POLITECNICO DI TORINO

Master Degree in Computer Engineering, Automation and Intelligent Cyber-Physical Systems.

Firmware Design and Implementation for a Quadrotor UAV

Yuri Sala

Supervisor: Elisa Capello

Co-Supervisor: Davide Carminati

A.a. 2020/2021

Abstract

In recent years UAVs (unmanned aerial vehicles), also called drones for simplicity, are becoming more and more widespread all over the world. Born for military employment, now they are used for everyday tasks and also in the field of research. Here we want to show our implementation of a working firmware for UAV.

The firmware main task is to link the software and the hardware. This is a key aspect because all sensors, motors and communication board must exchange a big quantity of data with each other. Moreover, those data are inputs or outputs of the software that is the 'brain' of the drones. The software is loaded into a ST micro-controller and is capable to get all needed information from others hardware peripherals thanks to dedicated libraries. A finite state machine that controls the global state of the drone handles his behaviour. Starting from the initialization, it switches in a runtime phase only if no errors occur. Errors detection is in charge of the commander functionality, which is always active like an external supervisor. Moreover, the firmware should include the data treatment, to guarantee correctness and synchronization of data with a specific module. Finally, another module is the control algorithm, for studying the drone behaviour. For safety motivations, also a manual mode is added with which you can control the drone using a joystick. All of these modules are implemented for a quadrotor UAV, designed at Politecnico di Torino in the Department of Mechanical and Aerospace Engineering. The firmware, afterward analysed, is overall working and it can be used with the target hardware. In the future, more functionalities can be added with different purposes.

Contents

Chapter 1

Introduction

1.1 Premise

Nowadays UAVs (unmanned aerial vehicles) are used for for a lot of applications either military either civil. But not only, they are also used by people for their personal interest because it's very easy make use of them.

This is possible because UAVs are different in shape, aim and even by definition. In the past years some different names and definitions were given to these objects. UAV can be interpreted as 'Unmanned Aerial Vehicle' or as 'Uninhabited Aerial Vehicle', then UAVs are also named UAS (Unmanned Aircraft System), ROAs (Remotely Operated Aircrafts), RPVs (Remotely Piloted Vehicles) or simply drones. Other names derives from physical characteristics of them, such as MAV (Micro Aerial Vehicles), which is used for for those who have small dimensions (length, width, or height of 15 cm or less).

More precisely, there are various technical definitions issued by different organizations: the AIAA (American Institute of Aeronautics and Astronautics) defines a UAV as:

> "an aircraft which is designed or modified, not to carry a human pilot and is operated through electronic input initiated by the flight controller or by an onboard autonomous flight management control system that does not require flight controller intervention."

The definition of Unmanned Vehicle given in the '2007–2012 Unmanned systems roadmap' is:

> "A powered vehicle that does not carry a human operator, can be operated autonomously or remotely, can be expendable or recoverable, and can carry a lethal or nonlethal payload. Ballistic or semi-ballistic vehicles, cruise missiles, artillery projectiles, torpedoes, mines, satellites, and unattended sensors (with no form of propulsion) are not considered unmanned vehicles. Unmanned vehicles are the primary component of unmanned systems."
>
> (U.S. Department of Defense. Office of the Secretary of Defense, 2007)

Similarly, EASA (European Aviation Safety Agency) defines UAS as:

> "An Unmanned Aircraft System (UAS) comprises individual system elements consisting of an "unmanned aircraft", the "control station" and any other system elements necessary to enable flight, i.e. "command and control link" and "launch and recovery elements". There may be multiple control stations, command and control links and launch and recovery elements within a UAS."
>
> (European Aviation Safety Agency 2009)

Moreover, UAVs can be classified according to their structural and mechanical characteristics. Knowing that the aim of the thesis is to develop a firmware for relatively small UAV, here is reported a brief description of some kind of drones to which the code implemented can be employed for real applications. Those type of UAVs are very widespread because they are quite easy to use and low-cost. However, given their small size there are many technical problems during the development phase, especially regarding the installation of their power supply module, communications apparatus and sensors.

UAVs can be roughly divided by looking at the shape of their wings. Basically there are three main types: fixed-wings, rotor-wings and tilt-rotor wings.

- **Fixed wing:** As shown in Figure 1.1, this kind of UAV has the classical shape of airplanes with a fuselage, one wing (or two in some cases), a tail and one or more propeller. They are similar to normal airplanes so they can have any dimension. Fixed wing aircraft can fly fast and for relatively long distances but their main disadvantages are that they cannot hover or fly at low speeds. It is also difficult, for fixed-wing aircrafts, to automatically takeoff and landing or doing these maneuvers in a quite limited area.

Figure 1.1: Example of fixed-wing aircraft.

- **Rotor-wing:** Due to the limitations of fixed-wing aircraft, recently, the research of rotor-wings is increasing. Their main advantage is the capability to hover in a fixed position, which is the principal skill needed to perform various important tasks such as monitoring at fixed points or surveillance/recordings from the sky. Another useful maneuver that only rotor-wing UAVs can perform are the vertical takeoff and landing. Normally, this type of aircraft are small, thus they often can be classified as MAVs. In addition, because they are lightweight, even if they crash the damage to the vehicle could be relatively small. As shown in Figure 1.2, rotor-wing UAVs can be classified by the number of propellers, in particular, there is a single main differentiation: single and multi-rotor vehicles. Compared with the single-rotor vehicle, multi-rotor units obtain their lift using two or more propellers, so a great variety of airframe can be designed considering that the development of them is not so much more demanding. Furthermore multi-rotor UAVs have a better stability and it's easier to control them.

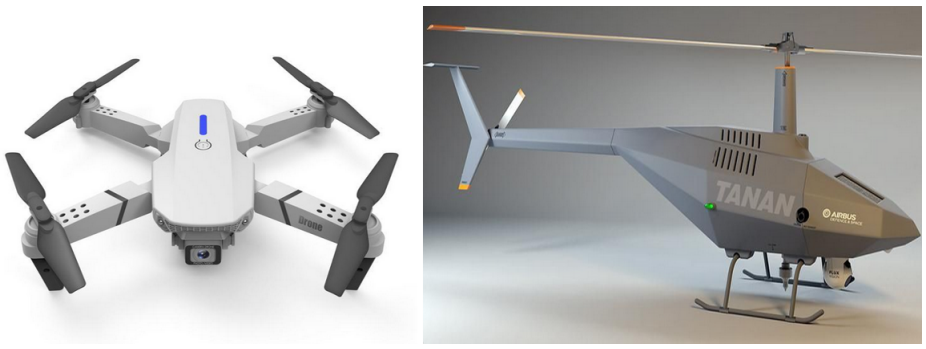

Figure 1.2: Example of quad-rotor and single-rotor (helicopter) aircraft.

- **Tilt-rotor wing:** Afterwards is born a new category of UAVs, the tilt-rotor type. They combine the advantages of both types thanks to rotors with a tilt mechanism on the tip of their wings. This is the key feature because it allows to perform vertical takeoff and landing, hovering, and high cruising speed flight by changing the angle of the rotors. The overall mechanics is more complex but the advantages are very noticeable. Figure 1.3 shows an example of a tilt-rotor wing airplanes and how the tilt mechanism works.

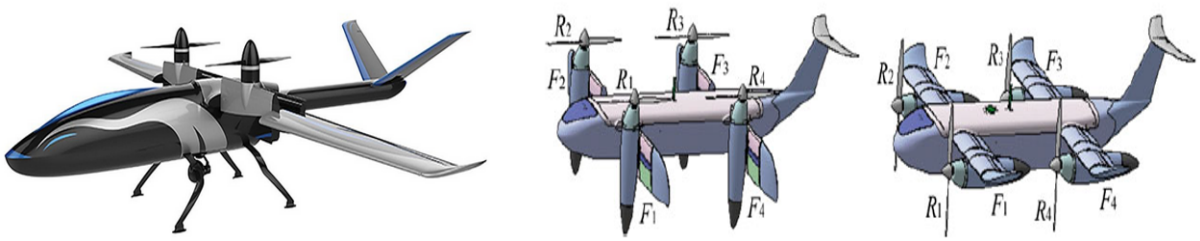

Figure 1.3: Example of tilt-wing aircrafts.

1.2 Work general explanation

The objective of this thesis is to develop a firmware that is able to make a drone work and fly in a safe condition. A firmware is a program loaded into a micro-controller and it is the main player during the working phase of an electronic object. It is a general term that can be used for a lot of applications, not only for drone. Its aim is to startup and control all the hardware components needed to achieve the goal, so we can say that the firmware is a software that should communicate with the hardware. In our case, its fundamental objective is to make a drone flies and it is possible only if all the hardware components work in a proper way, thus all of them should communicate (by means different protocols) with the micro-controller. Some essential hardware components are: battery, sensors and motors; then, for example, could be present an antenna to allow the drone to communicate with a manual radio control or with a work-station that exchanges some data with it.

Searching on the web it is possible to find several firmware implementations and here, for sake of clarification, two of them are briefly explained.

- **PX4:** It is a very powerful and flexible open source autopilot. It can be used to controls many different vehicle frames/types, including aircraft (multicopters and fixed wing), ground vehicles and underwater vehicles. It's compatible with a great choice of hardware components for vehicle controller, sensors and other peripherals. It includes different flight modes and safety features. PX4 architecture is quite complex as can be clearly see in Figure 1.4. The main part is the flight controller module that is managed by a state machine. This part contains the controllers and the interfaces towards sensor and motor drivers. All the data, into the software, are exchanged through virtual message busses. Some of these data are saved on a on-board storage and other (or all of them) are send to external devices by means communication protocols (for example using Mavlink). [*PX4 autopilot user guide* (2021)]

- **ArduPilot:** Also ArduPilot is an open source autopilot and its structure is very similar to PX4. It has a main flight control part but it is implemented using independent libraries instead of modules. Another difference is the usage of a dedicated layer for linkages to external hardware devices. Instead, external communications are treated at the same way as PX4, it uses Mavlink protocol/messages. An overview of ArduPilot firmware structure is shown in Figure 1.5. [*ArduPilot web page* (2021)]

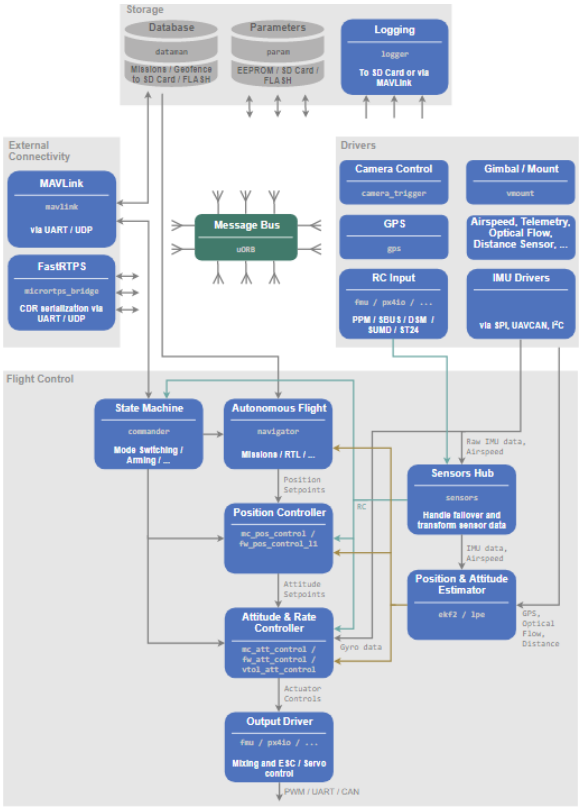

Figure 1.4: PX4 firmware architecture.

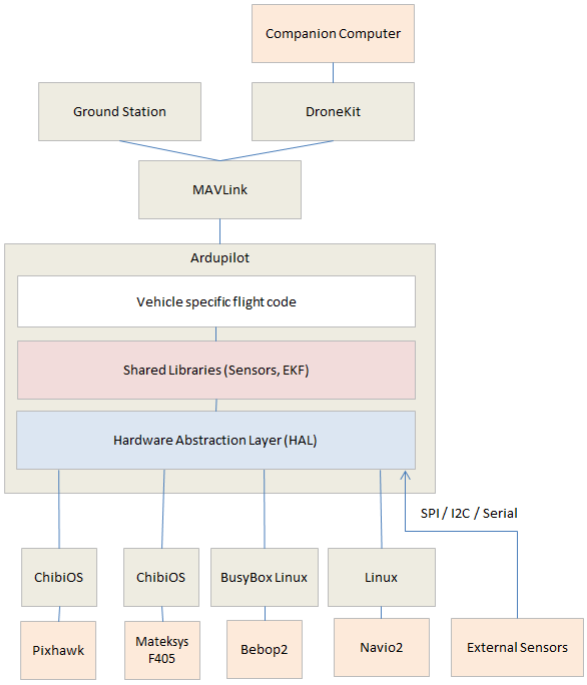

Figure 1.5: ArduPilot firmware architecture.

In this thesis a new lightweight firmware is implemented. The high level structure is equivalent to the others previously described but, in general, fewer functionalities are made available. First of all, this is not a extremely flexible code but it mainly depends on the hardware systems for which is built. Obviously some low level features can be shared to all system but some other modules, like the sensors management, should be adapted for different hardware. The code is generated with a specific target, that is the drone under development at Politecnico di Torino, in the Flight Mechanics Research Group (Dept. of Mechanical and Aerospace Engineering) (Figure 1.6). This is a QuadRotor MAV whose main goals are indoor applications that are performed in an autonomous way. As described in the article published by Capello et al. (2020), the control schemes are already designed and consolidated, so it was necessary to insert these schemes into a firmware to test them in a real hardware. This control scheme has been developed for another drone but it can be easily adapted for other quadrotors by changing some parameters. This is possible thanks to a very useful toolbox of Matlab/Simulink (*Matlab web page* (2021)) that allows you to export in C/C++ code format the control schemes designed. After having done it, you can import those auto-generated files in your external project. Our firmware, the one under analysis, has been developed in C/C++ code using Visual Studio Code (*Visual Studio web page* (2021)), with the help of the PlatformIO extension. PlatformIO (*Platformio web page* (2021)) is a software platform for embedded systems, it allows software development for cross-platform, cross-architecture and multiple frameworks. Its key feature is that developers no longer have to manually find and assemble an environment of tool-chains, compilers, and library dependencies to build applications for a specific target but they only have to specify on what micro-controller the code should be loaded.

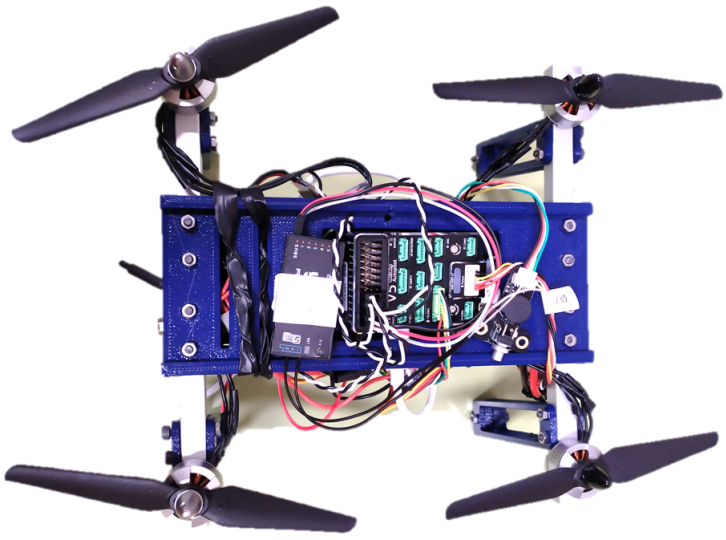

Figure 1.6: Drone developed at Politecnico di Torino

Chapter 2

Firmware

2.1 Operating System

The firmware implementation is the most important part of the thesis. The work started by searching on the web how to handle similar projects and the result was that they are structured in a quite fixed manner. For UAV projects, the centre of the program is the flight control task, then boundary functionality are developed around it. Some of them are more important, also mandatory, instead other are extras. After those studies, the structure of our firmware was defined.

As always, the central module is the flight control procedure that is under management of a Finite State Machine (Section 2.2). That strategy is very suitable for real applications that should guarantee particular behaviours due to safety criteria. Drones, as all the other objects that fly, are extremely subject of those notices. Parallel to the finite state machine lives the Commander module (Section 2.3) that is a supervisor of the global state of the drones. In few words, it checks if all hardware peripherals and some software modules work correctly. Other required modules are the sensor handler and the controller (Section 2.7). The former should read in a proper manner the data acquired by the physical sensors; then these data are used as inputs for the latter. For this motivation, sensors and controller are deeply linked by the data flow. In fact information from sensors can be received at a different rate respect to the rate with which the controller make use of them. That aspect must be considered during the development of the controller, action that is done separately from the one of the firmware. The controller should guarantees real-time performance due to the nature of the system, which has very fast dynamics. Real-time programs must guarantee response within specified time constraints, which are often understood to be in the order of milliseconds, and sometimes less. The controller should receive data, process them, and return the results sufficiently quickly to affect the system in a suitable way. These requirements lead to particular design either hardware either software. The hardware must be able to perform the needed calculations in an adequate time interval, so it must be chosen with special attention. At the same way, the controller must be able to maintain the drone in stable conditions and control it to make it reach the goal. Back to the main discussion, we were talking about data handling. For this purpose some modules should be designed (Section 2.6). In our case we added three of them: a lock functionality, a global, shared structure and a storage backup. Last, but very

useful, there is the Command Line Interface module (Section 2.4) that is mostly used during the development phase to interact with the micro-controller. It allows sending predefined commands that are received and interpreted directly by the micro-controller to test something, or to get information. What you can do with such commands is modifiable only by the programmer, so functionalities can be added or removed only in the development phase. That's why a lot of functionalities are added and then removed when, for example, a problem is found and then solved.

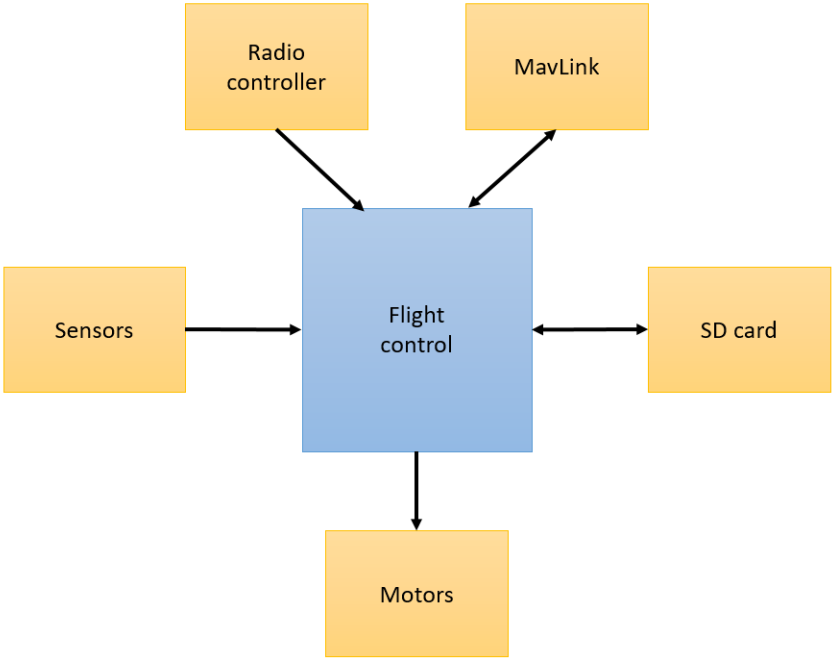

Figure 2.1: Firmware architecture.

That, briefly explained (and showed in Figure 2.1), is the global structure of the firmware that has been developed. In the following sections all the modules are analyzed singularly. Most of them interact with each other but the description will be focused on a single module at a time.

Before entering in the details of the sub parts of the program, it is worthwhile to get an overview of the project from the point of view of the tools used, the actual code implementation and the low-level features. The code is written in C/C++ language with the help of Visual Studio, installed on a Unix OS. The foundations of the project is the last version of Mbed OS (*MbedOS web page* (2021)), which is an open-source operating system for Internet of Things (IoT) Cortex-M boards. The full profile of Mbed OS is an RTOS (Real Time Operating System), so it supports multi-thread real-time software execution. It provides primitives functions, allowing drivers and applications to rely on threads, semaphores, mutexes, timers and other useful RTOS features. We make use of them for handling each part of the program:

- **Thread:** The Thread class allows defining, creating and controlling parallel tasks. Also the function *main()* is a special thread function that is started after system

initialization. From the *main()* function other threads are created and started. A thread is a component of the process that allows to execute in a concurrent way different sequences of code instructions. A process can contain more than one thread but a thread cannot be shared between processes. Threads execution is handled by a scheduler that assigns a state at each single thread so that only one of them can be in the *running* state. The others are assigned to different states: *ready* for threads that are ready to run. Once the running thread has terminated or is waiting, the ready thread with the highest priority becomes the running thread. *Waiting* for threads that are waiting for an event to occur. *Inactive* for threads that are not created or terminated. These threads typically consume no system resources.

In our firmware threads are widely employed to separate the execution of functions related to different modules. Starting from the main thread, others are created for sensors readings, control loop, MavLink communications and so on for all modules. The control loop has a particular behaviour because it can be executed with a single thread or more, it depends on how its code is created from Matlab/Simulink. Having so much threads run concurrently is quite expensive, either in terms of computing capability, either for memory usage. For this motivation you have to be careful to not going out of memory because the whole process stops in a critical way. With the help of the Mbed function for thread creation, you can easily set the maximum memory available for each single thread, so it is in charge of the programmer to correctly allocate the whole addressable RAM memory.

Memory management is an important aspect for threads handling, not only for critical situations. In fact, sharing resources, such as memory, is a real benefit of using multi-threads instead of multi-processes, which do not share information. In particular, the threads of a process share its executable code and the values of its dynamically allocated variables, as well as global variables, at any given time. Management of shared resources must be treated with high attention to grant a correct execution of the program.

- **Timer:** Timer class is easy to explain and even easier to understand it's importance for a real-time application. It provides functions to deal with very precise time intervals, either in an automatic way, either in a manual mode (saving start and end time instants). This timer interface allows for measuring precise times with better than millisecond precision. Very similar is the **Wait** class that should only be used for even shorter delays, as microseconds or nanoseconds. This because these functions spin the CPU to produce the delay and it will affect power and multi-thread performance, blocking the platform in deep sleep for the entire duration of the delay.

  These functions are used, in our project, to wait for small intervals during peripherals initialization and also in some threads to execute specific tasks with a defined rate.

- **Event Handling:** It's achieved thanks to the Event and EventQueue classes. The EventQueue class provides a flexible queue for scheduling events, which belong to the former class. Event objects have some tunable parameters, the most important

are delay and period timings, that are modifiable through methods of the class. Other utility functions allow to post an event to a queue or to cancel it. After having posted the events, the queue can be launched. Doing that, the thread which launched the queue stops and will continue its execution only when the queue is stopped. In some cases this is a waste of memory, because a thread is allocated only to call events, so Mbed OS provides two shared queues. This can avoid the need to create private event dispatch threads and reduce the total amount of RAM used. An example of usage of events is to use them for dealing with sensors: reading their data every time they are available, so with a fixed rate depending on each sensor.

- **Mutex:** Mutexes are exclusive locks that prevent multiple accesses to the same resources. They have a lot of applications such as preventing threads to enter in a critical section while a concurrent thread of execution is already accessing that section (race conditions) or preventing the access to the same memory address. These locks can be adopted to generate different type of locks with specific purposes. In our project we used mutex to develop a Read/Write Lock class. Its aim is to optimize the data management so that they can be accessed by multiple readers but only by one writer at a time. This behaviour ensure a faster handling when data are more read than they are written.

- **File System:** The FileSystem class provides a common interface for implementing a file system on a block-based storage device. The main purpose of a FileSystem object is to be instantiated specifying a block-based storage device and a mount point indicated as a string. The mount point can then act as the root directory of any paths in Mbed OS. This gives you access to files inside the file system and then Mbed OS provides the retargeting layer for the standard C library, with which manage files and directories. In particular, we used a FAT file system that is an established disk-oriented file system used in the majority of OS like Windows, Linux and macOS. Due to its age and popularity, the FAT file system has become the standard for portable storage, such as flash drivers and SD cards, the type that we mounted on our drone. The FAT file system is optimized for embedded systems but, thanks to its portability, it provides access to storage from both the embedded system and PC.

- **Drivers:** Drivers provide access to general purpose micro-controller hardware. Drivers typically access a micro-controller's hardware peripherals directly, and may form the lowest level of a protocol or network stack. The Mbed OS driver classes include analog and digital inputs and outputs on development boards, as well as digital interfaces, which allow your board to interface with a computer or external devices. With these drivers, you can read or set the voltage of analog pins, control digital pins individually or grouped into a bus and also control the frequency or duty cycle of a Pulse Width Modulated (PWM) digital output.

  We made a wide use of them in several modules: Serial (UART) drivers provide two classes to interact with hardware, the BufferedSerial class and the UnbufferedSerial class. Serial channels use separate transmit and receive pins to send and receive bytes of data in a sequence. In particular the BufferedSerial class, the one

that we used (Section 2.5), stores the byte(s) available to read from the hardware buffer or the data to be sent to an internal intermediary buffer. Using intermediary buffers allows the UART interface to be used reliably for input from non-interrupt context while avoiding excess spinning waiting for transmission buffer space. SPI drivers are also developed for communications and data transfer tasks. Unlike the normal serial interface, SPI is a synchronous serial communication interface that communicates in full duplex mode using a master-slave architecture. In this case four wires are needed to exchange data. I2C drivers are the last communication drivers used. They combine the characteristics of UART and SPI interfaces. I2C is a two wire serial protocol that allows an I2C Master to exchange data with an I2C Slave. You can use it to communicate with I2C devices such as serial memories, sensors and other modules or integrated circuits. It supports up to 127 devices per bus, so we used it to get data from all mounted sensors. In the end, electrical motors are controlled with PWM drivers, so generating PWM output signals. These drivers allow to setup the fundamental parameters that characterize a PWM signal, which are frequency and duty cycle.

2.2 Finite State Machine

A finite state machine (FSM) or finite state automaton (FSA), finite automaton, or simply a state machine, is a mathematical model of computation. It is an abstract machine that can be in exactly one of a finite number of states at any given time. The FSM can change from one state to another in response to some inputs, which are also named entry actions, performed when entering a state, and exit actions, performed when exiting the state. The change from one state to another is called *transition*. A *state* is a description of the status of a system that is waiting to execute a transition. There is always a special state that is the entry point of the machine, instead isn't always present an ending state or accepted state. This is a common situation for real-time applications where the machine is, most of the time, looping indefinitely around a working state. The behavior of state machines can be observed in many devices in modern society that perform a predetermined sequence of actions depending on a sequence of events with which they are presented. Moreover, FSMs can be used to model problems in many fields including mathematics, artificial intelligence, games, and linguistics. There are two types of FSM: Deterministic Finite State Machine (DFM) and Non-deterministic Finite State Machine (NFM). When implementing DFM, every input causes a state change, and the new state is completely determined by the input. Moreover, the machine can change state only after reading an input. In case a NFM is used, some inputs may allow a choice of resulting states, and some may cause the machine to choke, because there is no new state corresponding to that input. Moreover, the system may be constructed so that it can change state to some new state without reading any input at all. NFM are better to describe natural systems, which have a probabilistic behaviour. On the contrary, DFM can be designed to impose a strict and defined behaviour to artificial objects that must perform same actions for their entire existence. Thus, in our project

we developed that type of FSM.

FSMs are very useful to model the typical flow of execution of a electro-mechanic object. This is particularly important for real-time embedded systems, which are usually highly state dependent. They achieve simplification of a complex dynamic problem by dividing it to a set of conditions. All specific states are not easy to identify, but when the design is completed, the development is faster. The main advantage is to divide the different working phases to allow a separate, and therefore safer, implementation of them. At the same time, it's easier to modify the number of states by adding or removing some of them.

At the moment, the FSM implemented in our firmware is composed by seven states. In Figure 2.2 there is a visual representation of its complete structure. FSM are usually drawn as state transition diagrams, which is a useful way to visualise them. This is a good method to also better understand the overall behaviour.

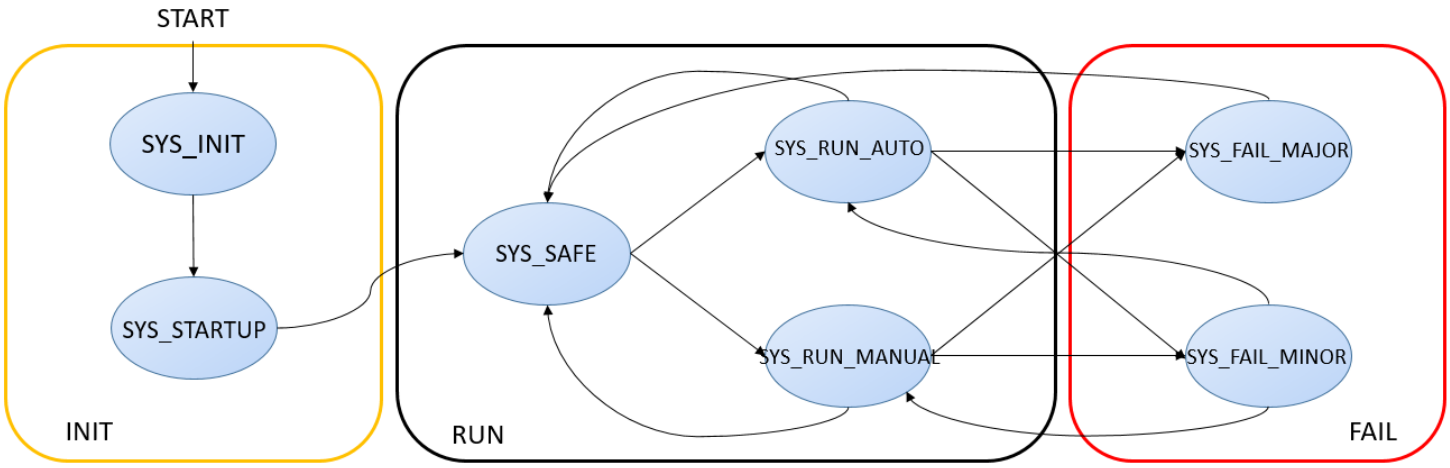

Figure 2.2: Finite State Machine structure overview.

From the global structure three sub-parts can be highlighted: an initialization part, a working part and a failure part. The first one is composed by two states that execute similar tasks but for different hardware or software components. Now, a detailed description of all the states is made and, after that, a pseudo code of partial implementation of the FSM is presented in Code Listing 2.1.

1. **SYS_INIT:** Let's start the description from the first state, the SYS_INIT state. Here the system is initialized performing some actions on the most important peripherals of the UAV. All sensors are waked up and their status is checked with a function provided by the Commander module (Section 2.3), which begin performing its duty. After the sensors, the communications with the radio-controller are started. This is done to guarantee a safe way to control the drone if some critical situations appear during the next steps. Again, for safety reason, the Commander put the system in a disarmed condition so that PWM signals cannot be sent

and the motors, consequently, cannot be turned on. Another peripheral that is checked is the SD card because it could contains the calibration values calculated from previous calibration sessions of the sensors. It is also used to store a log file where run-time information are saved (Section 2.6). In this state no more actions are made because it must be very fast and blocks the system if some fundamental devices can't be detected or show a failure.

2. **SYS_STARTUP:** If all is ok, the machine can proceeds to the following state: the SYS_STARTUP state. This is very similar to the previous one because, even it, is an initialization phase. Now, lower important hardware and software are started, always under supervision of the Commander. First of all, MavLink communications, either transmission either reception, are initialized and checked through the "*HeartBeat*" signal, which acts as an handshake signal. Then control loop threads are launched and PWM module is enabled, even if its output ports are still blocked by the disarmed status of the global system. As before, the Commander is the one which allows the machine to go to the next step, but in this case less strict constraints should be set.

3. **SYS_SAFE:** From this state on, the system enters in the running phase. All peripheral are active and the drone can be enabled to fly. However, motors are not activated until the user gives an input by means the radio-controller or commands of the Command Line Interface. Hence, this state could also be named IDLE state because the drone is not moving and it can be connected to a PC to allow the usage of the CLI. This is the only moment in which you can use it for interacting with the drone. It is mainly useful during the development phase because, with that, you can check the status of the drone or perform predefined tests using the implemented commands and relative procedures. This state is fundamental because it is the starting point of the operative conditions but it is also the ending point. In fact, from here the system can move on to the real active phases and it can return here from each following states due to a failure of the system. The forward transition is triggered either by the radio-controller, either by commands of the CLI, if that is active. Instead, the opposite transition is performed in a controlled way using, again, the radio-controller or, in a non-deterministic way if the system is forced to came back here due to a major failure detection.

4. **SYS_RUN_AUTO:** This should be the state active for the longest time because it corresponds to the automatic flight mode that is the main task of an UAV. Here the motors are controlled by PWM signals that are produced by the controller loop (Section 2.7) while inputs from the radio-controller are ignored, except a command that is needed to return to the SYS_SAFE state. The *main* function cycles continuously on this state until this command is received or a failure occurs. In a normal situation, without failures or explicit commands, the drone will remain in this state until the drone is turned off. Also if it completes its task and goes in a stationary condition, the FMS does not make any transition. Beside that, the status of the drone must always be checked to avoid an undesired behaviour that can put the system in dangerous situations. For these reasons we added a run-time check function, implemented in the Commander module, that is executed at

each loop of the machine. If a failure is detected, the machine is forced to a new state according to the failure type, major or minor.

5. **SYS_RUN_MANUAL:** This is a parallel state to the one just described, the SYS_RUN_AUTO state. It can be considered parallel because they are very similar but they implement a control technique completely different. Here motors are controlled by the user by means a radio-controller, so no help is carried by the electronics. The main duty of this state is to read the inputs received from the antenna and transform them in PWM signals for the motors. As in the previous case, a run-time check function is present, but with different implementation because different modules are active. Also here there is a continuous loop on this state, until the user or a failure leads to a transition. In the early development process this state should be used to test all the components without taking excessive risks. Further on, this state can be changed to facilitate the user to fly the drone. For example a lighter control loop can be added to automatically control its altitude.

6. **SYS_FAIL_MAJOR:** This and the last state are added to handle the failures that could occur during the lifetime of the drone. As they are made by a lot of electronic components, their failures must be detected by the micro-controller. During the thesis it has not been done an accurate analysis of the failure conditions so these states of the machine are not implemented in a complete way. Only main concepts are take into account, so has been done a distinction between major, or catastrophic, failures and minor, or temporary, failures. Until now, specific hardware failures are not assigned to one of the two classes but they are all addressed to this SYS_FAIL_MAJOR state. From this state no recovery actions are implemented but the system is directly forced to switch in the SYS_SAFE state. This means that motors are turned off instantaneously, action that could lead to a catastrophic damage for the drone.

7. **SYS_FAIL_MINOR:** Last state that, until now, was not widely used. It has been added for future use when a better distinction between errors will be done. Minor failure should not force the complete stop of the system but they should be noticed and reported to the user in some way. Examples of minor failure can be the malfunction of a redundant sensors, a temporary outgoing from normal working condition of a peripheral or the loss of some packets during a transmission.

```
1  int main(){
2
3      // variables initialization
4      FSM_active_state = SYS_INIT;
5
6      // infinite loop
7      while (1){
8
9          // switch between states
10         switch (FSM_active_state){
11
12         case SYS_INIT:
```

```
13              // wait initialization
14              // Stop here if errors
15
16              // if all ok
17              FSM_active_state = SYS_STARTUP;
18              break;
19
20          case SYS_STARTUP: {...}
21
22          case SYS_SAFE: {...}
23
24          case SYS_RUN_AUTO:
25              // Commander checks
26              Commander.run_auto_checks();
27
28              // if all ok
29              // pass control loop outputs to PWM module
30              FSM_active_state = SYS_STARTUP;
31
32              // else if fault major
33              FSM_active_state = SYS_FAIL_MAJOR;
34
35              // else if fault minor
36              FSM_active_state = SYS_FAIL_MINOR;
37              break;
38
39          case SYS_RUN_MANUAL: {...}
40
41          case SYS_FAIL_MAJOR:
42              FSM_active_state = SYS_SAFE;
43              break;
44
45          case SYS_FAIL_MINOR: {...}
46
47          default:
48              FSM_active_state = SYS_SAFE;
49              break;
50          }
51      }
52  }
```

Listing 2.1: Pseudo Code snippet of Finite State Machine implementation.

2.3 Commander

The Commander is another module that deals with the whole system. Together with the Finite State Machine, they are responsible of the system organization and status. They cooperate to ensure the achievement of the final objective, make the drone flies in the correct modality and with safety strategies. The FSM is more focused on the differentiation of the various tasks and the temporal sequence with which they are

executed. Instead, the Commander is responsible of the global status of the system and components that contribute at its operation, either hardware peripherals, either software modules.

In the PX4 Autopilot implementation, that is the main reference of our work, the Commander module is very wide and complex. Being an internal module, it's difficult to understand its behaviour, also because it is not well described as the user do not needs to directly deal with it. However, after having analyzed it in a deeper way, some information has been gathered. A first thing that is noticeable is that the Commander listens all the messages that are exchanged internally to the firmware. This is possible using shared channels of communications that are used in a master-slave strategy: the master writes on the channel, sending a message, then this message is available to be read from all the slaves that are listening that specific channel. The Commander is interested in a lot of information so it listens a great number of channels. Another important thing is how these data are used. PX4 Commander handles status information in three ways, depending on the system active status and the type of data. A first division is related to the state in which the drone is. If the drone is not flying, pre-flight checks are executed and, if no errors are detected, the firmware is allowed to arm the system. This means that the motors of the drone are allowed to start, so the drone can fly. Arming and disarming task are very important tasks handled by the Commander module. Once the drone is armed, it enters in a running state so different checks on the system should be performed. In fact, during this state pre-flight check are not useful, they are already been made and passed, so more lightweight checks are defined. When the drone flies, status checks should be executed in a very fast way because most of the cpu time is dedicated to the control system steps execution and, besides that, a failure has to be detected in the shortest possible time. For these reasons run-time checks are performed on flag-status boolean variables, the fastest variable type to be checked and also those that take up the lowest memory space. This is made to check the status of each needed component. Similar checks, using only a status bit, are used to monitoring over the most important physical variables related to the position of the drone, like roll, pitch, yaw angles or its altitude. These flags are set if there are inconsistent values of such measurements. All the functions described until now are partially implemented in our firmware because checks are performed in a similar way but the data are not acquired through shared channels, instead status flags are set directly by each software module using dedicated Commander methods.

Before entering in the description of the operation of our Commander module, is better to have an overview on how it is organized. We defined the Commander class, exploiting C++ functionalities, in the following way (Code Listing 2.2): attributes of this class are mainly boolean variables that represent all the components, either hardware parts attached to the drone, either software modules that must be monitored, and their relative information. All of them are stored in a single struct variable. This struct is defined with only boolean variables related to the component that they represent. Having a variable set to 0 means that the related component shows an error (the error can derive from multiple causes, examples are that the components is broken, or is not initialized, or simply it is not mounted). Instead, if the flag is set to 1, means that the component is correctly working. Then, there is a pointer variable linked to the

active FSM, in order to always know in which state the system is. This pointer should be carefully used because it can harm the behaviour of the FSM, which surely cause a malfunction of the drone. More precisely, this variable should only be read to get information and never be written. Last but not least, there is a lock variable used to protect multiple access to the attributes of the class. In addition to these attributes, there are a lot of methods used to deal with them and others to implement all the tasks that the Commander should provide. Regarding the attributes, there are two methods for each of them: a *getter* method to read its value, and a *setter* method to set the attribute value. So, after having defined how many flags are needed, then, two methods for each must be added. Same strategy is adopted for the FSM pointer variable, so a *getter* and a *setter* are implemented also for it. Notice that this methods, related to the FSM, modifies the pointer value, so the reference to a variable and not the actual value of it. Usage of *getter* and *setter* methods is recommended because it's a safe way to access, from external modules, to the internal attributes of a class. Moreover, it is easier to insert safety procedures, like adding a lock variable, because the code to be modified is tracked and isolated. Other methods implemented are the ones used by the *main* function, in the different states of the FSM. For each state there is a dedicated Commander method that performs checks on the system status. Also in this case, is not recommended to have a single check method to facilitate the errors handling during the active operation or the bugs identification during the development process. Until now, four check methods are implemented, for the respective FSM states: *Commander::check_init()* for the SYS_INIT state, *Commander::check_startup()* for the SYS_STARTUP state, *Commander::check_run_auto()* for the SYS_RUN_AUTO state, *Commander::check_run_manual()* for the SYS_RUN_MANUAL state. Each of them is implemented in such a way that the Commander performs the checks on the components that should correctly work in the current state in which the system is running. Another task under control of the Commander is the arming procedure, which is responsible of the system and motors enabling. For this purpose some methods are implemented.

To close the analysis of the Commander class is necessary to go deeper into the details either of the attributes, either of the methods.

Starting from the attributes we said that they are mainly boolean variables that represent the status of the components but we haven't specified how they are stored, who or what they represent and where they are modified. First of all, all flags variables are contained in a stuct defined outside of the class. It is partitioned in sub-level structs that represent group of similar components. There is a part dedicated to sensors, for each of them there is a flag that represents its status (working / not working) and, for the ones which a calibration is needed, a flag for the calibration status is also added. Then, there are parts dedicated to the communications and the control loop. The former contains a flag for the communications with the radio-controller and two separate flags for MavLink input and output communications. These flags represent the status of the respective channel. Instead, flags related to control loop tasks represents only the working state of the thread which executes the related task. If the thread is not terminated, the flag will be always set to 1. Lastly, there are flags for PWM handling and armed-disarmed state. The most important one of them is the *sys_armed_flag* that can be set to 1 only by calling the *Commander::arm()* method, which must set the flag

if the system is working correctly. Without this flag set, the system can't enable the PWM output ports, so the drone can't fly. Handling this flag is very critical because a bad value can lead to dangerous situations, not only for the drone but also for everything or everyone is around it.

Moving on to the analysis of Commander methods, let's start from *getter* and *setter*. They are not complex functions and their structure is the same for all of them. They have to deal with attributes, so they use a locks to gain access to the data. The *getter* methods tries to acquire the lock to read the target attribute, then, if this is available, they copy the data value in a temporary variable and, lastly, they free the lock. After all, they return the value just read. Instead, the *setter* methods tries to acquire the lock to write the target attribute and, if this is available, they modify that value. Then they end their execution. Functions for handling the pointer to the FSM active state are very similar, but they not use a lock because this variable is accessed only to be read.

On the contrary, the other group of methods works in a complete different way. They perform sequential checks on the attributes to monitor the current working conditions. If the checks don't show any errors, the methods return a positive value, instead a negative value is returned if a flag under analysis is not set to 1. The difference between the functions that perform checks is the group of flags that they test. For example, checks in the SYS_INIT state must be different to the checks performed in the SYS_RUN_MANUAL state because the system is doing extremely different things.

In future development of the firmware, if some FSM states will be added, also new Commander methods should be added and, same reasoning can be made with hardware and attributes.

```
1  class Commander
2  {
3      // attributes and methods visible outside the class
4      public:
5
6          // init methods
7          Commander();
8          void set_all_flags_to_zero();
9
10         // utility methods for FSM
11         bool check_init();
12         bool check_startup();
13         bool check_run_auto();
14         bool check_run_manual();
15
16         // methods for handling pointer to FSM state
17         void set_p_to_FSM_state(FSM_STATES*);
18         FSM_STATES get_main_FMS_state();
19
20         // arm - disarm handling
21         bool arm();
22         bool disarm();
23         bool is_armed()};
24
25         // getter - setter for attributes
26         bool get_flag_controller_active();
27         void set_flag_controller_active(bool);
28
```

```
29          bool get_flag_comm_joystick();
30          void set_flag_comm_joystick(bool);
31
32          ...
33
34
35      // attributes and methods visible only inside the class
36      private:
37
38          // struct containing all attribute flags
39          struct flags all_flags;
40
41          // lock for writing attributes
42          Read_Write_Lock lock_flags;
43
44          // pointer to FSM state
45          FSM_STATES* p_main_FSM_state;
46
47          ...
48
49  }
```

Listing 2.2: Pseudo Code snippet of Commander class implementation.

2.4 Command Line Interface

Normally, operating systems implement a command line interface in a shell window for interactive access to operating system functions or services. Those are used to processes text commands that are interpreted and transformed in a computer program execution. Today, many OS rely upon graphical user interfaces and menu-driven interactions because are more user-friendly; however, system with limited computing capability may still use a command line. Since our firmware runs on a low power micro controller, is impossible to implement a graphical user interface on it. So the CLI strategy is chosen. Its use may seem quite difficult to normal users because commands are given in a few characters that not always are easy to interpret. For this motivation a "help" option often provides a concise review of all commands and their description. In PX4 firmware there is a very long list of commands, with options, that can be launched to execute their relative task (link: `https://docs.px4.io/master/en/modules/modules_command.html`).

Possible uses of a CLI are extremely various because it's the developer of the code that assigns an operation to a sequence of characters. In commercial products, the commands must be related to the task that they perform, but, during the development phase, the programmer can exploit the full adaptability of the CLI. It helps a lot the bugs identification and the testing of the different parts of the software. Then, when the firmware reach a stable structure, only a sub-part of the commands are still active and the others are eliminated.

In our firmware the CLI modules works in a special way because it is not always active. First of all, being a module mainly needed in the development process, it can be deactivated by using a compilation flag variable. This is done for two reasons: the most important one is related to the hardware resources of the micro-controller. It has only two serial communication interfaces, so the command line can't be active together with MavLink and radio-controller communications because both of them make use of a serial communication channel. The other reason is that if the drone has to fly, the micro-controller cannot be linked to the computer with a USB cable, so the CLI can't be used. Moreover, also if the CLI is active it is not always available to the user. This is made for safety reasons, in fact, it is available only if the firmware is in the SYS_SAFE state. The *main()* function initializes a thread variable but does not start it until the SYS_SAFE state is reached. When is active, the CLI module works, as an infinite loop, into a thread parallel to the *main()* function. At each cycle of the loop, it awaits for a user input commands, which is then analyzed and compared with predefined sequences. The input sequence of characters is stored until an escape character is read, then, the whole string is assembled, so it is ready to be compared with some reference keywords. If a match is found, the respective task is executed, sometimes launching another thread. As can be seen in Code Listing 2.3, the selection of the task is made using a *SWITCH* statement.

```
1  void CLI()
2  {
3      // variables initialization
4      char cliBuffer[50];
5      __command command; // enum type
6
7
8      // infinite loop
9      while (1)
10     {
11         // active only in SYS_SAFE state
12         if(FMS_state() == SYS_SAFE){
13
14             // get user input, save it on buffer
15             handle_input(cliBuffer);
16
17             // get command from input char sequence
18             command = string_to_command(cliBuffer);
19
20             // switch command functions
21             switch (command){
22
23             case cmd_thread_info:
24                 threadinfo();
25                 break;
26
27             case cmd_help:
28                 help();
29                 break;
30
31             ...
32
```

```
33                default:
34                    // notify invalid input
35                    print("Type a valid command\n");
36                    break;
37
38                }
39            }
40        }
41    }
```

Listing 2.3: Command Line Interface main loop.

Different commands are identified using an *ENUM* type, defined in an external header file. There, are also listed all the functions related to the commands. This is made to centralize the information of available commands and to simplify the addition or removal of one of them. On the contrary, implementations of these functions are scattered in different files because some of them are quite long and complex. Knowing that is worthless showing all the implementation details of these functions, only a list of the available commands is presented, along to a description of each of them.

- **cmd_sys_info:** Display some hardware and software information, like the Mbed OS version, the firmware version and the total RAM available.
- **cmd_help:** Print a list of available commands with a brief description (Figure 2.3).

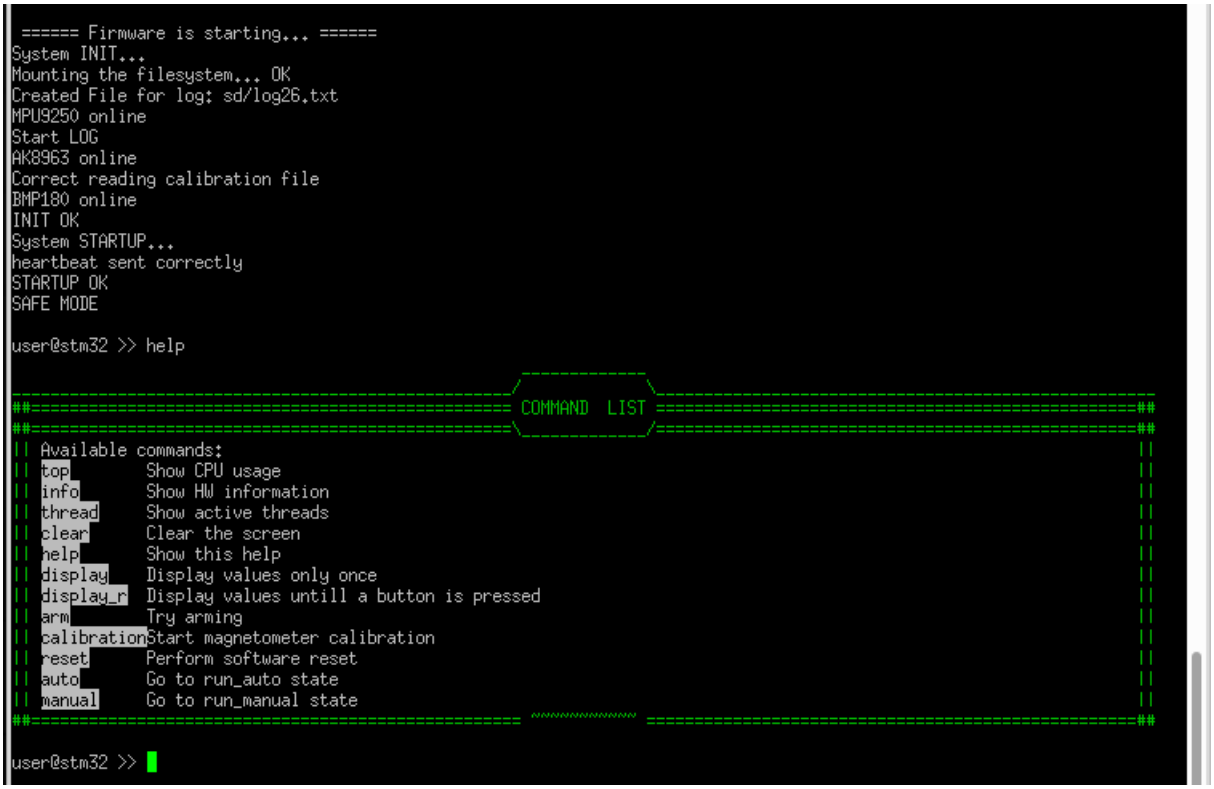

Figure 2.3: Prints of the firmware initialization phase and output of the "help" command.

- **cmd_clear:** Clear the output window.
- **cmd_return:** Do nothing, pressing only the *enter* keyboard button is considered a valid input.
- **cmd_invalid:** Print an error message. It is triggered by every not handled sequence of characters.
- **cmd_reset:** Perform a system reset. Same action that is executed when the physical reset button of the board is pressed.
- **cmd_run_man:** Force the system to switch to the SYS_RUN_MANUAL state of the main FSM.
- **cmd_run_auto:** Force the system to switch to the SYS_RUN_AUTO state of the main FSM.
- **cmd_arm_req:** Try to arming the system, calling the respective Commander function.
- **cmd_thread:** Show information about the active threads. (Figure 2.4)

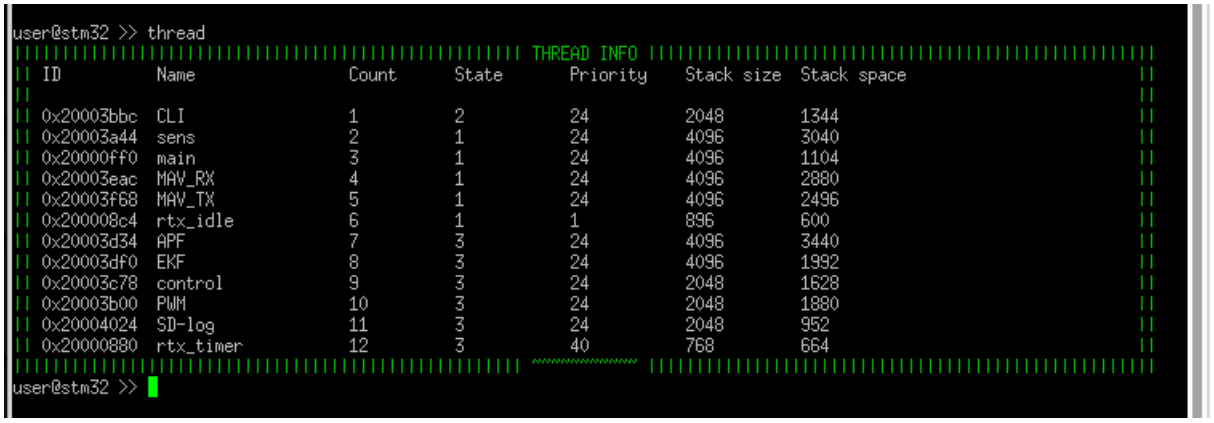

Figure 2.4: Output of the "thread info" command.

- **cmd_mag_calib:** Call a function that performs the calibration of the magnetometer and saves the result values.
- **cmd_top:** Show, with periodic timings, the real time usage of the CPU.
- **cmd_display:** Display, only once, the real-time values of some relevant data, such as the sensors readings or the roll, pitch, yaw angle values (Figure 2.5).
- **cmd_display_repeat:** Display, with periodic timings, the real-time values of some relevant data.

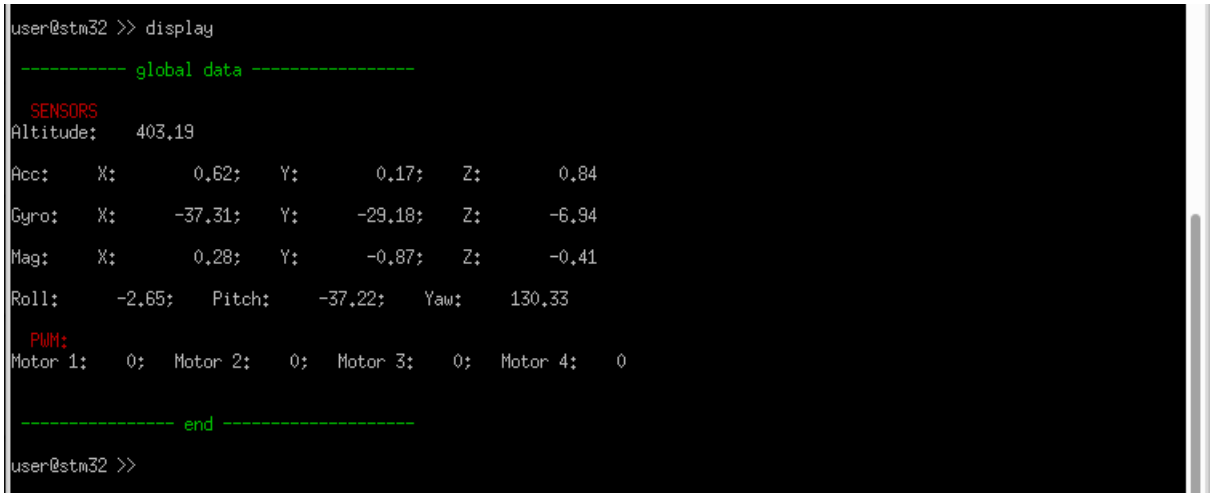

Figure 2.5: Output of the "display" command.

2.5 Communications

In an electro-mechanical object like the one under analysis, communications have a significant impact on the functioning of the system. Communications is a general term used to define all the tasks that contain some date exchange and they may differ in objective, implementation and importance. In the PX4 autopilot the *Middleware* is the layer that provides hardware integration and internal/external communications, using various modules. Instead, for our description, we want to make a clear separation between internal and external communication tasks. This division is made looking from the system perspective, thus, all streams of data that exit or enter in it are placed in the external type, instead, data streams that flow between hardware components are included in the internal type. In the following description, four communication interfaces are presented, two belonging to the former class, two to the latter.

MAVLINK

As in the PX4 autopilot, MavLink (*MavLink web page* (2021)) messages are employed to communicate with offboard systems like companion computers. This should be its final aim but, until now, that functionality is only partially implemented. This is possible thanks to the fact that MavLink supports different types of physical transport layers. In fact, it can be transmitted both through serial channels or through WiFi and Ethernet. The high level protocol is the same, so the switch from a physical layer to the other is quite easy to implement. In our firmware, MavLink messages are exchanged through serial channel but, in future versions, this will be changed and a more effective functionality will be developed to communicate with offboard systems.

MavLink is a very lightweight messaging protocol that defines a huge number of standard messages and microservices for exchanging data, however we use only a little part of them. It follows a modern hybrid publish-subscribe and point-to-point design pattern: data streams are published as topics, while configuration sub-protocols, such as the mission protocol or parameter protocol, are point-to-point with retransmission. It is very efficient and reliable standard. In fact, MavLink version 1 has just 8 bytes overhead per

packet, while MavLink version 2 has 14 bytes of overhead but is a much more secure. It provides methods for detecting packet drops, corruption, and for packet authentication. Because it doesn't require any additional framing it is very well suited for applications with limited communication bandwidth. To simplify the usage of MavLink protocol, a number of higher level APIs have been written, which typically provide implementations of the main microservices and simple/specific interfaces for sending commands and accessing vehicle information. Examples of them are MAVSDK, MAVROS and pymavlink. The MavLink microservices define higher-level protocols that MavLink systems can adopt in order to better inter-operate. They are used to exchange many types of data, including: parameters, missions, trajectories, images, other files. If the data can be far larger than can be fit into a single message, services will define how the data is split and re-assembled, and how to ensure that any lost data is re-transmitted. Some services also provide command acknowledgment and/or error reporting.

We use MavLink v2 that is a backward-compatible update to the MavLink protocol that has been designed to bring more flexibility and security to MavLink communications. The key features of MavLink v2 are:

- 24 bit message ID that allows over 16 million unique message definitions (MavLink v1 was limited to 256).
- Packet signing to authenticate that messages were sent by trusted systems.
- Message extensions that allows to add new fields to existing MavLink message definitions without breaking binary compatibility for receivers that have not updated.

MavLink messages, in the newest version, have the following field definition (Figure 2.6):

Figure 2.6: MavLink v2 packet fields.

1. **STX:** Protocol-specific start-of-text marker used to indicate the beginning of a new packet. Any system that does not understand protocol version will skip the packet. In MavLink v1 STX is 0XFE, whereas in MavLink v2 STX is 0xFD.
2. **LEN:** Indicates length of the following payload section. It is encoded in 1 Byte so the maximum value is 255.
3. **INC FLAGS:** Flags that must be understood for MavLink compatibility (implementation discards packet if it does not understand flag). Incompatibility flags are used to indicate features that a MavLink library must support in order to be able to handle the packet. This includes any feature that affects the packet format/ordering.

4. **CMP FLAGS:** Compatibility flags that are used to indicate features won't prevent a MavLink library from handling the packet (even if the feature is not understood). These flags can be ignored if not understood and the implementation can still handle packet.

5. **SEQ:** Denotes the packet sequence number. Used to detect packet loss. Components increment that value for each message sent. It is encoded into 1 Byte so takes values from 0 to 255. Once it reaches the maximum value, the sequence number is reset to 0.

6. **SYS ID:** ID of system sending the message. Used to differentiate systems on network. It is encoded into 1 Byte so takes values from 0 to 255. Note that the broadcast address 0 may not be used in this field as it is an invalid source address. The System ID 255 is typically allocated for ground station.

7. **COMP ID:** ID of component sending the message. Used to differentiate components in a system (e.g. autopilot and a camera). It is encoded into 1 Byte so takes values from 0 to 255. Specific values are defined for different hardware types.

8. **MSG ID:** ID of message type in payload. Used to decode data back into message object. It is encoded into 3 Bytes (low-middle-high bytes).

9. **PAYLOAD:** It is the actual message. It can contain up to 255 bytes depending on message type and contents.

10. **CHECKSUM:** It contains 2 bytes dedicated for checking the message correctness.

11. **SIGNATURE:** Optional Signature field of 13 bytes. It allows the authentication of the message and verifies that it originates from a trusted source. It is composed by three sub-field: the ID of link on which packet is sent (1 byte), timestamp (6 bytes) and the actual signature for the packet (6 bytes), based on the complete packet, timestamp, and secret key.

In our firmware, MavLink communications are handled using two threads: one for receiving messages and one to send them. Code Listing 2.4 shows how these threads are implemented. They initialize some utility variables then they enters in an infinite loop in which they work on messages. The sender thread must create, correctly fill and send the packets. The receiver must receive all packets, check they correctness and get the information from the payload using the MSG ID field to distinguish the type of data to be acquired. Along with normal messages, during the whole transmission time, a special message is exchanged to check the health of the system components. This is the *HEARTBEAT* message that is send to show that a system or component is present and responding. It contains also other useful fields as the vehicle or component type or the MavLink version active on the sender component. This message is send periodically and if it is not received for a defined time interval, the commander module is warned that the entering MavLink communications have a problem.

```
1
2  void mavlink_sender()
3  {
4      // variables initialization
5      uint8_t out_data[MAVLINK_MAX_PACKET_LEN];
6
7      uint8_t SYS_ID = 1;
8      uint8_t COMP_ID = 1;
9
10     // mavlink datastructure
11     mavlink_message_t msg;
12     mavlink_heartbeat_t heartbeat_msg;
13
14     // setup heartbeat message
15     mavlink_msg_heartbeat_encode(SYS_ID, COMP_ID, &msg, &
       heartbeat_msg);
16
17     ...
18
19     // infinite loop, start sending messages
20     while (1){
21
22         // send heartbeat periodically
23         if (timer_heart == 500ms){
24             serial_ch->write(heartbeat_msg, pck_len)
25         }
26
27         // fill other messages and send them
28         ...
29     }
30 }
31
32 void mavlink_receiver()
33 {
34     // variables initialization
35     uint8_t in_data[MAVLINK_MAX_PACKET_LEN];
36
37     // mavlink datastructure
38     mavlink_message_t msg;
39     mavlink_status_t status;
40
41     ...
42
43     // infinite loop, start collecting messages
44     while (1){
45
46         // check if something is arrived
47         if((byte_recived = serial_ch->read(in_data,
       MAVLINK_MAX_PACKET_LEN)) > 0){
48
49             // check if message is correct
50             if(mavlink_parse_char(MAVLINK_COMM_0, in_data, &msg, &
       status))
51             {
52                 // switch on message type
```

```
53                  switch (msg.msgid)
54                  {
55                  // heartbeat for communication status
56                  case MAVLINK_MSG_ID_HEARTBEAT:
57
58                  // notify commander that mavlink
59                  // communication is active
60                  main_commander->set_flag_comm_mavlink_rx(true);
61
62                  break;
63
64                  // other messages
65                  case MAVLINK_MSG_ID_SET_POSITION_TARGET_LOCAL_NED:
66
67                  // decode message and get useful data
68                  ...
69                  break;
70
71                  ...
72
73                  default:
74                  // error, message not decoded
75                  break;
76              }
77          }
78      }
79  }
```

Listing 2.4: MavLink receiver and sender threads implementation.

SBUS

The second protocol used for external communications is the SBUS protocol (*SBUS repository by bolderflight* (2021)), with which we can send manual commands to motors through a radio-commander. It has a digital bus format, it is an excellent means of receiving pilot commands from a transmitter and an SBUS capable receiver, often employed to send commands to servos. Unlike PWM, SBUS uses a bus architecture where a single serial line can be connected with up to 16 servos with each receiving a unique command. This property is very suitable for our application because it only uses a single pin to command more servos, so additional micro-controller pins are freed for other uses.

The SBUS protocol uses an inverted serial logic with a baud rate of 100000, 8 data bits, even parity, and 2 stop bits. The SBUS packet is 25 bytes long consisting of:

- **SBUS header:** Fixed field of 1 byte, its value must be 0x0F.
- **Data:** 22 bytes of data divided in 16 servo channels, 11 bits each.
- **Status byte:** Special byte divided in digital on/off channels, failsafe bit and frame lost bit.
- **SBUS footer:** Fixed field of 1 byte, its value must can be 0x00 or 0x04.

The integration of this functionality into the firmware is made by means a library, which is implemented to extract the useful data from the raw stream of bytes received through the serial channel. This library provides a C++ class that handles all the information internally, it provides as outputs the raw data and then it can divide them in channels. The main function is only responsible to instantiate an object of that class, specifying the pins needed for the serial communication interface, and call the available methods to get data.

SPI

Within the structure of the drone, others communication protocols are used to exchange data between hardware peripherals. One of them is the SPI protocol (Serial Peripheral Interface, (*Wikipedia: SPI* (2021))), used to read and write data on the SD card. It is a synchronous serial communication interface used for short-distance communication between electronics devices. SPI devices communicate in full duplex mode using a master-slave architecture, usually with a single master device that originates the clock for reading and writing data. In our system the master device is the micro-controller and the single slave is the SD card module. The SPI bus specifies four logic signals:

1. **SCLK:** Serial Clock (output from master).
2. **CS /SS:** Chip/Slave Select (often active low, output from master to indicate that data is being sent).
3. **MOSI:** Master Out Slave In (data output from master). MOSI on a master connects to MOSI on a slave.
4. **MISO:** Master In Slave Out (data output from slave). MISO on a master connects to MISO on a slave.

The data transmission works bidirectionally, timed by the clock signal from the master device. To begin communication, the bus master configures the clock, using a frequency supported by the slave device, typically up to a few MHz. Then, the master selects the slave device with the logic level 0 on the select line. Every slave on the bus that has not been activated using its chip select line must disregard the input clock and should not drive MISO. During each SPI clock cycle, a full-duplex data transmission occurs. The master sends a bit on the MOSI line and the slave reads it, while the slave sends a bit on the MISO line and the master reads it. This sequence is maintained even when only one-directional data transfer is intended. Transmissions often consist of eight-bit words. However, other word-sizes are also common, for example, sixteen-bit words.

I2C

Last analyzed protocol is the I2C protocol (*Wikipedia: I2C* (2021)), used to read data from sensors. It is a synchronous, multi-controller/multi-target, packet switched, single-ended, serial communication bus, widely used for attaching lower-speed peripherals to processors and micro-controllers in short-distance, intra-board communication. A

particular strength of I2C is the capability of a micro-controller to control a network of device chips, through addressing, with just two general-purpose I/O pins. Many other bus technologies used in similar applications, such as the SPI protocol previously described, require more pins and signals to connect multiple devices. I2C uses only two bidirectional lines: serial data line and serial clock line. The number of nodes which can exist on a given I2C bus is limited by the address space, which, in the I2C standard design, is 7-bit wide, but rarely, also a 10-bit extension is used. Data exchange is handled by means of transactions. I2C protocol defines fixed types of transactions, each of which begins with a START and ends with a STOP. Three basic transactions are:

- Single message where a controller (master) writes data to a target (slave).
- Single message where a controller (master) reads data from a target (slave).
- Combined format, where a controller (master) issues at least two reads or writes to one or more targets (slaves). Each read or write begins with a START and the target (slave) address but it isn't ended by a STOP so that the targets know that the next message is part of the same transaction.

2.6 Data and Synchronization

Another important discussion must be focused on the data that populate the firmware. In this chapter, all the previous analysis has been focused on modules and how they work, but it is worthwhile to provide a description of how data are handled and, specially, how they are exchanged between threads. This is a particularly sensitive issue due to the nature of the application, which is a real-time application. Data shall be available as fast as possible to the whole firmware that must guarantee the correctness of them. Must be noticed that data are useful, at the same way, either during the run-time phase either after that, for subsequent analyses. Thus, a suitable strategy has to be designed to take care of them and satisfy all the requirements needed to develop a well implemented firmware.

The following paragraphs are dedicated to the description of how data are stored during the run time phase, how they are saved in a non-volatile memory and how the correctness of them is granted in our firmware.

The first thing to say about data handling is how they are collected and stored to be utilized in the working phase of the drone. This is quite easy to explain because the fundamental idea is that all of them are stored in a single object. This is done to minimize the effort to maintain a lot of reference pointer for the different variables. Using a single reference, all the information are available through the same starting point. Moreover, if this is a global object (with global we refer to the visibility property of the variable in the firmware), it is accessible from all the threads that are running in the program. The object under analysis, used as a container of all the data, is an instance of a special class, the *GlobalData* class, implemented only for this purpose. This

class, as shown in Code Listing 2.5, has only private attributes and public methods. An attribute corresponds to the data to be stored, groped in a structure variable, while the others are some lock variables used to protect the access to the data. The structure variable has a particular definition because it is generated exclusively for this hardware and software. It is composed by sub-parts that identifies a specific part of the system: there is a structure used to store data read from the sensors, others to store inputs and outputs of the control loop blocks and another one to store the PWM signals sent to the motors. Then, there are methods that are the only ones visible to the outside of the class, to guarantee the correct modification of the shared data. For each protected variable, or a group of them, is available a dedicated method to read it and another to modify it. Locks are used to protect the access to the variables only when they are modified (a better description of the locks behaviour is made in the next paragraph). Beyond these, the class provides other two helpful methods used to display the data on the command line and to store them on the SD card.

```
class GlobalData
{
    public:
        // constructor
        GlobalData();

        // methods
        void display_on_CLI();

        void write_on_SD(FILE_PATH*);

        // getter - setter for data
        struct_sensors_data read_sensor();
        void write_sensor(struct_sensors_data);

        struct_pwm_data read_pwm();
        void write_pwm(struct_pwm_data);

        ...

    private:
        // struct containing all data
        data_t all_data;

        // locks
        Read_Write_Lock lock_sensor;
        Read_Write_Lock lock_pwm;

        ...
}
```

Listing 2.5: Pseudo Code snippet of GlobalData class implementation.

The locks used in the *GlobalData* class are instances of the *Read_Write_Lock* class, that is designed to simplify the usage of locks in a special way. From the literature, readers–writer locks (or single-writer locks, or multi-reader locks) are synchronization primitive that allow concurrent access for read-only operations, while write operations

require exclusive access. This means that multiple threads can read the data in parallel but an exclusive lock is needed for writing or modifying data. When a writer is writing the data, all other writers or readers will be blocked until the writer is finished writing. This is exactly our case, where data must be accessed by a single thread at a time when modified to guarantee its correctness. Our implementation of this class (Code Listing 2.6) is constructed on top of a mutexes primitive, which the *rtos* API of MbedOS make available.

```
class Read_Write_Lock {

    public:
        // contructor
        Read_Write_Lock(){};

        void read_lock(){
            // if lock is locked (owner != 0)
            if ((int)this->lock.get_owner() != 0){
                // wait and then unlock (read is not blocking)
                this->lock.lock();
                this->lock.unlock();
            }
            // else do nothing
        };

        void read_unlock(){}; // do nothing, read is not blocking

        void write_lock(){this->lock.lock();}; // write is blocking

        void write_unlock(){this->lock.unlock();};

    private:
        rtos::Mutex lock;
}
```

Listing 2.6: Read_Write_Lock class implementation.

In the end, the last functionality that involves data is their saving on the SD card. This is done in a parallel thread that is activated only if the SD card is mounted and correctly initialized. During the initialization, is checked if a file system is present on the SD card and if not, a new one is created. If an old file system is found, the firmware looks for the log file with the highest indexation. Then, it creates a new file with the next index (if no file is found the index will be 1). These operations are done using the File System API of MbedOS.

If no errors occur, the log file is correctly created and the log task can be started. As a first information, a starting line is written with the labels of the data that will be put into the file. Then, the log file is periodically updated adding, every time, a new line containing the useful data. Along to the data, a reference time point is added to identify a correct timing and sequence of log.

```
void SD_log_loop(void){

    // update period
```

```
4       std::chrono::milliseconds log_step = 500ms;
5
6       // open log file in write mode
7       FILE* log_file = fopen(filename, "w+");
8
9       // write first line of file
10      fprintf(log_file, "TIME (ms), ALTITUDE, ...");
11
12      // get starting time point
13      Kernel::Clock::time_point start_log = Kernel::Clock::now();
14
15      // start infinite loop
16      while (1)
17      {
18          // get actual time
19          log_time = Kernel::Clock::now();
20
21          // write time point on file
22          fprintf(log_file, "%lld,", (log_time - start_log).count());
23
24          // write other data
25          global_data->write_on_SD(log_file);
26
27          // wait until next step
28          ThisThread::sleep_until(log_time + log_step);
29      }
30  }
```

Listing 2.7: Pseudo Code snippet of SD_Log thread function.

2.7 Control scheme

In this section a description of the control scheme implemented is made. The design of the control loop wasn't in the scope of the thesis, so, it is displayed entirely, but the analysis is reported with few technical details. The relevant aspect is how this control scheme is inserted into the firmware.

The overall designed structure is made up of several blocks as shown in Figure 2.7, thus, in the firmware, a module for each block is added. Obviously, the "plant" block doesn't have a respective module because it corresponds to the hardware, the real quadrotor. This block, used only for simulations, includes the mathematical model of the quadrotor dynamics and of the actuators mounted on it. The resulting model provides a relationship between inputs, the PWM signals provided to the motor, and outputs, the forces and moments generated, the linear and angular velocities and positions of the drone. Also the "sensors" block is particular for the same reason but, conversely, it has a dedicated software module. To better simulate the behaviour of the real sensors, in this block, input signals are corrupted by adding a bias and noise on these. Other blocks are the Extended Kalman Filter, used as an observer, a trajectory planner and the actual controller. These are integrated in the firmware using three modules that have a similar

flow of execution. They are started by the main thread, they begin with an initialization phase followed by an infinite loop. The cycles of the loops are executed periodically, according to the time step used during the simulations. This is important because the system has very fast dynamics, it is a real-time system, so, if the loop is executed with too slow rate, it can lead to unstable conditions. The code that has to be inserted in these loops is handled in a very effective way thanks to the "Embedded coder" toolbox of Matlab/Simulink. Using this toolbox it is possible to export the Simulink model to external projects as a C code. The generated code is created so that the execution of a single step of the model is entirely included in a function. Inputs and outputs of the model are handled using specific structure variables. So, after having generate the C code from Simulink, the modules developed in the firmware have only to call these functions periodically and manage the data flow (inputs and outputs) correctly.

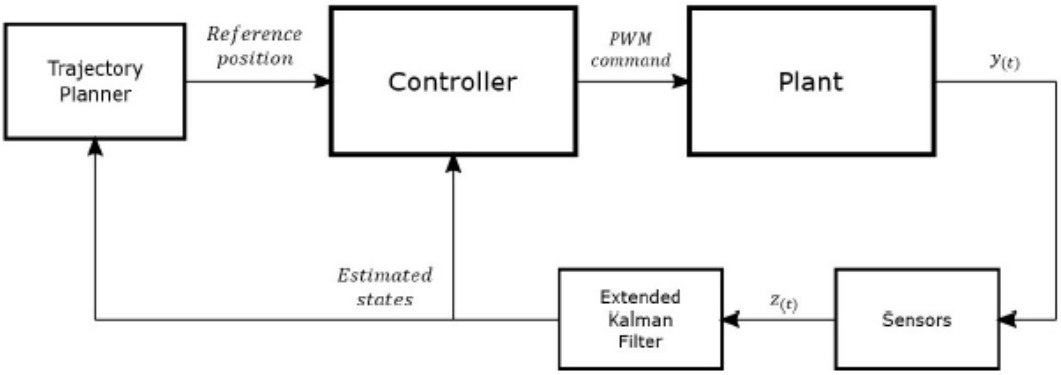

Figure 2.7: Control loop block diagram.

Now let's move the description to the designed control loop blocks, starting from the controller. It is a cascade controller, as shown in Figure 2.8, in which there are an outer loop and an inner loop. The former acts as a simple position controller and it is related to the slow dynamics of the system. It tracks the desired position in the X-Y plane exploiting a PD (proportional-derivative) controller structure (Figure 2.9 when the Ki parameter is set to 0, so without Integral compensation). Instead, the latter, is related to the fast dynamics of the system. It deals with the attitude and the altitude of the drone, controlling the angular velocities of the rotors. In particular, it generates torque commands having as input a desired attitude and a thrust commands having as input the desired altitude and vertical velocity. This is a complete PID (proportional-integral-derivative) controller (Figure 2.9). PID controllers are widely-used linear controllers that use information about the error, between a given setpoint or reference signal and the measured output of the system, at a certain time. The command action is generated by three contributions: one proportional to the errors (proportional action – P), one to its derivative (Derivative action – D) and one to its integral (Integral action – I).

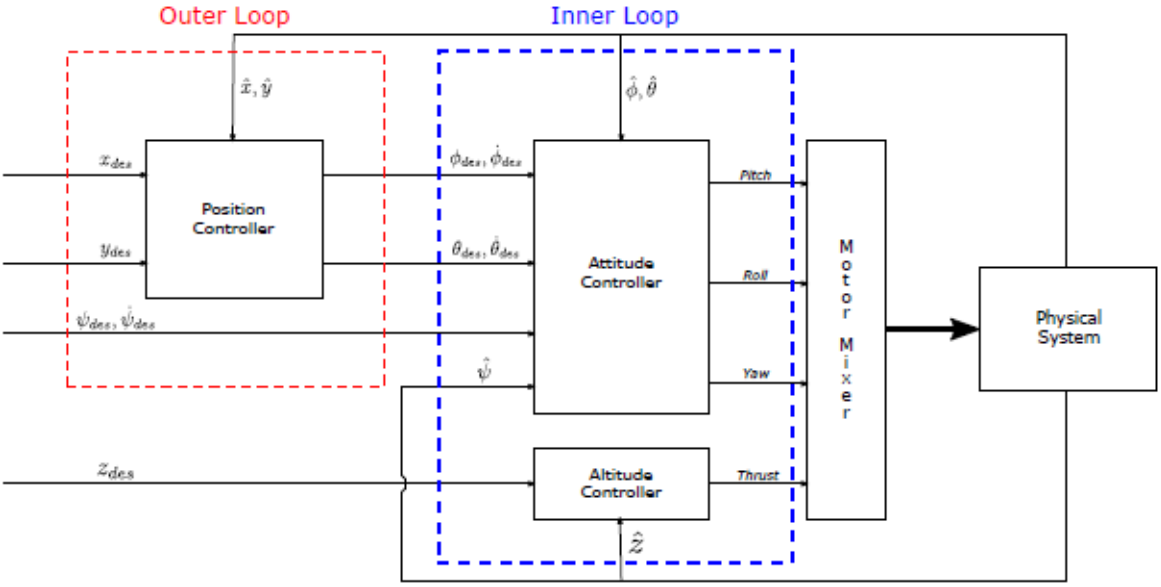

Figure 2.8: Controller block diagram.

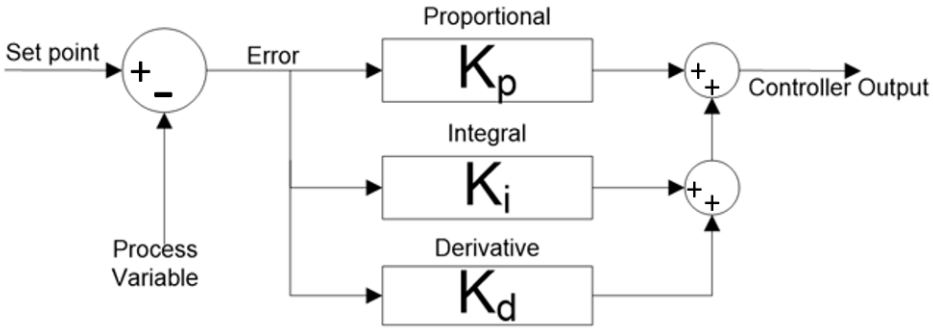

Figure 2.9: PID block command.

The trajectory planner block contains an algorithm that generates a time-dependent path given a set of waypoints with their Times of Arrival (ToA). It imposes a trapezoidal linear velocity profile on the segment linking two consecutive waypoints and it computes, by integration, the quadrotor reference position as a function of time. The maximum acceleration can be set being a tunable parameter, while the maximum velocity is determined such that the maneuver can be completed respecting the assigned ToA. The algorithm receives as input arguments the waypoints, in a specific inertial frame, with their ToA and the number of points between two consecutive waypoints. It gives as output the reference trajectory in the used reference frame and the reference time vector. With this algorithm is possible to generate a lot of flight profiles such as takeoff maneuver, landing maneuver and '8 shape' trajectories.

Lastly, there is the Extended Kalman Filter (EKF) block. The EKF is a near-optimal, nonlinear observer; it is the nonlinear version of the Kalman filter ((*Wikipedia: Kalman filter* (2021))). Kalman filtering is an algorithm that provides estimates of some unknown variables given the measurements observed over time with relatively simple calculations, hence it requires small computational power. Thanks to that it has been used in various real-time applications and also in our. In this application, the EKF is used to filter the noise coming from the sensors, to fuse redundant signals and as an estimator of the states of the system.

Chapter 3

Hardware

3.1 Technologies review

Previous chapter was centered on software and its implementation, instead now the focus of the analysis will be on the hardware. As already explained in Section 1.1, the lower-lever division of rotor-wings UAVs is between single-rotor and multi-rotors. The latter type is much more employed due to its better stability. In fact with a single rotor is difficult to maintain a fixed and stable condition. Multi-rotor and especially Quad-rotor can take advantage of more propellers to lift and stabilize the assets at the same time. Redundant lift sources can also give increased margin of safety. Paying special attention on quad-rotor configurations, we can differentiate 3 configurations of them as shown in figure 3.1.

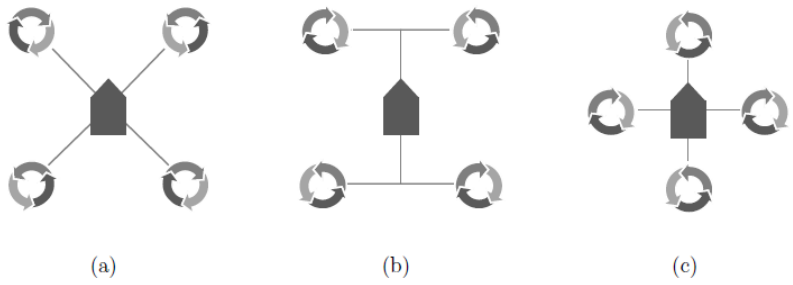

Figure 3.1: Quad-rotors 'X'(a), 'H'(b) and '+'(c) configurations. Arrows represent propeller direction.

Each type of configuration offers advantages and disadvantages. The 'X' configuration is the most commonly utilized since it is simple to construct, is symmetrical and robust. The 'H' configuration is very similar, they differs only in the linkage structure between body and motors. A disadvantage of these configuration is an increase in modelling and control complexity. In contrast to the others, the '+' configuration is the simplest on that aspect. However, the disadvantage is that moving forward this type of drones will leave the vehicle with lesser stability in the longitudinal rotation motion. Also

performing other maneuvers is easier with 'X' or 'H' configurations due to the use of 2 motors simultaneously. For example, to make the quad-copter turns left or right, you have to speeding up two motors that are diagonally across from each other and slowing down the other two. Similarly, if you wants to rotate forward, you have to speeds up the back two motors and slows down the front two. Instead, altitude is controlled by speeding up or slowing down all motors at the same time, for all configurations.

For our project a 'X' configuration of quad-copter is chosen (Figure 1.6.)

After having analyzed the global architecture of drones, is now reported a detailed list of useful components that should be mounted on an aerial drone. Some of them are essential whereas other are optional.

- **Micro-controller board:** This is the 'brain' of the drone. A micro-controller is a small computer mounted on an integrated circuit chip. It contains a processor cores along with memory and programmable input/output peripherals to interact with other components. A non volatile memory (typically a flash memory) is in charge of storing the compiled code that will run on the processor when it is turned on, instead, a smaller volatile RAM memory is used by the program to execute the code and storing variables during the run-time phase. Micro-controllers are designed for embedded applications and are often used in automatically controlled devices. They usually contain from several to dozens of general purpose input/output pins (GPIO). GPIO pins are software configurable to either an input or an output state. When GPIO pins are configured to an input state, they are typically used to read sensors or external signals. Configured to the output state, GPIO pins can drive external devices such as LEDs or motors, often indirectly, through external power electronics.

- **Body structure:** A solid and stable structure is always needed to mount all the other components. The structure is what give a shape to the drone, so it is very variable due to its target task.

- **Inertial Measurement Unit (IMU):** The IMU is the most crucial sensor for determining the UAVs state. The IMU uses a combination of accelerometers and gyroscopes to determine the craft's velocity, orientation and gravitational pull. Interpreting its output allows us to represent the craft in 3D space with 6 degrees of freedom (DOF), using both the 3D position of the centre of mass and the 3 flight dynamic angles (yaw, pitch, roll). Generally, UAVs carry MEMS-based IMUs. MEMS stands for Micro Electro-Mechanical Systems, which allows for the IMU to be much lighter and smaller than others. However, MEMS based technology is more susceptible to noise due to their dimensions. Additionally of note, IMUs can represent the UAV with variety of DOF depending on the sensors it utilizes. For instance, the most basic IMU units contain an accelerometer and a gyroscope and represent 6 DOF systems. However, because both rely on gravitational forces, neither can represent the actual yaw angle. To be able to correctly calculate the current yaw angle, a magnetometer (compass) must be used, and this combination of sensors would allow for 9 DOF. Sometimes, also the altimeter is considered a part of the IMU.

- **Altimeter:** Altimeters are devices designed to calculate the altitude of an aircraft. This reference level may be Above Ground Level (AGL) or Above Sea Level (ASL). Different types of altimeter use different technologies, methods and physical quantities to calculate that. There are four main types of altimeter: barometric, radio, GNSS and laser altimeter.

  Barometric altimeter can calculate the altitude level by comparing the atmospheric pressure at the current height with the pressure at sea level (a fixed value). In general terms, the greater the altitude the lower the pressure. However, air pressure is affected not only by altitude but also by temperature (air pressure may also fluctuate due to changes in the weather which may cause changes in both pressure and temperature). These variables must be taken into account in order to obtain an accurate reading from a barometric altimeter.

  Radio or sonar altimeter send pulses of sound with known speeds at a surface. By measuring the time it takes for the pulse to return to the sensor, it determines its distance from the surface. Because sounds waves are relatively slow, sonar is used mostly in indoor environments or during landing maneuvers, where the UAV is very close to the surfaces it is trying to detect. Additionally, these sensors are particularly susceptible to interference. For instance, if multiple UAVs utilizing sonar sensors are operating in close proximity, they will 'hear' each other's pulses, resulting in inaccurate measurements. These sensors are relatively cheap, and lightweight, so there are specific use cases where they are suitable.

  Laser or lidar altimeters work in a similar way to radio altimeters, but they use electromagnetic waves instead of radio waves. Again, the time taken for the emitted signal to travel from the transmitter to the surface and back is measured. Once reflected, the beam of light is received and collected using a series of mirrors and lenses which focus the beam onto a photocell detector, which is sensitive to infrared light. Laser is unaffected by sunlight, and they are the fastest at measuring distance to surfaces of all the sensors. These sensors are particularly useful as the amount of light in the environment will not affect the measurements, but they can be used in both the day and night. However, they are very expensive.

  Global Navigation Satellite System (GNSS) receivers can also determine altitude by trilateration with four or more satellites. To make this calculation, the time of flight of radio waves from a known point to another is again used. Examples of that sensors are GPS and Galileo. Altitude calculated using GNSS is, however, not accurate such as a barometric altimeter. Errors in height calculation using GNSS are typically in the region of 5 meters, so they cannot be used for precise and indoor applications.

- **Vision sensors:** They are used to get a qualitative representation of the world around the UAV and create useful maps of it. In recent years such type of sensors are widely used for artificial intelligence or deep learning tasks. The majority of them are cameras. Cameras is the general term to describe a lot of different kind of technologies that takes in light and converts it to a digital representation of the world. This task can be performed in very different ways.

  Monocular cameras are traditional cameras with one lens and one sensor capturing photons. **Stereo cameras** have two or more lenses to allow them to capture 3D images by simulating human binocular vision. Range of stereo systems is limited by the baseline of the cameras. **RGB-D camera** capture traditional RGB color images, but augment each pixel with a "depth" of data to get an information about the distance of objects.

- **Infrared scanners:** They create heat maps by detecting infrared energy and converting it into electronic signals that we can represent as a picture of a given environment. These are especially useful in night-time or cloudy conditions.

- **Signal receiver:** This peripheral is a simple antenna used to get signals from a joystick or radio-controller actuated by the user of the UAV. It is an optional module because the majority of drones can flies in an autonomous way, but it's also important to have an alternative way to control them for safety reasons. At the moment there is a wide choice of receivers and it's possible to choose the most suitable one according to the task that it should fulfill. Some aspects to consider when choosing are: the power consumption, the protocol used for transmissions of data and the coverage area (maximum distance from user).

- **Motors:** Motors are one of the most important part of the drone because they actuate the rotors. Different size and technologies can be used but the most common are brush-less electrical motors. They have small size and are very powerful, moreover they can be combined with specific gearboxes to get even more efficiency. They must be light-weight specially in multi-rotors where there is a single motor for each rotor. In some cases, but very rarely, thermal motor are used instead of electrical ones.

- **Battery:** This is a needed component for electrical UAVs because is the power supply source. Its task is to make all electronic components work with the correct energy. Rarely is substituted by a fuel tank when thermal motor are employed.

- **Storage:** A non volatile memory is often presents on drones for different purpose. Obviously a data storage module is mainly used to store data, as the name hints. Then all the saved data can be used as you want. An example of usage is to save some significant data during the flight, as a black box for getting information if a fail occurs. Or simpler, to analyze these data after a test flight.

Figure 3.2: Main hardware components of a quad-rotor UAV.

3.2 Project setup

The project target architecture is a small size quad-rotor UAV, the one showed in Figure 1.6. Lateral and longitudinal dimension are 0.20 *m*, instead the height is 0.15 *m*. The weight is approximately 0.5 *Kg*. At the same time, other two autonomous vehicles (Figure 3.3) are under development: another quad-rotor UAV and an UGV (unmanned ground vehicles). The firmware analyzed in Chapter 2 is suitable for all quad-rotor drones that use the same components/sensors, therefore it can be easily employed onto drone 'a' in Figure 3.3. On the contrary is not so easy to adapt it for a UGV because their behaviour and tasks are completely different.

From now on, we will consider only the quad-rotor hardware architecture of drone in Figure 1.6 for a more detailed explanation.

(a) (b)

Figure 3.3: Alternative quad-rotor architecture (a) and UGV (b)

The chosen **micro-controller** is a ST NUCLEO-32 board (*STM web page* (2021)): the ST NUCLEO-L432KC (Figure 3.4). That is a ultra low power board with small size and few (32) GPIO pins. It has an ARM cortex-M4 CPU with 64KB of RAM and 256KB of flash memory. On the board are also mounted a reset button, a single user-programmable LED and a micro USB connector used either to load programs onto the flash memory, either for serial communication during the working phase. The board power supply can be provided from the USB connectors or directly on dedicated pins (the 'VIN' and 'GND' pins, see Figure 3.4). From that, the board make available two output voltage supply (3.3V and 5V) through others pins (the '5V' pin and the '3V3' pin, see Figure 3.4). All the others GPIO pins can be used for various functionalities that are grouped in two categories: communications and signals input/output. ADC, DAC and PWM modules are in charge of deal with external signal. ADC and DAC receive them from peripherals instead PWM send them out, for example, to move an electric motor.

Communications can be handled with some protocols and, based on these, different pins should be used. The pins must be configured in the correct way setting some internal addressable registers of the micro-controller. ST NUCLEO-L432KC board has 14 communication interfaces but only 32 pins, so some of them are shared by different tasks. It is important to correctly setup the pinout because, at runtime, a pin can perform only one task at a time. Most common protocol for communications are: UART (Universal Asynchronous Receiver-Transmitter, the simplest serial communication), CAN o CAN-BUS (Controller Area Network), SPI (Serial Peripheral Interface) and I2C (Inter Integrated Circuit). In our project we made use of almost all of them.

The UART, or simply serial communication, is employed, through USB when it is connected, to exchange data with the main computer. We make use of two UART channels. The former is the one that make the Command Line Interface (Section 2.4) works. The latter is used to exchange MavLink messages (section 2.5). At runtime, so When USB is not connected to the main computer, the first channel is connected with the radio-controller receiver to get user input commands.

I2C protocol, as the name says, is appropriate to connect integrated circuits, thus we used it to attach sensors with the micro-controller. All of them are connected to the same I2C channel and are identified by different addresses. That make possible to attach a lot of sensors to the same pins (until the address value allows).

Lastly, SPI interface is used to exchange data between the micro-controller and the storage unit, the SD card, because it has an higher throughput than I2C. This protocol is better when great amount of data have to be read or written.

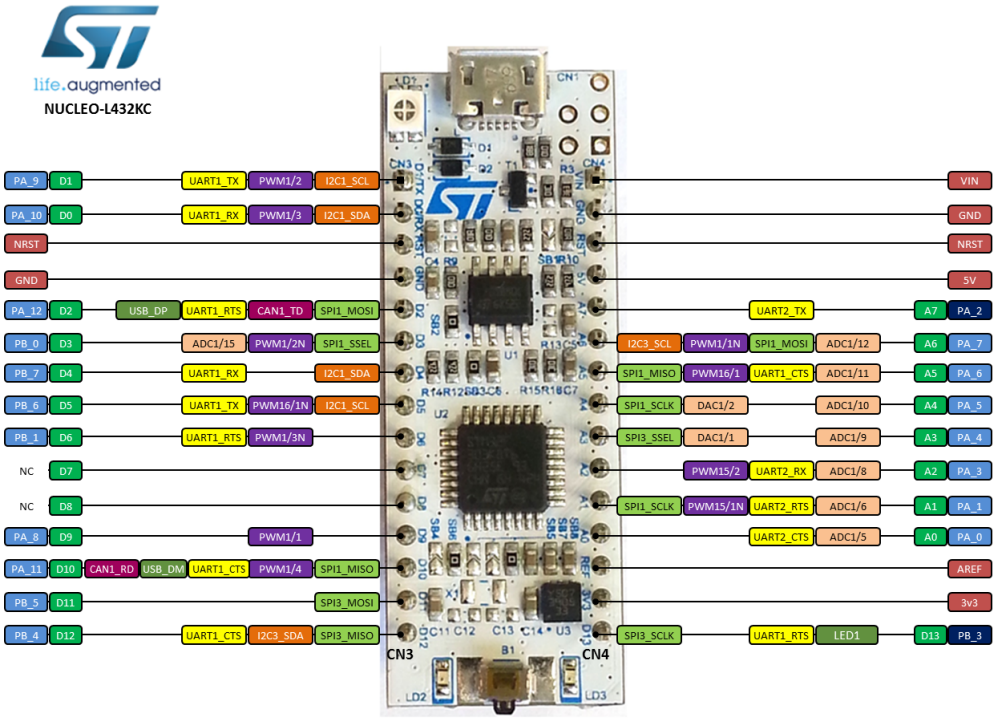

Figure 3.4: STM32 Nucleo-32 development board with STM32L432KC MCU

Going on, adopted sensor boards are presented. As IMU sensor a MPU-9250 is used (Figure 3.5). It is a 9 DOF MEMS motion tracking device. It is very small (3x3x1 *mm*) and low power consumption. The MPU-9250 is a system in package that combines two chips: the MPU-6500, which contains a 3-axis gyroscope and a 3-axis accelerometer; the AK8963, a 3-axis digital compass. For ease of use it is mounted on a board that provide 10 bigger pins that are directly linked with the pins of the sensors by means electronic tracks. We use only four of them: two for the voltage supply, two for I2C communications.

At the same way the pins of BMP-180 are used. BMP-180 is an high precision barometer. It is useful for getting the altitude of the drone measuring the pressure and transform it in the desired measurement. Also if it is very precise, it is not optimal for indoor application because pressure variation is noticeable only when there is a big

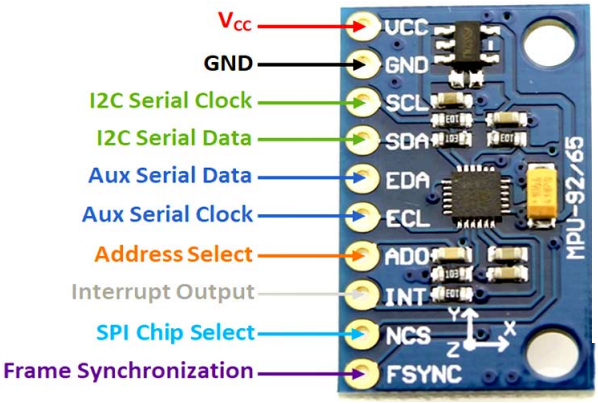

Figure 3.5: MPU-9250

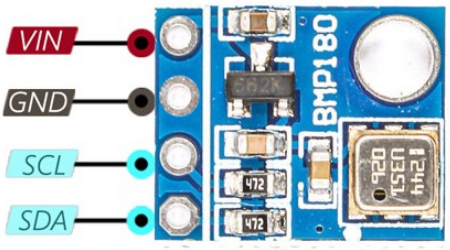

Figure 3.6: BMP-180

height variation. The formula for conversion from pressure to altitude is also influenced by temperature, so it is another factor that can add noise to the measurements.

For next development steps, the possibility to add more sensors will be evaluated. Sensors that can be added are sonar and/or laser sensors for more precise altitude measurement and cameras for other functionalities.

Another component that has been mounted on the drone is a FrSky (*FrSky web page* (2021)) antenna (Figure 3.7). This antenna receive signals from a radio-controller/joystick that is handled by the user. Linked to the antenna there is an electronic box (the X8R package by FrSky) that convert those signals so that they can be correctly interpreted by the micro-controller. Those outputs are read on the SBUS port through a simple serial communication interface. SBUS protocol is convenient because, using a single signal, it allows to send 16 values/channels. These channels are representative of buttons or commands sent by the radio-controller.

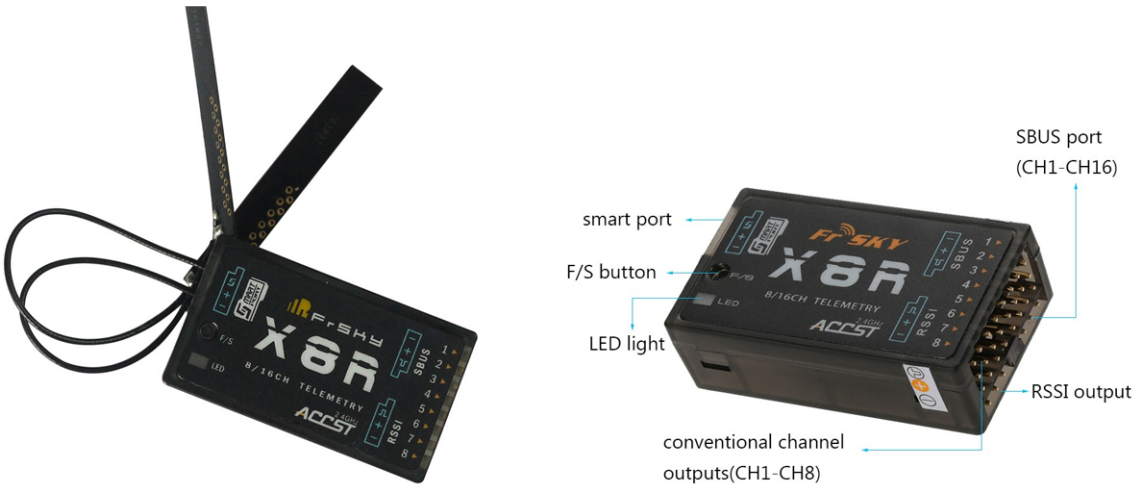

Figure 3.7: FrSky antenna and X8R package interfaces overview

An optional component is the storage module. In our project a MicroSD card is used to store runtime data as log book of the flight. The micro-controller can't access directly to the SD card so it is necessary to add an intermediate component: the DFRobot MicroSD card module (Figure 3.8). This module has an adapter for MicroSD cards and allows to read or write on those using the SPI protocol. It has six pins, two for the voltage supply and all the others are needed for the communication protocol.

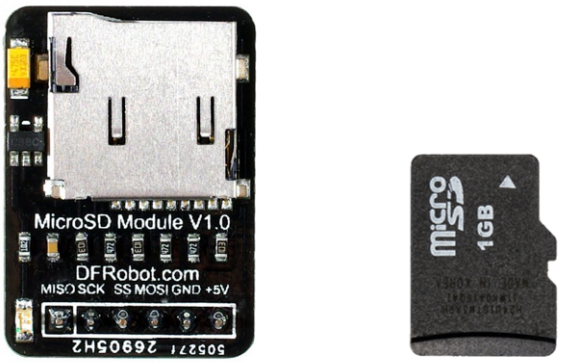

Figure 3.8: DFRobot MicroSD card module and MicroSD card example

To close the chapter about the hardware description, a concise presentation of motors is made. Brushless electrical motors by Tarrot (MT1806 kv2300) and propellers 5035 by T-Motor are used. There are four of them, each controlled individually through PWM signals sent by the micro-controller and generated by the control law.

Figure 3.9: Brushless electrical motors by Tarrot and propellers by T-motor

Chapter 4

Conclusions

Day after day, usage of drones is growing fast and they are becoming part of our normal life. Predictions on aerial traffic are astonishing. Today, there are nearly 50,000 aircraft operations in a day across the United States, while daily drone flights will number in the millions. For this reason, also research on drones is growing and this thesis project is a tangible proof of this situation.

The principal objective of the work was to develop a firmware for a quad-rotor UAV. A firmware is composed by a lot of functionalities and the time needed to develop them in the best way can be enormous. For example, the PX4 autopilot project started in 2009, the first release was published in 2012 and, nowadays, is still under development. Knowing that, our work was limited to the development of a basic, but stable, version of a personal firmware that has to deal with pre-defined hardware. The work started by studying other reference implementations that can be found on the literature and on the web. As can be expected, at least for UAV dedicated firmware, the central component is the flight controller block. It is usually composed by sub-modules that, cooperating, make the system work as desired. In our firmware these modules are the Finite State Machine, the Commander and the Controller. The FSM is responsible of the flow of execution of the code. This means that it should grant that the system is always in the correct state of execution and transitions between different states must be coherent with the desired design. The sequence of states where the firmware passes must be defined and analyzed with special attention because it is fundamental that all the peripherals are initialized and checked to be working. For this motivation, before the system enters in a running state, it has to pass through two initialization states. When the system is allowed to run, it enters in a running state that can be either the manual state or the automatic state. From here no transition has to occur unless a failure is detected, which leads to a transition to a failure state. For this purpose (failure detection) the Commander module is activated. It works parallel to the FSM module and it acts only when called by the main function. This is done each time a functionality has to be tested, so either in the initialization phase, either in the run time phase where it is called periodically. The other purpose of the Commander, also related to checks, is the arm/disarm functionality. Thanks to this, the firmware can allow the movement of the motors and so the drone can fly. At this point is needed the intervention of the controller module whose task is to produce output PWM signals that are sent to the motors. The controller module is composed by three parts (the path planner, the extended Kalman

filter and the actual controller) that cooperate to generate suitable commands for the motors to fulfil the imposed task.

Around this central structure just described, there are other modules that help the operation. The most important concern the data and communication handling. They are deeply linked because data are exchanged by means communication channels, which can be internal of the system or external. The data flow has a fixed path and, in a certain way, it can be considered a cyclic path. The starting point are sensors whose acquire information from the real world and transform such reads in relevant information for the micro-controller. So, from sensors, data flow towards the micro-controller where are stored in a global and shared instance of a class. Then, these information are used by the controller components to perform calculation and generate output PWM signals, which are passed to the motors. Finally, motors act as commanded and the drone moves accordingly. Now the cycle restarts and sensors have to acquire new data to get information about the new position of the drone. This cycle is active during the whole operative phase of the drone. At the same time, in parallel threads, data are stored on the SD card and are exchanged with an external device through a MavLink communication channel.

In the end, last module implemented is the Command Line Interface module. This is needed by the user to interact with the drone in real-time, when it is in an idle state. Through the CLI, the user can get information about the status of the drone or test some functionalities. This module is very useful to the developer of the system because he can insert special commands that are linked to function that execute a defined task. During the development phase, these commands can be changed or modified quite easily, so it is a very handy way to test different components of the firmware. Instead, when the firmware reaches a stable status, the list of commands is fixed and the user has access only to these.

It must not be forgotten which is the final aim of a firmware, that is to put it on a real hardware. The software development process is always followed by an integration step where the software meets the hardware. In our project, this integration test is done every time a module, related to an hardware component, has to be added. Particular attention has been focused on sensors readings, because they are fundamental peripherals that work as interface between the world and the drone.After sensors, the attention switched to the SD card and, lastly, to the radio-controller and the relative receiver. Only after having checked the correctness of single modules, a global integration has been done. This final step wasn't so easy to be completed because all the modules interact with each other and perform a real-time debug is very tricky and complex.

List of Figures

Listings

List of Abbreviations

AGL: Above Ground Level

AIAA: American Institute of Aeronautics and Astronautics

APF: Artificial Potential Field

ARM: Advanced RISC Machine

ASL: Above Sea Level

CAN: Controller Area Network

CPU: Central Processing Unit

DFM: Deterministic Finite State Machine

DOF: Degrees Of Freedom

EASA: European Aviation Safety Agency

EKF: Extended Kalman Filter

FAT: File Allocation Table

FSM: Finite State Machine

GND: GrouND level

GNSS: Global Navigation Satellite System

GPIO: General Purpose Input/Output

KB: Kilo-Bytes

IMU: Inertial Measurement Unit

IoT: Internet of Things

I2C: Inter Integrated Circuit

LED: Light Emitting Diode

MAV: Micro Aerial Vehicle

MCU: MicroController Unit

MEMS: Micro Electro-Mechanical System

NFM: Non-deterministic Finite State Machine

OS: Operating System

RAM: Random Access Memory

RGB: Red-Green-Blue

ROA: Remotely Operated Aircraft

RPV: Remotely Piloted Vehicle

RTOS: Real Time Operating System

SD: Secure Digital

SPI: Serial Peripheral Interface

UAS: Unmanned Aircraft System

UART: Universal Asynchronous Receiver-Transmitter

UAV: Unmanned Aerial Vehicle

UGV: Unmanned Ground Vehicle

USB: Universal Serial Bus

VIN: Voltage INput

References

ArduPilot web page (2021). https://ardupilot.org/.

Capello, E., de Pierrepont Franzetti, I. D. D. M., Carminati, D. and Scanavino, M. (2020), 'Model-in-the-loop testing of control systems and path planner algorithms for quadrotor uavs', *International Conference on Unmanned Aircraft Systems (ICUAS)* .

FrSky web page (2021). https://www.frsky-rc.com/.

Gomaa, H. (2016), *Real-Time Software Design for Embedded Systems*, Cambridge: Cambridge University Press.

Matlab web page (2021). https://mathworks.com/products/matlab.html.

MavLink web page (2021). https://mavlink.io/en/.

MbedOS web page (2021). https://os.mbed.com/docs/mbed-os/v6.15/introduction/index.html.

Nonami, K., Kendoul, F., Suzuki, S., Wang, W. and Nakazawa, D. (2010), *Autonomous Flying Robots*, Springer.

Platformio web page (2021). https://platformio.org/.

PX4 autopilot user guide (2021). https://docs.px4.io/master/en/.

SBUS repository by bolderflight (2021). https://github.com/bolderflight/sbus.

Shakhatreh, H., Sawalmeh, A. H., Al-Fuqaha, A., Dou, Z., Almaita, E., Khalil, I., Othman, N. S., Khreishah, A. and Guizani, M. (2019), 'Unmanned aerial vehicles (uavs): A survey on civil applications and key research challenges', *IEEE Access* **7**, 48572–48634.

STM web page (2021). https://www.st.com/en/evaluation-tools/nucleo-l432kc.html.

Valavanis, K. and Vachtsevanos, G. J. (2015), *Handbook of Unmanned Aerial Vehicles*, Springer.

Visual Studio web page (2021). https://code.visualstudio.com/.

Wikipedia: I2C (2021). https://en.wikipedia.org/wiki/I2C.

Wikipedia: Kalman filter (2021). `https://en.wikipedia.org/wiki/Kalman_filter`.

Wikipedia: SPI (2021). `https://en.wikipedia.org/wiki/Serial_Peripheral_Interface`.

Yanushevsky, R. (2011), *Guidance of Unmanned Aerial Vehicles*, CRC Press.

Zimmerman, N. M. (2016), *Flight Control and Hardware Design of Multi-Rotor Systems*, Master Thesis in Electrical and Computer Engineering, Marquette University.