo, ArduPilot allows only SITL in Gazebo; HITL is not implemented yet for copters, only for planes.

4.3.8 License

License is one of the main differences between the two software; actually, considering the huge similarities, both in terms of hardware compatibility and software structure, it may be the final criterion to make a choice about the autopilot to be used.

- ArduPilot license is GPL, meaning that people modifying and then selling ArduPilot are obligated to make their modifications open.
- PX4 license is BSD, meaning that any company who forks the PX4 code can modify and sell products without making the modifications open.

4.3.9 Final comment

After a detailed comparison, I would like to make a short personal comment about the two communities that I developed during this period. It is worth to mention that ArduPilot community appears more active than PX4 one, since most of the questions from users are plenty of advises either from developers or from users themselves. Nonetheless, users seem to be working on easier projects, for instance it is rare to found questions about vision-based systems or swarm navigation. On the other side, PX4 community seems to be working on more complex projects, including any possible field aimed at improving autonomous vehicles. Unfortunately, it is rare to get answers from experts in a short period. However, PX4 represents the best autopilot for professional research-oriented projects due to its versatile structure suitable for many different systems.

Customization

5.1 Current built

The platform on which the customization will take place is the one developed by Simone Silvestro at PIC4SeR during the previous year [37]. The components are:

- **PixRacer autopilot**: small Flight Controller capable to run complete autopilot stack firmware like PX4 and run complex mission task with precise waypoints and fully autonomous navigation. Equipped with 32-bit STM32F427 microcontroller, MPU-9250 9-axis IMU, ST LIS3MDL magnetometer, MEAS MS5611 barometer, microSD slot, many standard protocols like I2C, SPI, UART, PPM, PWM and others, ESP8266 WiFi module included to exchange data with companion computer.
- **ESP8266**: tiny WiFi module to create an access point on the drone to exchange data with companion computer.
- **Tattu 800mAh 2S LiPo Battery**: 7.4V, 45C rated, the right compromise between weight and a total Time Of Flight around 10 min.
- **ACSP5 Power module**: Up to 42V voltage input (10S LiPo batteries), Up to 80A current sensing with built in INA1x9 High-Side Current Monitor, 17 mm x 17 mm package, JST-GH connector.
- **BLHeli 4-in-1 ESC**: Integrate BusyBee2 controller running Up to 48MHz, FD6288 MOSFET Drivers and Dual N-Channel 30V 12A MOSFET for phase legs. The 4-in-1 solution is common for space saving, without compromise performance in the case of a small build.
- **FrSky X4R Telemetry receiver**: 16 channel S-BUS and 3 channels PWM.
- **T-Motor MT-1306-10 3100KV**: tiny and efficient motors, only 11.2 g each, capable of a max thrust of 218 g each with 6020 props (872 g total), more than the double of the total mass of the drone, so enough for this application.
- **T-Motor 6020 carbon fiber propellers**: 2-blades lightweight propellers, better than 3-blades solutions when goal is not thrust but Time Of Flight.
- **ImmersionRC 5.8GHz Video Tx**: 600mW output power, SMA connector for external antenna, 7 bands.
- **Triumph 5.8GHz FPV Antenna Set**: TBS circular antenna, SMA connector.

- **mRo uGPS ublox SAM M8Q + Compass**: micro GPS, supports GPS and GLONASS constellations with built-in LIS3MDL compass, standard JST-GH connectors PixRacer compatible.
- **Caddx Turtle V2 HD Camera**: 1080p 60fps mp4 High Definition video recording, 155° Field Of View angle on the horizontal axis, microSD card slot up to 64GB, Input power accepted from 4.5 to 20 V, 20x20 cm standard mounting holes.

5.2 Servomotor integration

The first customization goal consists in adding a servomotor able to control and/or stabilize the camera pitch. Before diving into the firmware modification, a brief context is required. As deeply described in previous sections, the drone used for this part of the thesis is equipped with Pixracer flight controller, therefore only 6 PWM output pins are available. Unfortunately, 4 of them are needed for the motors, hence both the remaining 2 pins must be employed for the camera and the servomotor implementation. The chosen configuration is the basic one: the first 4 pins (1, 2, 3, 4) are used to drive the drone motors, the 5th pin is given to the servomotor, whereas the last one is used for the camera trigger.
The camera is intended to start recording in specific waypoints, therefore the following configuration should be adopted:

- **TRIG_PINS = 6**: specifying which pin is used for the camera.
- **TRIG_INTERFACE = 1**: setting GPIO mode for pin 6.
- **TRIG_POLARITY = 0**: set trigger polarity, 0 means Active low, 1 Active high.
- **TRIG_ACT_TIME = 1000**: set active trigger time at 1s (1000ms) since manufacturer of the Caddx HD camera declare 1 sec press on the button to start.
- **TRIG_MODE = 1**: this set the camera trigger mode to on command.

Unfortunately, due to Pixracer architecture, it is not possible to use pin 5 as PWM and pin 6 as GPIO, therefore the camera will start recording at the beginning of the flight mission (pressing a specific button) and will have to be stopped manually after landing. This limitation is discussed deeply in section 6.4.1.

More details about the camera options can be found at `http://docs.px4.io/master/en/advanced_config/parameter_reference.html#TRIG_POLARITY`.

As far as the servomotor is concerned, it must be tuned using the PWM output value on pin 5. The first idea was to directly influence that specific parameter publishing a message on the `actuator_outputs` topic, which contains PWM outputs of each port (value between 1000 and 2000, 900 in disarmed condition). Unfortunately, this message created conflict with the position and attitude controller running in background, making the drone unstable. A simple way to overcome this problem was to act indirectly on that parameter, publishing information into the `actuator_controls` topic, which is linked to the PWM values passing through `fmu` module and the mixer files. In particular, publishing a suitable value into the 6th component of the third control group (RC passthrough,

figure 5.1) it is possible to obtain the desired output on the 5th pin of the Pixracer board. Such a mapping mapping is hidden inside mixer files. Using the standard configuration (`quad_x.main.mix`), the relevant line is:

```
S: 3 5 10000 10000 0 -10000 10000
```

Meaning that the 6th element (mapped with number 5 since numbering starts from 0) of the control group number 3, will be scaled within the full range (from 1000 to 2000) with uniform slope, as shown in figure 5.2. An important consideration about the servomotor behaviour is the following one: any output value is related to a specific angle, not to a speed rate. This means that having 1600 as output value, the servomotor will remain in a position proportional to 1600, and it will not reach the end of the motion range.
A detailed overview about the conversion from an `actuator_controls` value to a real output was explained in section 3.8.4.

Control Group #3 (Manual Passthrough)

- 0: RC roll
- 1: RC pitch
- 2: RC yaw
- 3: RC throttle
- 4: RC mode switch (Passthrough of RC channel mapped by RC_MAP_FLAPS)
- 5: RC aux1 (Passthrough of RC channel mapped by RC_MAP_AUX1)
- 6: RC aux2 (Passthrough of RC channel mapped by RC_MAP_AUX2)
- 7: RC aux3 (Passthrough of RC channel mapped by RC_MAP_AUX3)

Figure 5.1: Manual passthrough Control Group. Image source: PX4 developer guide.

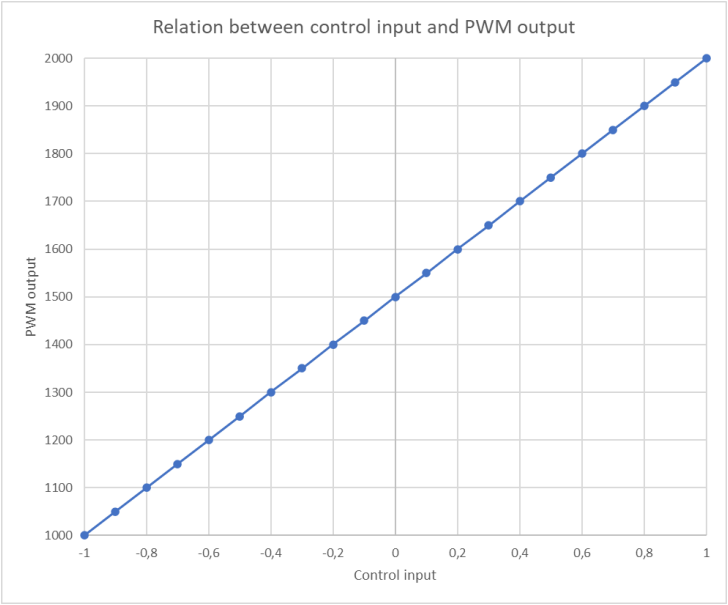

Figure 5.2: Relationship between control input and PWM output.

5.2.1 Controlling servo using RC

The first target to control the servomotor is to use it by means of the RC. The idea is very simple: a specific RC channel is associated to pin 5, so that manual passthrough is

automatically performed. The settings needed to work in this fashion are:

1. Connect the camera to pin 6 and the servomotor to pin 5.
2. Build the code and open QGC.
3. Use an airframe **without** built-in camera gimbal. Unfortunately, built-in camera airframes rely on the aux.mix mixers (real AUX ports) which are not consistent with our platform (Pixracer has only 6 PWM output and no AUX).
4. In QGC map AUX1 (pin 5) with the chosen channel (for instance 7) through `RC_MAP_AUX1` parameter.
5. Arm before using it (in principle only AUX2 (pin 6) requires being armed, AUX1 should work anyway).

With this settings camera pitch can be controlled directly using the RC selecting the desired channel. At any stick position corresponds a specific servomotor position. Holding the stick in a certain place will set the servomotor in a specific position.

A graphical representation of the involved topics and modules can be seen in figure 5.3. The RC stick position is published by `rc_update` module on `actuator_controls` topic, in particular on group 3, as it consists in manual passthrough.

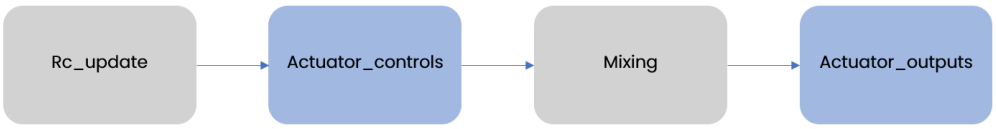

Figure 5.3: Modules and topics involved in RC passthrough chain.

5.2.2 Pitch stabilization

The second target is to stabilize the camera pitch based on the drone attitude. The idea is to create a module which reads attitude information, elaborates them and publishes a suitable control value on control group 3, creating a counter action with respect to the drone current pitch.

Attitude information can be sensed from `vehicle_attitude` topic, which contains the quaternion `q` representing the drone orientation. Such a quaternion must be converted into Euler angles in order to be both understood by developers and used by the code itself to create a counter action to the orientation variation. The conversion from quaternion to Euler angles is based on trigonometric considerations, and the mapping is shown in figure 5.4. Once converted the drone attitude, it is necessary to create the correct control input that allows to maintain constant the camera orientation. Such a control input is inversely proportional to the current drone pitch, therefore it is obtained inverting the pitch sign and rescaling it within the range -1 +1. Each control input between -1 and +1 will be converted through the mixer into an output value between 1000 and 2000, where any output represents a specific angular position.

The update rate is 250 Hz (4 ms), hence no significant variation can be assessed in such a small amount of time. A possible solution is to add a counter allowing to extend the time spanning between two consecutive corrections. The smaller the counter value, the faster will be the correction, nevertheless, servomotors needs some time to perform the desired move, consequently having a too small counter threshold is useless. On the other hand, the bigger the counter value, the slower will be the pitch correction, causing a delay between the required action and the actual one. Depending on the final application, the counter value must be chosen accordingly.

$$\begin{bmatrix} \phi \\ \theta \\ \psi \end{bmatrix} = \begin{bmatrix} \operatorname{atan2}(2(q_0 q_1 + q_2 q_3), 1 - 2(q_1^2 + q_2^2)) \\ \operatorname{asin}(2(q_0 q_2 - q_3 q_1)) \\ \operatorname{atan2}(2(q_0 q_3 + q_1 q_2), 1 - 2(q_2^2 + q_3^2)) \end{bmatrix}$$

Figure 5.4: Relationship between quaternion and Euler angles. Image source: Wikipedia

It is clear how the control strategy used to stabilize the camera pitch is a simple proportional controller. The default value is 40. This choice is a consequence of simulation measurements where, during standard missions, the maximum pitch assumed by the drone was about 33 degrees (absolute value); adding a reasonable safety factor around 7 degrees, 40 is obtained. It basically converts the drone pitch (value generally between -40 to 40 degrees) into a control input within the range -1 to 1. In case of drone pitch greater than 40 degrees (absolute value), the servomotor saturates.

Moreover, an additional feature is added: it is possible to set an initial offset acting as the reference around which the stabilization is performed. The procedure is very simple: module `servo_control` is started with a parameter indicating the offset (in degrees) using indicator `-p`. For instance, using the following command the stabilization will take place around 15 degrees.

```
servo_control start -p 15
```

Module `servo_control` publishes specific values on topic `actuator_control` so that the actual output is a counter action with respect to the drone pitch variation.

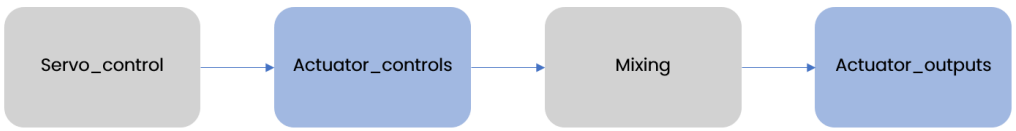

Figure 5.5: Modules and topics involved in `servo_control` chain.

The last needed step is to allow the simultaneous presence of both the RC passthrough and the stabilization effect, so that the operator can change the camera pitch having always stable images. Although being very simple in principle, in practice, the presence of two modules (`rc_update` and `servo_control`) publishing on the same topic (`actuator_control`) causes a conflict, resulting in a useless application. The proposed solution relies on a support topic: `rc_update` instead of publishing directly on `actuator_control`, publishes on a brand new topic, `actuator_control_rc`, that is read by `servo_control` and integrated with its control strategy adding the RC

contribution to the pitch variation one. In such a way, only one module is publishing on `actuator_control` and no conflict holds.
A simple schematic is shown in figure 5.6.

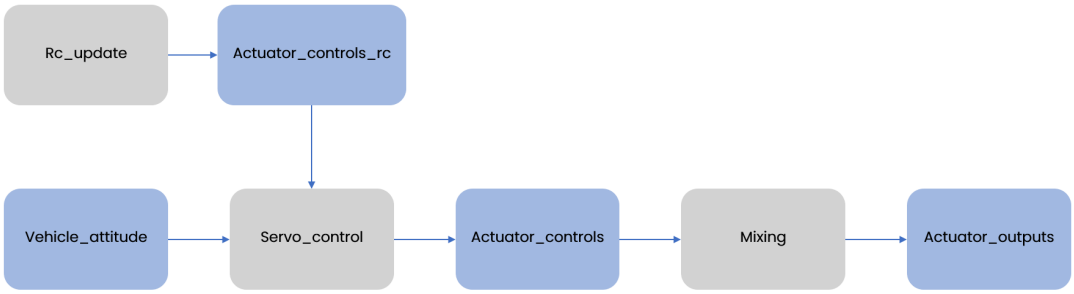

Figure 5.6: Topics involved in the final version of `servo_control` module.

Transformation from control value to pitch angle

To summarize, the control input is obtained scaling the drone pitch through a specific gain (40, but it can be changed). Such a control input can be a value between -1 and 1, and it is converted into a specific angle by means of the mixer file. The angle depends on the servomotor: in this case a cheap and light component is used, it has about 90° of motion, therefore -1 is mapped with 0° whereas 1 corresponds to 90°. Then, depending on how it is mounted, different orientations can be obtained.

Pixhawk

Dealing with Pixhawk boards things are much easier: the 6 AUX ports allow the usage of a 3-axis gimbal and PX4 is already structured to serve this purpose. The control group to be used is the second one (figure 5.7) and the mixer file used to convert the control input into a PWM output value is `mount.aux.mix`. The defaul conversion is the one discussed before (between 1000 and 2000 with unitary slope figure 5.2).

Control Group #2 (Gimbal)

- 0: gimbal roll
- 1: gimbal pitch
- 2: gimbal yaw
- 3: gimbal shutter
- 4: camera zoom
- 5: reserved
- 6: reserved
- 7: reserved (parachute, -1..1)

Figure 5.7: Control group 2: gimbal. Image source: PX4 developer guide.

Using the command `listener actuator_outputs` it is possible to see the output values on each channel. Dealing with Pixhawk boards, there are 6/8 channels dedicated to the motors and 6 channels to manage the mount. Figure 5.8 shows the mapping between each channel and the corresponding action.

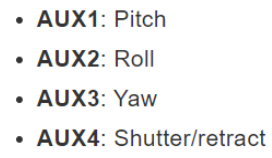

Figure 5.8: AUX port settings using Pixhawk boards.

Using Pixhawk boards a second module must be created because the publication topic is `actuator_controls_2`, not `actuator_controls_3`.

5.2.3 Servomotor position in specific waypoints

The third target is to set specific positions whenever the drone reaches specific waypoints. For doing this, a custom mission plan must be created. A simple approach consists in creating a sketch of the waypoints in QGC, saving the plan file, integrating additional features like camera trigger and servo input on the local .plan file, and finally loading it again on QGC.

The MAVLink command to be added to control the servo input is `MAV_CMD_DO_SET_SERVO` (number 183). Unfortunately, it is not publishing the value to the correct topic, in fact it publishes on `actuator_controls_2`, which can not be used with Pixracer flight controller as not equipped with AUX ports, therefore a firmware modification is needed.
The .cpp and .h file where `MAV_CMD_DO_SET_SERVO` is defined are `mission_block.cpp` and `mission_block.h`, placed in directory `Firmware/src/modules/navigator`.

The first operation to be done is to change the publication topic from `actuator_controls_2` to `actuator_controls_3`, hence into `mission_block.h` it is enough adding the last two lines shown below. We decided to add two lines instead of modifying them because this small change will only affect Pixracer boards; Pixhawk boards will use the previous publication topic.

```
1  uORB::Publication<actuator_controls_s>
2  _actuator_pub{ORB_ID(actuator_controls_2)};
3
4  uORB::Publication<actuator_controls_s>
5  _actuator_pub3{ORB_ID(actuator_controls_3)};
```

Then a modification of the original `MAV_CMD_DO_SET_SERVO` implementation is needed (`mission_block.cpp` file) because the original conversion did not work properly. According to MAVLink guide the parameters to be set for this command are 2: the pin number (between 0 and 7) and the PWM value (from 1000 to 2000). Such values are saved respectively into `item.param[0]` and `item.param[1]` variables. The conversion should create a value between -1 and 1 to be sent to `actuator_controls_3` so that the desired PWM output value is achieved on the desired pin. Nonetheless, the original formula was generating values between -1 and -0.5, which is only a part of the entire variation range. The new conversion is based on figure 5.2 and it allows to obtain any PWM value within the allowed range.

```
if (item.nav_cmd == NAV_CMD_DO_SET_SERVO) {
  PX4_INFO("DO_SET_SERVO command");

  actuator_controls_s actuators = {};
  actuators.timestamp = hrt_absolute_time();

  // params[0] actuator number to be set 0..5
  // (corresponds to AUX outputs 1..6)
  // params[1] new value for selected actuator in ms 900...2000

  // Original
  //actuators.control[(int)item.params[0]] =
  //1.0f / 2000 * -item.params[1];

  // New
  actuators.control[(int)item.params[0]] =
  1.0f / 500 * (item.params[1]-1500);

  _actuator_pub3.publish(actuators);
```

Since any output value is associated with a specific servomotor position, it is enough to specify the pin we want to set followed by the desired output value. For instance, needing a PWM value of 1800 on pin 5, just launch the command `MAV_CMD_DO_SET_SERVO` with parameters 5 and 1800, as shown in figure 5.9.

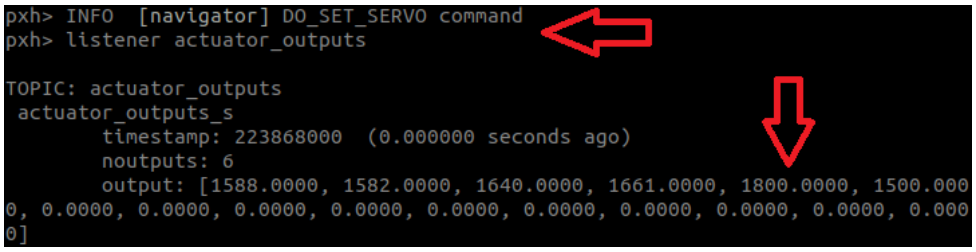

Figure 5.9: Simulation results. A simple plan was created: three waypoints, when reached the second one, the command DO_SET_SERVO is launched setting 1800 as output.

Dealing with Pixhawk boards things are easier because the standard `MAV_CMD_DO_SET_SERVO` configuration shall be used.

5.3 UWB sensor integration

The second main goal of this thesis is to integrate a UWB sensor to the current built. The standard approach (mostly used in literature and suggested by the developer guide) is to add it by means of a companion PC (like Raspberry Pi). Nevertheless, our goal is to try to directly connect the module to the Pixracer/Pixhawk board creating a custom new driver. The main difference between the two approaches is that using a custom driver implies interfacing the flight board directly to the module, whereas leaning on a companion PC means abstracting the sensor, since the flight controller will only receive messages containing data. Both scenarios have their own advantages: a custom driver reduces the weight of the drone and the number of involved components, causing a higher reliability of the entire system, decreasing the failure possibilities. On the other side, a companion PC allows the integration of more complex features, such as vision-based systems, and provides better re-usability due to the intrinsic abstraction. Nonetheless, using a companion PC only to read UWB measures is a wrong design choice as it

introduces a complex and heavy system to perform an easy task that could be managed directly introducing a custom driver.
Before starting with the technical details, a brief introduction about ultra wideband sensors is needed.

5.3.1 UWB technology at a glance

"Ultra wideband is a wireless technology intended for digital data transmission over short distances at low power density" [47]. The approach behind such a technology is clearly noticeable in figure 5.10: instead of using a narrow frequency range at a high power density (like most of the standard technologies), the UWB relies on a wide frequency range (3.1 GHz to 10.6 GHz) used at a low power spectral density.

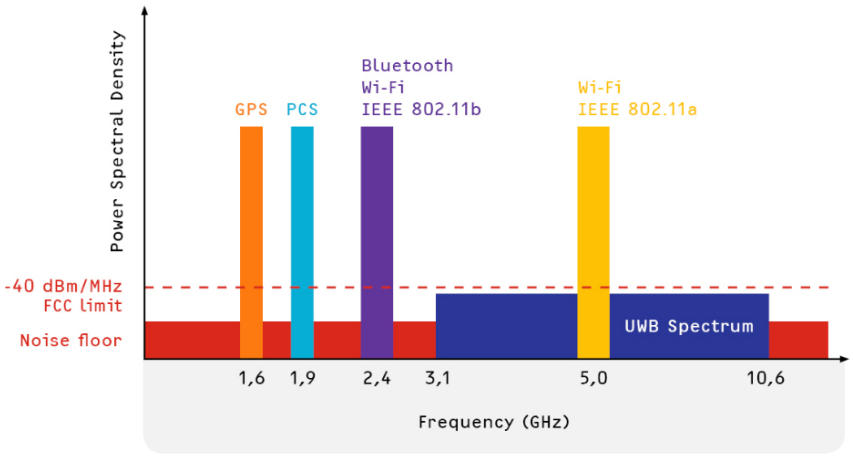

Figure 5.10: UWB technology compared to the standard ones. Image source: ELIKO

The field where UWB technology plays a key role is indoor positioning, since it allows to reach a localization precision down to 10 cm. Its main features against the competitors are:

- Obstacle penetration: standard technologies such as GPS (that relies on longer wave length) cannot penetrate obstacles, therefore they do not allow non-line-of-sight missions. UWB due to the wide frequency range enables such missions with relatively high precision and reliability.
- Reduced noise: having a low power spectral density it is a noise-free oriented technology. Moreover, its low power spectral density allows to coexist with other signals causing no interference.
- High precision ranging and reliability: its working principle is based on TOF (time of flight) measurements. Short pulses are sent by modules, then measuring the time difference between the sent and received instants it is possible to estimate distance between sender and receiver very precisely.

Working with UWB sensors is quite simple: given at least 3 fixed anchors the position of a tag (mounted on the device to be located) is estimated measuring the time of flight between each anchor and the tag. Using more anchors means introducing redundancy causing higher precision (true up to a maximum number of redundant elements). For this

project Decawave UWB sensors will be used, in particular the model DWM1001-DEV, since it allows to be programmed on the fly either as a tag or an anchor, other than having multiple interfaces to share data.

Figure 5.11: DWM1001-DEV picture. Image source: Decawave website

5.3.2 System configuration

As previously mentioned, modules can be programmed either as tag or as anchors. Depending on the working scenario, different settings could be used, nonetheless, some key points can be stated:

- At least one active tag is necessary. Tags represent devices to be tracked based on specific references.
- At least one active anchor is needed. Anchors represent reference points to define the tag position. Between all the anchors, at least one must be set as initiator.

The simplest scenario is when one tag and one anchor are present; in such a condition, the only parameter to be tracked is the relative distance between the two components. In general, the active tag should be connected to the drone, as it is the device to be tracked in space. The connection is performed with just 3 cables (Tx, Rx and GND) plus power cables:

- (DMW1001) Tx to Tx (Pixracer/Pixhawk).
- (DMW1001) Rx to Rx (Pixracer/Pixhawk).
- (DMW1001) GND to GND (Pixracer/Pixhawk).

Power side, DWM1001 requires 3.3V, not 5V and J13 do not need to be desoldered. The module configuration is performed by means of the DRTLS manager app (Android) from Decawave (downloadable from this link). The procedure is stated at pages 15 and 16 of the Firmware user guide downloadable from the same link as before.

The connection is performed using one of the serial ports of the Pixracer/Pixhawk boards according to figure 5.13. Port choice is relevant because it must be specified within the code in order to open the correct serial communication.

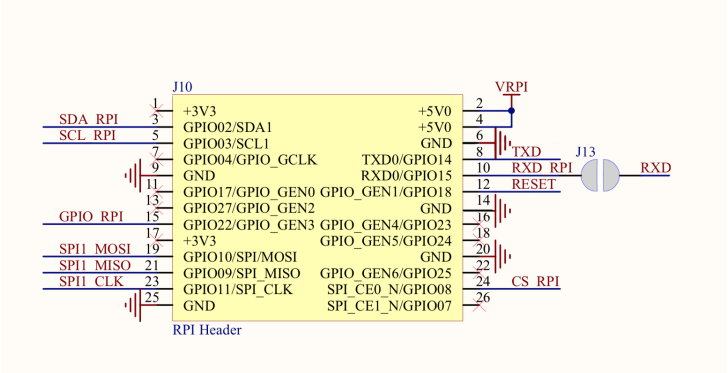

Figure 5.12: DWM1001-dev pin layout from Decawave datasheet.

```
# UART mapping on OMNIBUSF4SD:
#
# USART1                /dev/ttyS0              SerialRX
# USART4                /dev/ttyS1              TELEM1
# USART6                /dev/ttyS2              GPS
#
# UART mapping on FMUv2/3/4:
#
# UART1                 /dev/ttyS0              IO debug (except v4, there ttyS0 is the wifi)
# USART2                /dev/ttyS1              TELEM1 (flow control)
# USART3                /dev/ttyS2              TELEM2 (flow control)
# UART4
# UART7                                                 CONSOLE
# UART8                                                 SERIAL4
#
#
# UART mapping on FMUv5:
#
# UART1                 /dev/ttyS0              GPS
# USART2                /dev/ttyS1              TELEM1 (flow control)
# USART3                /dev/ttyS2              TELEM2 (flow control)
# UART4                 /dev/ttyS3              TELEM4
# USART6                /dev/ttyS4              TELEM3 (flow control)
# UART7                 /dev/ttyS5              ?
# UART8                 /dev/ttyS6              CONSOLE
```

Figure 5.13: Pixracer/Pixhawk serial port breakdown. Image source: PX4 forum.

5.3.3 Driver breakdown

Drivers in PX4 are implemented as modules. Modules can be written according to 2 different principles:

1. *Tasks* having their own stack and priority process.

2. *Work queue tasks* running on work queue thread and sharing the priority and the stack with other modules on the work queue.

Using work queue tasks allows to save a bit of RAM performing also fewer task switches. Though, these tasks are not allowed to sleep or poll on messages, which may be a limitation. Based on the current firmware state of the art, most of the standard drivers are designed to be work queue tasks, nevertheless, the GPS (one of the main drivers) is written as a stand alone task. Being this application essentially developed for indoor positioning, namely comparable to the GPS relevance, the driver is developed as a simple task.

Driver functions

The driver execution is very simple: whenever started with command `dwm1001 start`, the main function is launched: a task named `dwm1001`, with default priority and stack equal to 3000 is created. Such a task points at the function `dwm1001_thread_main` implementing the actual module as a basic task, namely composed by a first initialization part and then an endless loop collecting data and publishing them on a specific topic.

During the initialization part three functions are called for setting the DWM1001 module up:

1. `uart_init`: responsible for opening the serial communication with the correct port (specified within the header file, for instance in case of TELEM2 it is /dev/ttyS2).
2. `set_uart_baudrate`: responsible for the configuration of the serial communication parameters: 115200-8N1, namely 115200 as baudrate, start bit, 8 bit data length, no parity bit, 1 stop bit.
3. `dwm1001_programming`: responsible for programming the DWM1001 module so that it will send back the correct information to the drone.

Then, during the endless loop, the read information are parsed, anchor distances and tag position are isolated and published on a new topic named `dwm1001.h` according to the custom message definition `dwm1001.msg`.

uart_init

It is basically aimed at opening the serial port with the correct interface (TELEM2). The standard `open` function is used and the parameter to be set are:

- `O_RDWR`: to open the the communication in read and write mode.
- `O_NOCTTY`: needed to be portable.
- `O_SYNC`: to not cause problems with the writing operations during the module programming phase.

```
1     int serial_fd = open(uart_name, O_RDWR | O_NOCTTY | O_SYNC);
```

set_uart_baudrate

It follows the standard C configurations for setting the communication up (115200-8N1). The only relevant command to be added is `tcflush` in order to flush both the received data but not read and the written data but not transmitted (using parameter `TCIOFLUSH`).

dwm1001_programming

This is the non-standard component of the code since it needs to directly interact with the module, programming it to respond with specific commands.
The first operation to be performed is the module reset, because the initial condition is not known, although required. According to the module datasheet, the command `reset\r` is

needed. The main issue with writing operation is that it requires some time to actually write the desired bits, this is why the function `tcdrain` must be involved. Nevertheless, the module itself needs some time to digest the incoming reset command, therefore an escamotage should be used to let some time pass without freezing the application (so no `px4_sleep` command can be used, otherwise the application would be moved to pending state). This is why the absolute time is called and a fixed amount of time is added to the absolute time, so that a loop can continuously run (and so all the other real time parallel processes) until the next absolute time is the one given by the previous, plus the threshold; at that time the code can continue.

```
num_bytes = ::write(_fd, "reset\r", 6); // \r is working, it counts as 1 char
tcdrain(_fd);


hrt_abstime time_now = hrt_absolute_time();
const hrt_abstime timeout_usec = time_now + 2000000; // us
while (time_now < timeout_usec) {
 time_now = hrt_absolute_time();
}


if(num_bytes != 6) {
  PX4_ERR("Something went wrong with writing reset!");
  return false;
}
```

After the reset command, two consecutive `enter` commands are passed to land into the programming mode, according to the DWM1001 datasheet. Before continuing with next instructions some time must be waited.

```
num_bytes = ::write(_fd, "\r\r", 2); // \r is working
tcdrain(_fd);


time_now = hrt_absolute_time();
const hrt_abstime timeout_usec_2 = time_now + 2000000; // us
while (time_now < timeout_usec_2) {
  //PX4_INFO("time_now: %llu\ntimeout_usec: %llu", time_now, timeout_usec);
  time_now = hrt_absolute_time();
}


if(num_bytes != 2) {
  PX4_ERR("Something went wrong with writing enter twice!");
  return false;
}
```

Once the board is in the programming mode, it is possible to write the ultimate command used to extract data from the module. In this case, the command is `lec`.

```
num_bytes = ::write(_fd, "lec\r", 4); // \r is working
tcdrain(_fd);

if(num_bytes != 4) {
  PX4_ERR("Something went wrong with writing lec!");
  return false;
}

PX4_INFO("The dwm1001 module should be programmed with lec");
```

The `lec` command is going to return a message containing 3 relevant information:

- **The number of anchors** being detected and used to estimate the position of the tag.

- **The position of the i-th anchor** with respect to the coordinate system's origin (this is actually defined by the environment designer using the DWM1001 app) plus the distance between the tag and the i-th anchor (measured).
- **The tag position estimation plus the quality level of the estimation**. This part of the message is present only if there is a sufficient number of anchors.

An example of message could be:

`DIST,4,AN0,1151,5.00,8.00,2.25,6.44,AN1,0CA8,0.00,8.00,2.25,6.50,AN2,111C,5.00,0.00,2.25,3.24,AN3,1150,0.00,0.00,2.25,3.19,POS,2.55,2.01,1.71,98\r\n`

Where:

- `DIST`: indicator for the first part of the message containing the location of the anchors in the reference system. These coordinates are defined by the environment designer and set by means of the Android app.
- `4`: number of anchors detected.
- `AN0,1151`: number of the anchor + anchor's ID.
- `5.00,8.00,2.25`: anchor position coordinates into the reference frame defined a priori.
- `6.44`: distance between the i-th anchor and the tag (measured).
- `POS`: indicator for the second part of the message containing the estimated position of the tag into the reference system.
- `2.55,2.01,1.71`: X, Y and Z estimated coordinates of the tag into the reference system.
- `98`: quality level of the estimation.
- `\r\n`: end of the message.

Driver's main function

The main challenges to be managed by this driver were two:

1. The capability of reading a serial message with dynamic length.
2. The capability of adapting to the presence of different number of anchors.

To handle the first requirement, the design choice was to read serial characters 1 by 1, store them in a specific container, and parse the entire message in a second instance. In particular, the driver will spot `DIST` over the serial bus and it will start storing all components into a char array until the end of the message is detected (`\r\n`). Once the message is stored in this array, it will be split into its relevant components in order to be published on the topic of interest.

Managing the second requirement was a bit more difficult due to the PX4 architecture. In an ideal condition it would be enough to read the number of anchors being detected, and

to publish a topic containing a float array with a length proportional to the number of anchors (in fact the number of distances is proportional to such a number). However, PX4 requires topic messages to be with a fixed length. As a consequence, there was the need of constraining the maximum number of anchors so that the message will be published always with a fixed length equal to the established one. Before a cycle starts, the driver will set to 0 the entire array and it will fill it only with the detected components (that is proportional to the number of anchors read); the remaining ones will be set to 0.
The application subscriber will read the at least two values: the number of anchors and the float containing the distances. Knowing the number of anchors, it will read the array containing the distances only until the correct value, discarding the remaining ones.
If the `POS` indicator is detected within the message, a boolean variable will be set to true and the estimated position will be delivered in the topic, otherwise the published array will be made of zeros. Similarly to what described before, the user application will read both the array and the boolean variable deciding what to do based on the values of these variables.

Testing and troubleshooting

6.1 Introduction about testing

Testing is generally one of the biggest challenges in engineering projects, as it provides a real feedback about the work done. Furthermore, it is a great opportunity to stress the entire system or the single units understanding their behavior in non-standard operating conditions. According to the Verification-Validation model (V model, figure 6.1), there are different layers of testing in software engineering, each of them corresponding to a specific development stage. The layer carried on this thesis is mainly the first one: the unit testing, where the integrated unit is tested with the board alone, so that other components cannot create interference.

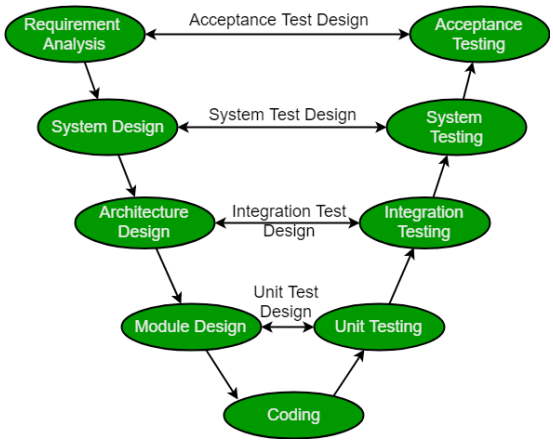

Figure 6.1: Graphical representation of the Verification-Validation model. Image source: Geeksforgeeks.

6.2 Servo testing

Testing the servo means connecting it to the Pixracer, moving the board simulating a pitch variation, and then watching the servo rotation that should be opposite to the pitch variation. However, this is only the final testing stage, because there are many different steps to be checked before being able to say that the test was successful, even though the motion was "correct". As described in the previous chapters, the module should read the drone pitch from `vehicle_attitude`, create a counteraction to be published in `actuator_controls` so that the mixer file will convert it into `actuator_outputs`. The first testing level consists in understanding whether each topic was published in the correct manner or not; if it wasn't, for sure the test did not succeed. For doing this, the listener command is very useful because it allows to list the last published message on a topic. Within the created module (`servo_control`), 4 topics are involved: `input_rc`, `vehicle_attitude`, `actuator_controls` and `actuator_outputs`. As far as the the first two is concerned, they are only used in subscription mode, therefore if any problem occurs within the read message, it is not caused by the new module, but by the publisher of such a topic. A different scenario happens if the read message is correct, but the module is not behaving as expected, which means `actuaor_controls` is not publishing the right values. In this scenario there could be either a problem into the subscription/publication mechanism or into the module logic. Lastly, if `actuator_controls` is published correctly, but `actuator_outputs` is not responding accordingly, it probably means there is a problem with the mixer file or the airframe configuration.
Once topics are correctly published according to the different scenarios, it is possible to continue with the real implementation, connecting the Pixracer board with the servomotor. The expected result is that a positive drone pitch position corresponds to a negative servo position and vice versa. Moreover, setting the radio controller to a specific channel and moving the knob, the position of the servo should change accordingly (while maintaining active the stabilization).

6.2.1 Test description

Goal and expected result

The goal of this test is to check the servomotor behavior with respect to the pitch variation of the Pixracer. The expected result is a servomotor movement opposite to the pitch variation of the flight controller. Moreover, setting AUX1 passthrough to channel 7 and moving the knob it is possible to see a variation of the stabilization point.

Needed gear

- Pixracer.
- Servomotor.
- 5V source (Arduino in my case, but it could be whatever source able to provide 5V).
- Radio controller.

Setup

The first step is the connection between the servo and the Pixracer using the 3 equipped cables. According to my setup:

- Orange cable: 5V.
- Yellow cable: signal carrying the position.
- Brown cable: ground.

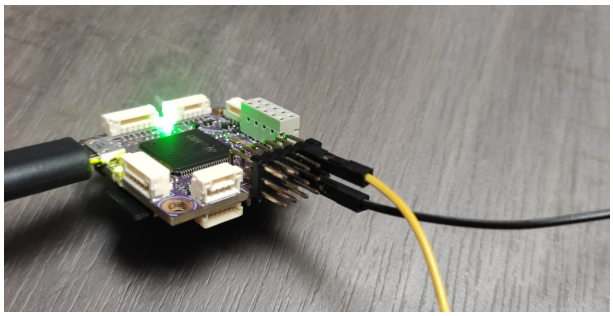

Figure 6.2: Connections between Pixracer and servomotor.

The orange cable should be connected to the 5V source. In my case such a voltage is provided by the Arduino board, so it is directly connected to the 5V pin available on Arduino. The brown cable must be connected with both the source GND and the Pixracer GND, so that the three devices (flight controller, servomotor and source) will have the same reference (this is a key point, if missed the test won't work). Lastly, the yellow cable must be connected with the Pixracer signal pin, that is the top one (according to the figure). Once the connection between the servo and the Pixracer is performed, it is possible to plug the flight controller to the PC with the micro USB cable and to open QGC. Running the command `<< make px4_fmu-v4_default upload >>` the firmware will be loaded into Pixracer and the module `servo_control` will automatically start. As a consequence, after a few seconds it will be possible to manually change the Pixracer pitch observing the variation of the servo position.

The next two figures are an useful representation of the communication layout.

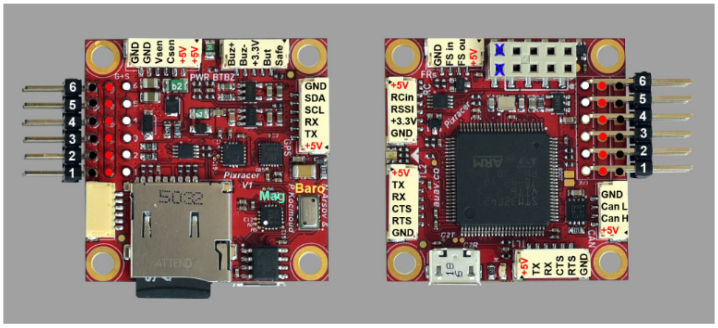

Figure 6.3: Pixracer interfaces. The 6 pins on the right represent the 6 PWM outputs; white is the signal, red is the 5V (present only if there is a battery connected to Pixracer) and black is ground.

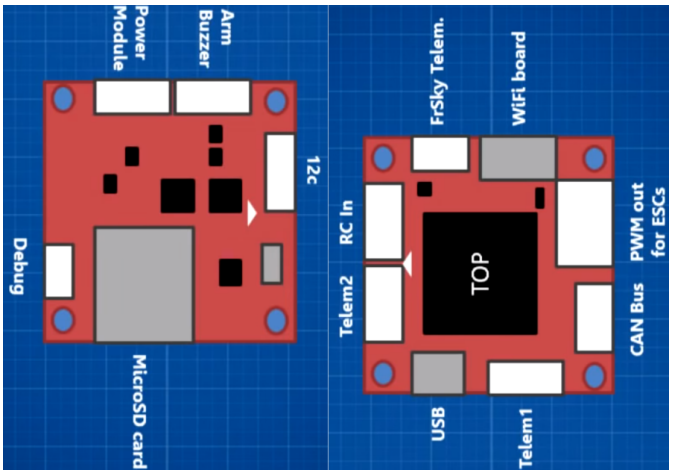

Figure 6.4: Pixracer port schematic.

Lastly, for testing the RC contribution, go on Radio setup in QGroundControl and set channel 7 as AUX1 passthrough (figure 6.5). Switch the RC on and after a while start moving the associated knob; you should start seeing the servo changing its stationary point.

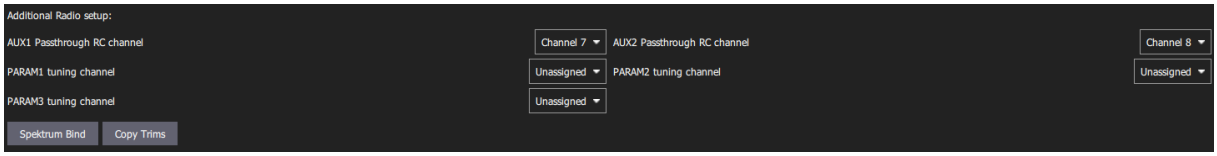

Figure 6.5: QGroundControl AUX1 configuration for RC passthrough.

6.3 Driver testing

Testing phase was highly affected by the COVID-19 pandemic, which strongly constrained the access to the research lab for master thesis students. Instead of giving up, postponing such a challenging and important part of the work, I decided to find an alternative way to carry the testing part. Being this driver based on a serial communication, I programmed an Arduino MEGA board to publish specific messages over one of its serial ports so that the autopilot could be connected to this port, simulating a connection to a DWM1001 module. The actual features to be tested were mainly 2: the capability of adapting to different number of anchors without losing consistency, and the capability of recognizing the presence of the estimated tag position within the payload.

Before delving into the test details, it is worth to mention the actual message sent over the topic dwm1001.

```
uint64 timestamp          # time since system start (microseconds)
float32[98] distances     # Distances between anchors and tag, each
                            anchor provides 4 measurements (x,y,z
                            coordinate of the anchor with respect
                            between the anchor and the tag).
float32[4] positions      # Position of the TAG into the reference
                            system defined by the anchors (x,y,z,Quality)
uint16 anchor_num         # Number of anchors
bool pos_detected         # Is POS present into the payload?
```

Arduino was programmed to send the following messages, each 4 second spaced:

1. `DIST,1,AN0,1151,5.00,8.00,2.25,6.44\r\n`

   Here the driver should set `anchor_num` equal to 1 and `distances` should have the first 4 values different from 0, whereas the remaining ones equal to 0. POS is not detected, therefore `pos_detected` should be false and `positions` set to 0.

2. `DIST,4,AN0,1151,5.00,8.00,2.25,6.44,AN1,0CA8,0.00,8.00,2.25,6.50,AN2,111C,5.00,0.00,2.25,3.24,AN3,1150,0.00,0.00,2.25,3.19,POS,200.55,2.01,100.24,100\r\n`

   Here the driver should set `anchor_num` equal to 4 and `distances` should have the first 16 values different from 0, whereas the remaining ones equal to 0. POS is detected, therefore `pos_detected` should be true and `positions` set to components read into the message.

3. `DIST,4,AN0,1151,5.00,8.00,2.25,6.44,AN1,0CA8,0.00,8.00,2.25,6.50,AN2,111C,5.00,0.00,2.25,3.24,AN3,1150,0.00,0.00,2.25,3.19\r\n`

   Here the driver should set `anchor_num` equal to 4 and `distances` should have the first 16 values different from 0, whereas the remaining ones equal to 0. POS is not detected, therefore `pos_detected` should be false and `positions` set to 0. This test was made to check the ability of adapting to POS variations. For whatever reason, modules could stop estimating the position and the drone should be able to detect such a condition.

4. `DIST,3,AN0,1151,5.00,8.00,2.25,6.44,AN1,0CA8,0.00,8.00,2.25,6.50,AN2,111C,5.00,0.00,2.25,3.24,POS,300.55,1.01,6.24,100\r\n`

   Here the driver should set `anchor_num` equal to 3 and `distances` should have the first 12 values different from 0, whereas the remaining ones equal to 0. POS is detected, therefore `pos_detected` should be true and `positions` set to read components. This test was made to check the ability of adapting to number of anchor variations; in a real application it could happen that an anchor stops working or just go outside of coverage, the driver should detect this variation and clear the previous values.

5. `DIST,3,AN0,1151,15.00,800.25,2.25,6.44,AN1,0CA8,100.00,8.00,2.25,600.50,AN2,111C,5.00,0.00,2.25,3.24\r\n`

   Here the driver should set `anchor_num` equal to 3 and `distances` should have the first 12 values different from 0, whereas the remaining ones equal to 0. POS is not detected, therefore `pos_detected` should be false and `positions` set to 0. This test was made to check the ability of adapting to variable distances (more than 100 meters, even though it is very unlikely that an anchor has such a high coverage).

Tests highlighted the driver capability of adapting to different number of anchors, the presence of POS and the variability of the detected distance.
The used setup is shown in figure 6.6. It is worth to say that contrarily to the connection between Pixracer and DWM1001, the serial link between Pixracer and Arduino was made in a standard approach: Arduino TX connected to Pixracer RX, Arduino Rx connected to Pixracer Tx.

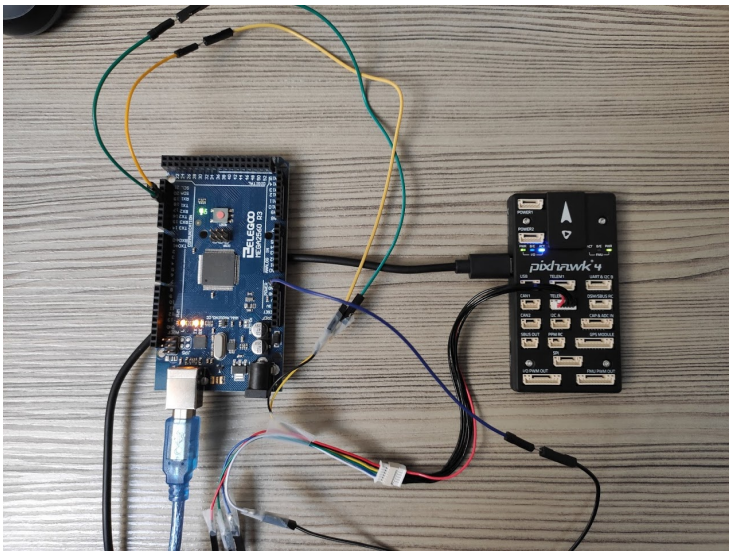

Figure 6.6: Driver testing setup using Arduino.

6.3.1 DWM1001 testing description

Goal and expected results

The goal of this test is to check the driver functionalities using two real DWM1001-dev modules. The expected result is that once the driver is started, the active tag will be programmed to send the relative distance from the anchor publishing this value to the topic `dwm1001`.

Needed gear

- 2 DWM1001-dev modules.
- Android phone or tablet (not too old) with Decawave app installed.
- Pixracer/Pixhawk board.
- PC with my version of the PX4 Firmware (https://github.com/francimala/Firmware) and QGC.
- 2 micro USB cables.

Setup

The first operation is setting up the 2 DWM1001-dev modules using the Decawave Android App. If you haven't installed it yet, you can download it from this link: `https://www.decawave.com/source-code-for-the-android-application/`. I highly recommend using an up-to-date smartphone because I tried on a 2015 Huawei model and I was not able to complete the setup, although having full compatibility. One module should be set as an active tag (left picture), whereas the other one should be set as an active anchor (initiator, right picture). The anchor should be plugged to a fixed position (like the main network) whereas the tag to the flight controller, supplied by whatever source, either a PC through micro USB cable, or a power bank or a battery (in this case pay attention to the voltage, it must be 3.3V, not 5V!!). My suggestion is to connect it

to a PC through micro USB cable so that PuTTy can be used to check what is going on over the serial port.
Important note: if you have any problem with the DRTLS app (you don't see a tag or an anchor) you should try to flash again the firmware on the DWM1001 modules. It happens that if you use a different smartphone from one test to another, it somehow remember the old configuration and the new smartphone will not detect the modules in the correct way. If you need to flash again the firmware, follow page 15 and 16 of the guide that you can find here: `https://www.decawave.com/wp-content/uploads/2019/01/DWM1001-Firmware-User-Guide-2.1.pdf`.

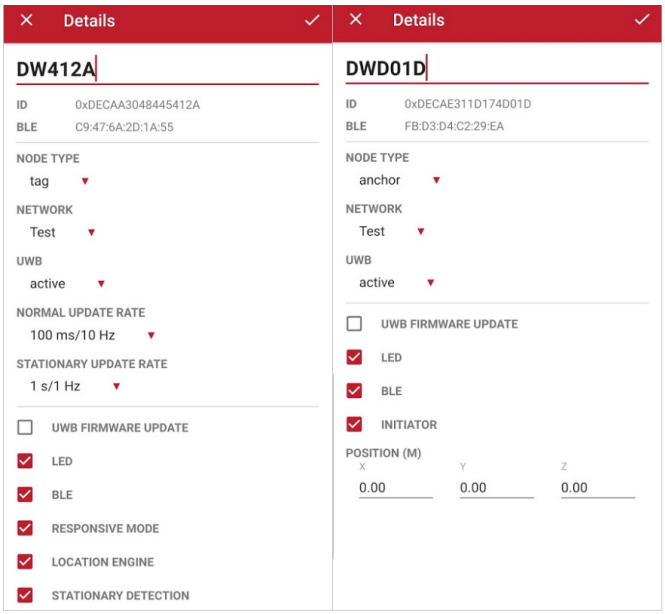

Figure 6.7: Decawave app configuration needed to run the test.

Connect flight controller to the DWM1001-dev module using the TELEM2 port of the Pixracer/Pixhawk (within the code the default port I used is /dev/ttyS2, but you can change it if you need it). Figures 6.4, 6.3 and 5.13 may help you finding the right port on the board, especially Pixracer, because Pixhawk is pretty clear.

The connection should be the following one:

- Black cable is the Pixracer/Pixhawk ground and must be connected to DWM1001-dev GND (third pin starting from above).
- Yellow cable is Pixracer/Pixhawk TX and must be connected to DWM1001-dev TX (fourth pin).
- Green cable is Pixracer/Pixhawk RX and must be connected to DWM1001-dev RX (fifth pin).

Now it is possible to start powering devices up.

1. Plug the anchor to the main net.
2. Plug the tag to the PC.

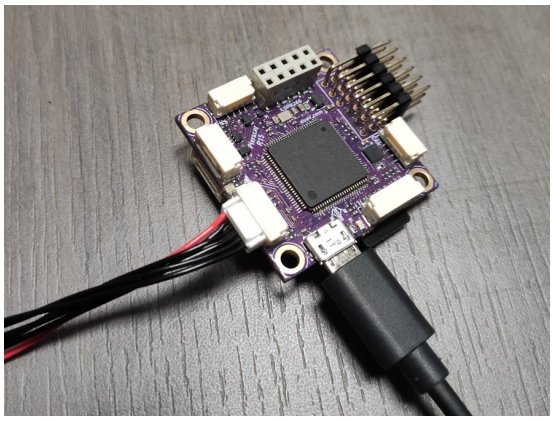

Figure 6.8: TELEM2 focus on Pixracer.

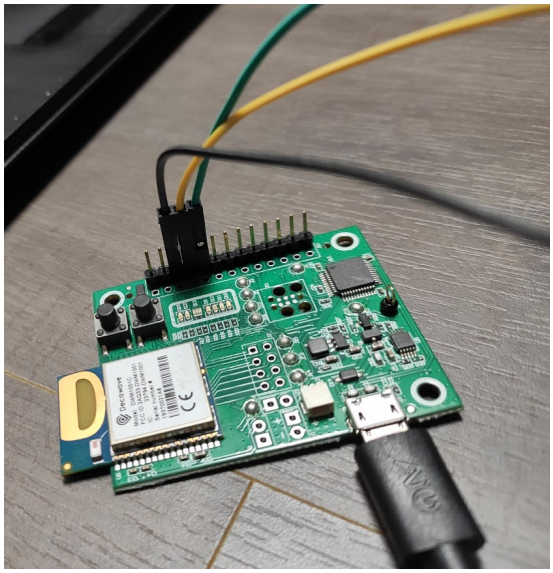

Figure 6.9: DWM1001-dev connection with Pixracer.

3. Plug the flight controller to the PC.

   If you don't need PuTTy you can jump to point 4, otherwise If you want to use PuTTy to monitor what's going on the DWM1001-dev module you can open a serial communication following these 3 steps:

   (a) Open PuTTy and click on "Serial". Then, if you don't know the device name follow point 2, otherwise jump to point 3.

   (b) Open a new terminal and launch command `<< dmesg | grep tty >>`: the last two devices should be the flight controller and the DWM1001-dev module; in principle you should use the last-but-one, but if you want to be sure you can unplug the flight controller and launch again the command `<< dmesg | grep tty >>`, the last device should be the correct one (ACM... or ttyS...).

   (c) Write the name you discovered and set the baudrate to 115200.

4. Open QGroundControl.

5. Go into the MAVLINK console.

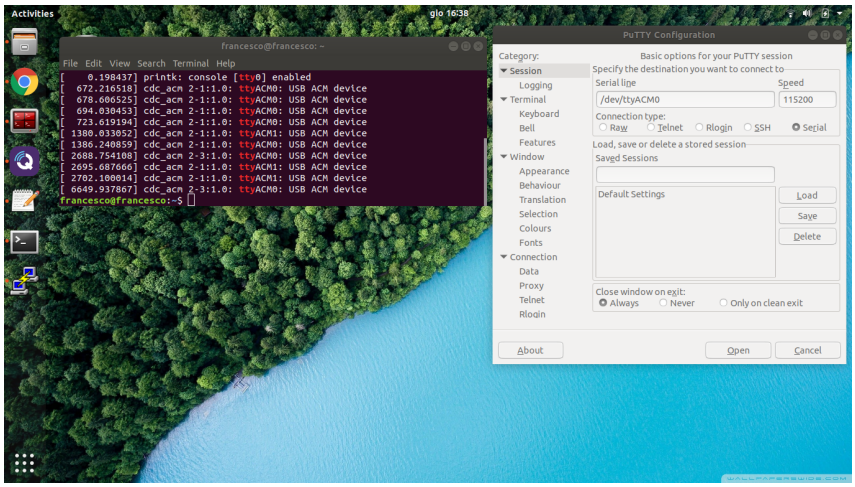

Figure 6.10: PuTTy settings to monitor DWM1001-dev serial communication.

6. Type `<< dwm1001 start >>` to start the driver. You should see something happening on PuTTy if you have prepared it. If you read on the MAVLink console *Got the beginning!* it means that the module was programmed correctly and the serial messages started to flow.

7. Type `<< listener dwm1001 >>` to check whether the driver is working correctly.

The test is now complete and it should be possible to see results similar to the ones in figures 6.11 and 6.12.

Testing 4 anchors and 1 tag

Only at the end of my working period I had the opportunity to test the basic system on the complete setup (4 anchors and 1 tag). Testing setup is basically the same as the one presented in this subsection, but with more anchors. The tag should be always connected to the drone and set as active tag. One anchor should be set as active and initiator, whereas the 3 other anchors shall be set as simple active anchors (no initiator). Moreover, indications about the relative position between anchors should be set by means of the DRTLS Android App; this is an important step, as all the measurements will depend on this setting. For doing this in the best way, it is convenient to place them in a rectangular shape all at the same level. However, the Android app will guide you through the installation step, so do not worry. The test starts again with the same command: `<< dwm1001 start >>` and the result should be the same as the one obtained in the Arduino testing subsection.
For testing purposes I forced logging on SD of the dwm1001 topic so that the estimated position could be seen and compared with visual measurements taken with another drone placed above the entire system. A real complete test was carried out in a flight field out of Turin. During such a test, a square path was drawn, as shown in figure 6.13. The test was a complete success because the estimated position is not only reliable (almost no missing data), but also very close to the real movements if compared with the video coming from the drone. Nevertheless, the maximum working range was a square of 5 meters by 5 meters since when we doubled the square dimension, the result was a complete failure as only one anchor was found. Further test will be done to investigate about this problem.

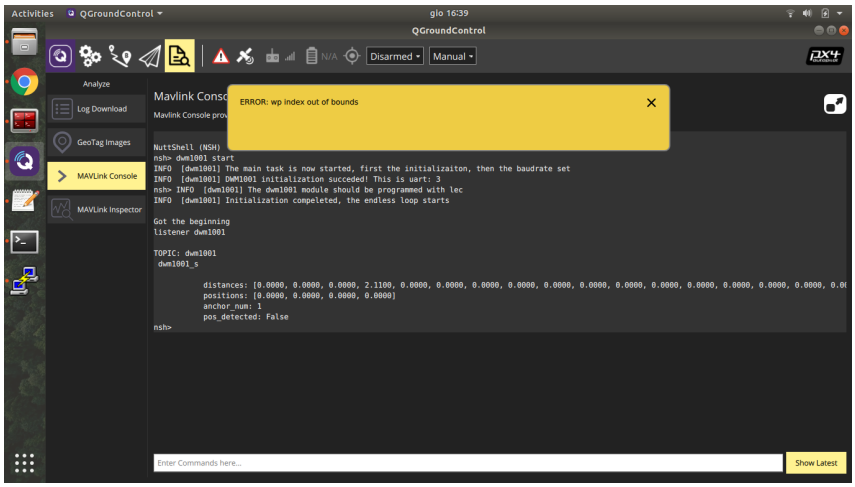

Figure 6.11: Test output.

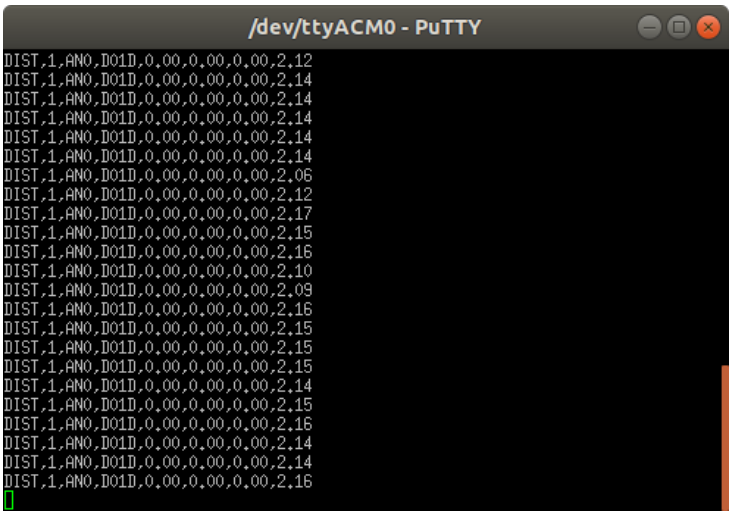

Figure 6.12: Example of PuTTy output during a test.

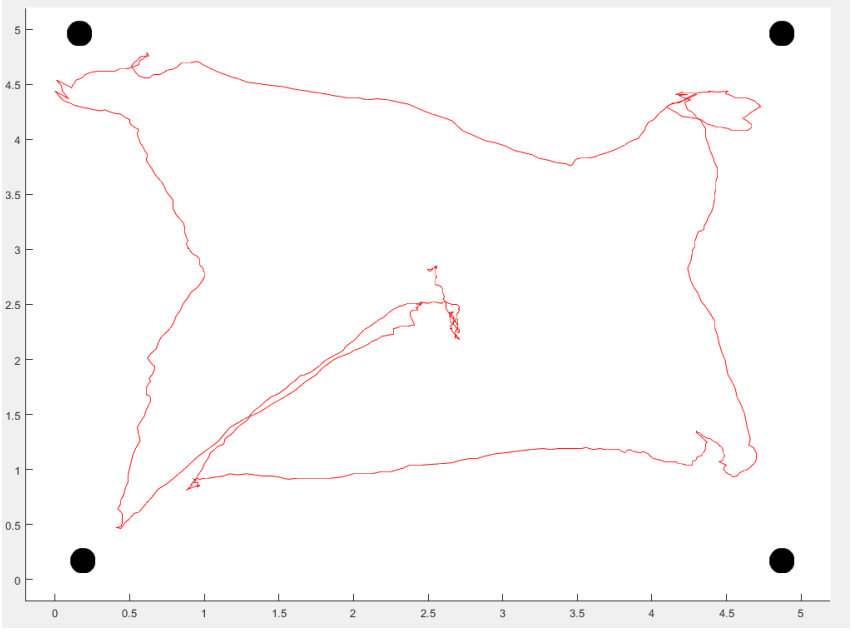

Figure 6.13: Real test of the UWB driver performing a square path.

To summarize:

1. Make sure the firmware is correctly uploaded on Pixracer (`make px4_fmu-v4_default upload`).
2. Power on all the modules and program them with the DWM app: set all the initial positions (measuring them) and check that one anchor is the initiator (active), and the tag is an active tag.
3. Power on the Pixracer and the stabilization module `servo_control` will automatically start. The driver will not be initialized automatically, it must be started manually using a specific command within the MAVLink console. However, before starting it, the logger shall be started.
4. Start the logger using `logger on`.
5. Start the driver using `dwm1001 start`. You will be sure about the initialization when you see "Got the beginning" on the MAVLink console. If you don't, the driver did not recognize the serial message coming from the tag, which probably means that the connection is not correct.
6. When the flight mission is over, you can read the log file accessing the microSD log directory. For the conversion use `pyulog` (`https://github.com/PX4/pyulog`). Copy the correct .ulg file in a directory you know and run the command `ulog2csv ulog file.ulg` in a new terminal. This procedure will create a csv human-friently document in that folder containing all the logs.

6.3.2 Fault tolerance testing using Arduino

Goal and expected results

The goal of this test is to make a sort of fault tolerance test of the system simulating wrong messages sent over the serial port. The expected result is a the same as the DWM1001-dev: PX4 should think of being receiving messages coming from DWM1001-dev.

Needed gear

- Arduino MEGA (or whatever Arduino having more than 1 RX-TX channels, Arduino UNO cannot be used).
- Pixracer
- PC with my version of the PX4 Firmware (https://github.com/francimala/Firmware) and QGC.

Setup

Before plugging anything to the PC, it is necessary to connect Arduino and PX4. For doing this, only three cables are needed: TX, RX and GND (the same GND must be guaranteed linking them together). Conversely with respect to the previous test where TX cables had to be linked together, here the connection is slightly different:

- Black cable is the Pixracer/Pixhawk ground and must be connected to Arduino MEGA GND.
- Yellow cable is Pixracer/Pixhawk TX and must be connected to Arduino MEGA RX.
- Green cable is Pixracer/Pixhawk RX and must be connected to Arduino MEGA TX.

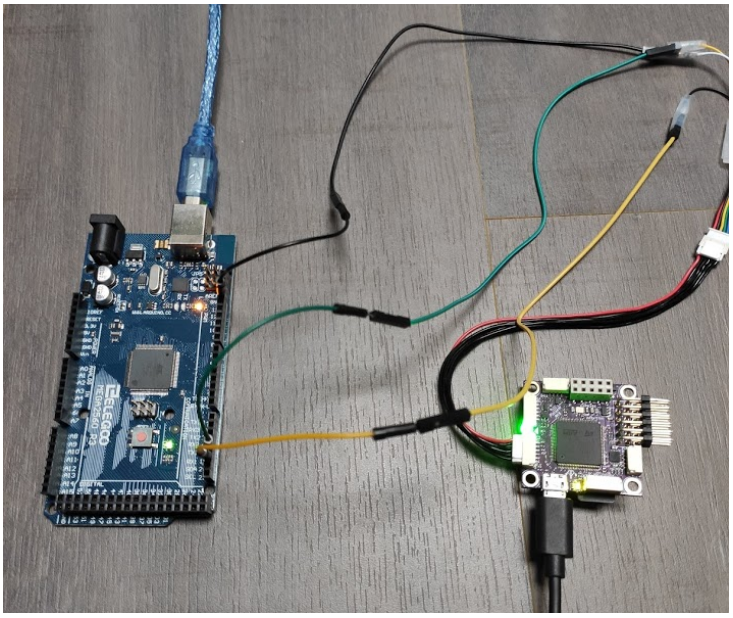

Figure 6.14: Connection between Arduino and Pixracer.

Once completed the connection, both Arduino and PX4 can be plugged to the PC using their specific cables. Arduino should be programmed with the sketch I wrote that is downloadable at this link. Whereas Pixracer must be equipped with my PX4 version. Once everything is uploaded, it is possible to start the test by opening QGC and starting the driver into the MAVLINK console (using the same command used in the previous test `<< dwm1001 start >>`). The result should be the same as before, but with different payloads periodically.

6.4 Troubleshooting

6.4.1 Camera and servo coexistence

The first part of the work was focused on writing a module able to stabilize camera pitch while flying on Pixracer, since it is not implemented by default. Having 6 pins available, the idea was to use the first 4 in PWM mode to control the 4 motors, the 5th one in PWM mode to stabilize servo, and the last one in GPIO mode to switch on/off the camera when needed. Unfortunately, as stated in developer guide, it is not possible to set pins 5 and 6 to different modes, because pins 5 and 6 are managed by the same timer. In fact, setting pin 5 as PWM output and pin 6 as GPIO, the result is a conflict, freezing either the stabilization or the camera (no values published on topics). One of the first solutions was to try simulating a GPIO using PWM, setting duty cycle to 1 or 0 depending on the value to be simulated (high or low). Unfortunately, this solution was not feasible, because the actual output value was never a real high or low.

Another idea was to change the output on the fly, that is setting pins 5 and 6 as GPIO only when camera needed to start (removing stabilization for a certain time). Again, such an approach was not feasible because changing this parameter requires rebooting the drone.

The only solution using the selected camera is to start recording manually before mission starts and then select interesting parts in post-production.

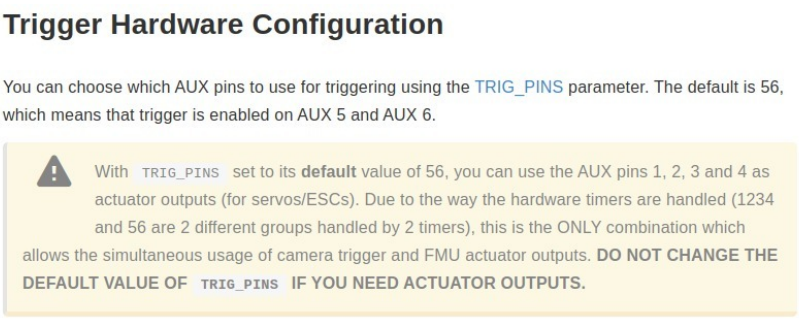

Figure 6.15: TRIG_PINS parameter constraint.

6.4.2 `actuator_control` conflict

In the last part of section 5.2.2 the goal was to achieve the simultaneous presence of the RC passthrough and stabilization contributions. However, without introducing a support topic, the two involved modules, `rc_update` and `servo_control`, were publishing on the same topic (`actuator_control`) making the application unusable due to continuous conflicts (figure 6.16).

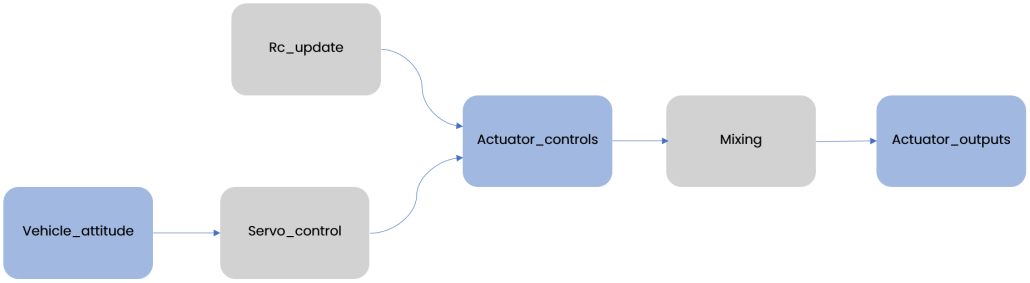

Figure 6.16: Modules and topics involved when creating conflict between RC passthrough and pitch stabilization.

The proposed solution rely on a support topic, `actuator_control_rc`, that is used as intermediate step between `rc_update` and `actuator_control`. The module `rc_update` now publishes on `actuator_control_rc` instead of `actuator_control`, so that the stabilization module (`servo_control`) can read the information coming from the operator, integrate them with the stabilization strategy and publish the resulting value to `actuator_control` topic. Through this approach only one module publishes on `actuator_control`, therefore no conflict holds. Figure 6.17 shows the final chain.

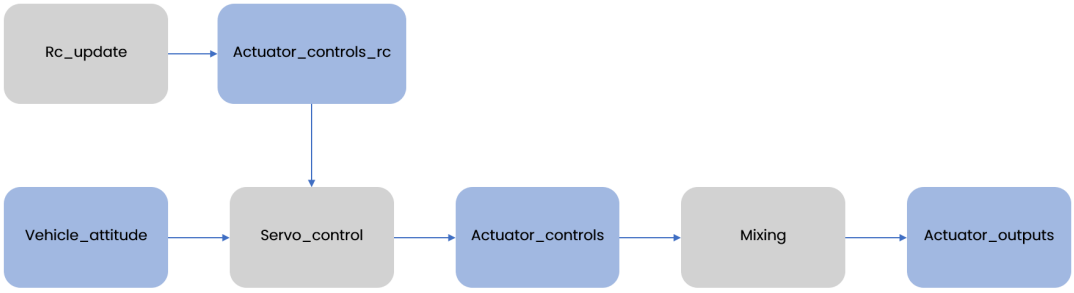

Figure 6.17: Topics involved in the final version of `servo_control` module.

6.4.3 Work queue driver

As previously stated, in PX4 drivers are contained in modules which could be either developed as stand alone tasks or as part of a work queue. The original idea was to make two versions of the same driver, one for each of the two approaches, so to compare them and use the most efficient one. Unfortunately, developing it as a part of a work queue was not successful, as the driver was never scheduled, despite being correctly compiled and initialized. After several days spent trying to understand what was going wrong, we decided to give up, since the first approach was returning quite good performances both in terms of reliability and memory usage.

6.4.4 The carriage return dilemma

Carriage return is a control character used to reset line going back to the beginning. Given this definition it may seem very similar to new line character (\n), though they are different. In fact, carriage return is represented by a different character: \r. However, depending on the used system, they may be used in some peculiar ways to represent the ENTER key.

Developing the UWB application I faced this problem twice:

1. When programming the DWM1001-dev board using command reset plus ENTER.
2. When detecting the ENTER command into messages sent by DWM1001-dev over the serial port.

When dealing with reset command I discovered that ENTER was considered valid if represented as \r. In order to understand it, since the application did not want to start, I used PuTTY to program the board using the keyboard (where ENTER is unequivocally defined) seeing what was printed on console; after some tests I figured out that the correct character was \r. On the other side, when parsing the message sent over the serial port, I had to understand the ending character to spot it and define the end of the instance. In this case, the ENTER command was represented by the sequence \r\n; I figured it out following a similar approach to the one presented in the first case.

6.4.5 Maximum dw1001 area coverage

When I finally had the opportunity to test the real system with 5 different modules (1 tag and 4 anchors) I noticed a problem that needs to be investigated a little more. When performing a test with 4 anchors placed in a square shape of 5 meters length, everything worked fine: the tag was able to get information from all the anchors and the estimated position was reliable. However, when increasing the square dimension to 10 meters, only one anchor was detected by the tag. This problem could be either related to a coverage constraint or to a wrong environment setting (DRTLS app).

Conclusions and further development

7.1 Conclusions

The goal of this thesis was to customize the open-source autopilot PX4 in order to introduce non-standard components focusing on ultralight gimbal and UWB modules. As demonstrated by this work, it is possible to integrate both technologies on any drone, including ultralight builds. In particular, the gimbal can be implemented using a lightweight servomotor, controlled by a firmware module able to both set the desired camera orientation according to RC inputs and stabilize images. On the other side, UWB modules can be integrated leveraging a custom serial driver that programs the tag, parses the received message and publishes the correct information in specific topics to be used by the flight stack.

7.2 Future development and use cases

This work was only the first step of many other projects for service robotics under development at PIC4SeR. Here some examples of applications that will take place in the near future.

7.2.1 Indoor navigation

Indoor navigation is definitely one of the main use cases under development, since all GPS-based applications cannot implement such a feature, although being very important for safety and rescue operations. The idea behind indoor navigation is to have anchors placed in a pre-defined position inside a building, creating a set of known reference points to be used to localize the drone (equipped with a UWB tag). Then, the flight stack should be modified so that the GPS is bypassed and the drone can fly relying only on UWB technology. This application can be implemented in a relatively easy way, since the Decawave firmware is already prepared to provide position estimation given a set of known anchors, hence only the flight stack should be modified and tested.

7.2.2 Swarm navigation in harsh conditions

On the other side, one of the final and most futuristic use cases of such a study, is to replace fixed anchors with mobile ones mounted on drones. The idea is to create a swarm composed by a set of reference drones (mounting UWB anchors) and a set of drones to be localized (mounting UWB tags). This technology could be used in all those environments

where buildings are no longer available (fires, post earth quake scenarios) and no GPS signal is present. This working conditions require anchor positions to be variable, therefore the algorithm should be able to define the coordinate system in a dynamic fashion on the fly. Unfortunately, at the moment this thesis was written, the anchors positions had to be set manually in a static way. However, a project under development at PIC4SeR, is aimed at solving this problem so that the application could really take place. Once the UWB algorithm will be prepared, the two projects could be merged together creating such an incredible and useful technology.

7.2.3 Development of new drivers

This work represented only the starting point for the customization of the autopilot PX4, paving the way for any further development concerning new sensors integration and firmware customization. In fact, concurrently to the development of this thesis, I created a short guide for the introduction of new firmware modules (like the one I developed for the stabilization) and new drivers. Such a guide is intended to help all the students and researchers willing to delve into PX4-based projects.

Figure 7.1: Guidelines table of contents

Bibliography

[1] *Adding new airframe*. Accessed: 2020-04-21. URL: https://dev.px4.io/master/en/airframes/adding_a_new_frame.html.

[2] *Apache Software Foundation*. Accessed: 2020-03-28. URL: https://cwiki.apache.org/confluence/display/NUTTX/Nuttx.

[3] *Ardupilot*. Accessed: 2020-04-27. URL: https://ardupilot.org/ardupilot/.

[4] *Ardupilot*. Accessed: 2020-04-27. URL: https://ardupilot.org/dev/docs/learning-ardupilot-introduction.html.

[5] *Ardupilot Flight Modes*. Accessed: 2020-04-27. URL: https://ardupilot.org/copter/docs/flight-modes.html.

[6] *Ardupilot Threading*. Accessed: 2020-04-11. URL: https://ardupilot.org/dev/docs/learning-ardupilot-threading.html.

[7] Brescianini et al. *Nonlinear Quadrocopter Attitude Control: Technical Report*. Jan. 1970. URL: https://www.research-collection.ethz.ch/handle/20.500.11850/154099.

[8] *ChibiOS*. Accessed: 2020-04-279. URL: http://chibios.org/dokuwiki/doku.php.

[9] *ECL EKF*. Accessed: 2020-03-21. URL: https://docs.px4.io/v1.9.0/en/advanced_config/tuning_the_ecl_ekf.html.

[10] *FastRTPS bridge*. Accessed: 2020-04-20. URL: https://dev.px4.io/master/en/middleware/micrortps.html.

[11] *Flight Modes*. Accessed: 2020-04-05. URL: https://dev.px4.io/master/en/concept/flight_modes.html.

[12] *Full parameter reference*. Accessed: 2020-04-12. URL: https://docs.px4.io/v1.9.0/en/advanced_config/parameter_reference.html.

[13] *Gimbal Mount Control*. Accessed: 2020-04-02. URL: https://dev.px4.io/master/en/advanced/gimbal_control.html.

[14] *Hardware Architecture*. Accessed: 2020-04-29. URL: https://dev.px4.io/master/en/hardware/reference_design.html.

[15] *I2C BUS*. Accessed: 2020-04-21. URL: https://dev.px4.io/master/en/sensor_bus/i2c.html.

[16] A. Koubâa et al. "Micro Air Vehicle Link (MAVlink) in a Nutshell: A Survey". In: *IEEE Access* 7 (2019), pp. 87658–87680.

[17] *MAVLink*. Accessed: 2020-03-26. URL: https://dev.px4.io/master/en/middleware/mavlink.html.

[18] *MAVLink*. Accessed: 2020-04-10. URL: https://mavlink.io/en/about/overview.html.

[19] *Mixing and Control Groups*. Accessed: 2020-03-22. URL: https://dev.px4.io/master/en/concept/mixing.html#mixing-and-actuators.

[20] *Modules and Commands (communication)*. Accessed: 2020-03-19. URL: https://dev.px4.io/master/en/middleware/modules_communication.html.

[21] *Modules and Commands (controllers)*. Accessed: 2020-03-15. URL: https://dev.px4.io/master/en/middleware/modules_controller.html.

[22] *Modules and Commands (drivers)*. Accessed: 2020-03-21. URL: https://dev.px4.io/v1.9.0/en/middleware/modules_driver.html.

[23] *Modules and Commands (estimators)*. Accessed: 2020-03-21. URL: https://dev.px4.io/master/en/middleware/modules_estimator.html.

[24] *Modules and Commands (systems)*. Accessed: 2020-03-20. URL: https://dev.px4.io/master/en/middleware/modules_system.html.

[25] *NuttX Initialization*. Accessed: 2020-05-21. URL: https://cwiki.apache.org/confluence/display/NUTTX/NuttX+Initialization+Sequence.

[26] *NuttX Overview*. Accessed: 2020-04-04. URL: https://raw.githubusercontent.com/engehcall/technology/master/NuttX/nuttx-overview.pdf.

[27] *NuttX Tasking*. Accessed: 2020-05-21. URL: https://cwiki.apache.org/confluence/display/NUTTX/NuttX+Tasking.

[28] *Open Source for Drones*. Accessed: 2020-03-15. URL: https://px4.io/software/software-overview/.

[29] *Pixhawk Customization*. Accessed: 2020-04-05. URL: http://nutshellking.com/articles/xue-xi-zong-jie/customize_Pixhawk/.

[30] *Plan file formats*. Accessed: 2020-09-03. URL: https://dev.qgroundcontrol.com/master/en/file_formats/plan.html.

[31] *PPM*. Accessed: 2020-04-24. URL: https://en.wikipedia.org/wiki/Pulse-position_modulation.

[32] *PX4 Architectural Overview*. Accessed: 2020-03-18. URL: https://dev.px4.io/master/en/concept/architecture.html.

[33] *PX4 Controller Diagrams*. Accessed: 2020-03-19. URL: https://dev.px4.io/master/en/flight_stack/controller_diagrams.html.

[34] *QGroundControl*. Accessed: 2020-04-04. URL: https://docs.qgroundcontrol.com/en/.

[35] *Sensor Combined*. Accessed: 2020-03-26. URL: https://dev.px4.io/master/en/middleware/modules_system.html.

[36] *Sensors module*. Accessed: 2020-03-22. URL: https://github.com/PX4/Firmware/blob/master/src/modules/sensors/sensors.cpp.

[37] Simone Silvestro. *Optimization of an Ultralight Autonomous Drone for Service Robotics*. Dec. 2019. URL: https://webthesis.biblio.polito.it/13238/.

[38] F. B. Sorbelli and C. M. Pinotti. "Ground Localization with a Drone and UWB Antennas: Experiments on the Field". In: *2019 IEEE 20th International Symposium on "A World of Wireless, Mobile and Multimedia Networks" (WoWMoM)*. 2019, pp. 1–7.

[39] *SPI*. Accessed: 2020-04-24. URL: `https://en.wikipedia.org/wiki/Serial_Peripheral_Interface`.

[40] M. Strohmeier et al. "Ultra-Wideband Based Pose Estimation for Small Unmanned Aerial Vehicles". In: *IEEE Access* 6 (2018), pp. 57526–57535.

[41] *System startup*. Accessed: 2020-03-29. URL: `https://dev.px4.io/v1.9.0/en/concept/system_startup.html`.

[42] *Thread definition*. Accessed: 2020-04-04. URL: `https://en.wikipedia.org/wiki/Thread_(computing)`.

[43] J. Tiemann, A. Ramsey, and C. Wietfeld. "Enhanced UAV Indoor Navigation through SLAM-Augmented UWB Localization". In: *2018 IEEE International Conference on Communications Workshops (ICC Workshops)*. 2018, pp. 1–6.

[44] *UART*. Accessed: 2020-04-24. URL: `https://en.wikipedia.org/wiki/Universal_asynchronous_receiver-transmitter`.

[45] *UAVCAN*. Accessed: 2020-04-21. URL: `https://uavcan.org/`.

[46] *uORB*. Accessed: 2020-04-06. URL: `https://dev.px4.io/master/en/middleware/uorb.html`.

[47] *UWB technology*. Accessed: 2020-04-21. URL: `https://www.eliko.ee/uwb-technology-indoor-positioning/`.

Acronyms

Acronyms

BSD Berkeley Software Distribution.

EKF Extended Kalman Filter.

ESC Electronic Speed Control.

GCS Ground Control Station.

GPL General Public License.

IMU Inertial Measurement Unit.

LPE Local Position Estimator.

NED North East Down coordinate system.

PWM Pulse Width Modulation.

RC Radio Controller.

ROS Robotic Operating System.

RTOS Real-Time Operating System.

UWB Ultra Wide Band.

VTOL Vertical Take-Off and Landing.