Politecnico di Milano
Scuola di Ingegneria Industriale e dell'Informazione
Corso di Laurea Magistrale in Computer Science and Engineering

A flight mode management and setpoint generation system for UAV flight control

Advisor:	Prof. Marco LOVERA
Co-Advisor:	Dr. Simone PANZA

Thesis by:
Davide COPETA SACCOMANI
Matr. 882957

Academic Year 2018–2019

Alla mia famiglia, a Elena.

Acknowledgments

Vorrei ringraziare il Professor Marco Lovera, relatore di questa tesi, per la professionalità e la disponibilità dimostrata durante il periodo di sviluppo e stesura di questo lavoro.
Un sentito ringraziamento va a Simone che mi ha seguito per l'intero percorso di progettazione e implementazione, i suoi consigli e il suo supporto mi hanno aiutato alla realizzazione di questo progetto.
Ringrazio tutti i ragazzi del laboratorio con cui ho condiviso questi mesi.
Un sincero ringraziamento va ai miei genitori, che hanno sempre sostenuto ogni mia scelta e hanno reso possibile questo traguardo.
Un ringraziamento speciale va a Elena che mi ha sempre incoraggiato anche nei momenti più difficili.
Grazie a tutti i miei amici per questi anni trascorsi insieme.

Abstract

The evolution of Unmanned Aerial Vehicles (UAVs), commonly known as drones, in the last years has led to a real revolution in several areas, from surveillance to military use, from photography to entertainment. The application fields in which they can be employed are always expanding.
A research area of interest involves the flight control and in particular the flight modes that UAVs can perform. As a consequence of the numerous existing modes, it is necessary to manage the transitions between the different flight modes and to facilitate the integration of new ones.
The purpose of the thesis is twofold: to design and implement a multi-mode setpoint generation system for UAV flight control based on the PX4 flight stack, managing the flight modes and generating the setpoint based on the current flight mode enabled; to modify the custom flight controller to allow the management of the main flight modes: manual, altitude and position. This custom flight controller has been developed at the Aerospace System and Control Laboratory (ASCL) at Politecnico of Milan for education and research activities.
In order to achieve the first goal, a setpoint mapper system is designed and implemented in Simulink. From the Simulink model, automatic code generation is then exploited to integrate the system into a custom module in the PX4 firmware.
To fulfill the second goal, the flight controller interface is modified in order to run the proper control law requested by the autopilot.

Sommario

L'evoluzione degli Unmanned Aerial Vehicles (UAV), comunemente noti come droni, negli ultimi anni ha portato ad una vera e propria rivoluzione in diversi settori, dalla sorveglianza agli usi militari, dalla fotografia all'intrattenimento.
I campi in cui possono essere impiegati sono in continua espansione.
Un'area di ricerca di interesse riguarda il controllo di volo ed in particolare le differenti modalità che gli UAV possono eseguire. Come conseguenza alle numerose modalità esistenti, è nata la necessità di gestire le transizioni tra le modalità di volo e facilitarne l'integrazione di nuove.
Lo scopo della tesi è duplice: progettare e implementare un sistema per la generazione di setpoint sulla base della modalità di volo abilitata; modificare il controllore di volo per consentire la gestione delle principali modalità di volo: manuale, quota e posizione. Questo controllore è stato sviluppato presso l'Aerospace System and Control Laboratory (ASCL) del Politecnico di Milano per attività didattiche e di ricerca. Al fine di raggiungere il primo obiettivo, è stato progettato e implementato un sistema di mappatura dei setpoint in Simulink. Dal modello Simulink, è stata poi sfruttata la generazione automatica di codice per integrare il sistema nel firmware PX4. Per raggiungere il secondo obiettivo, sono state apportate modifiche all'interfaccia del controllore di volo al fine di poter eseguire la legge di controllo richiesta dall'autopilota.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The evolution of Unmanned Aerial Vehicles (UAVs), commonly known as drones, in the last years has led to a real revolution in several areas, from surveillance to military use, from photography to entertainment. The application fields in which they can be employed are always expanding.
A research area of interest involves the flight control and in particular the flight modes that UAVs can perform. In order to manage the different flight modes, UAVs are equipped with an autopilot software.
One of the most popular autopilots for UAVs is PX4, a widely used software both in academic and industrial contexts. From an architectural point of view, PX4 firmware is composed by modules, each of them part of the system dedicated to compute specific tasks. The most relevant modules are the commander and the position controller.
The commander module is the core of the firmware and it is a very complex module that manages many tasks, starting from what the vehicle should do to security checks. It manages also the transitions between the different flight modes, rejecting a specific flight mode if some conditions are not satisfied.
The position controller module has a twofold role: it runs the position control law if it is requested by the autopilot and acts as a setpoint generator based on the enabled flight mode. In fact, based on the radio controller input, it generates the setpoint depending on the current flight mode enabled.

Objectives

The main objective this thesis aims to fulfill is the design and implementation of a system to manage the generation of setpoints based on the flight mode of interest and on the commands provided from the RC controller and the ground control station.
This custom component, besides mapping the setpoints, uses the information coming from external commands to handle the management of the transitions

between flight modes. Another objective this thesis aims to achive is to modify the interface of the flight controller so that, based on the information related to the currently active flight mode, the controller shall run the proper control laws.

Approach

This work starts from the analysis of the commander module to understand how the current PX4 state machine has been implemented. The commander state machine manages the switching between the flight modes and sets its state based on the flight mode if the transition to that particular mode is not rejected.
The analysis of the state machine has been performed in terms of how the current state is associated with respect to the enabled flight mode and how the transitions between the states are managed. Subsequently, focus has been put on how the position controller module manages the setpoint mapping.
A setpoint mapper system has been designed; it is composed of a state machine which is in charge of managing transitions between modes, and a setpoint computation system which generates the appropriate setpoint based on the current mode. This system has been implemented in Simulink; subsequently, code has been automatically generated from the model by means of the Embedded Coder. The last part of the work consists in the firmware integration of the generated code: the model subscribes and publishes the uORB topics in order to communicate with the other modules of the PX4 firmware.

Thesis structure

The thesis is structured in the following way:

- Chapter 2 will present what is a multicopter, how it works and how it is composed in terms of eletrical and mechanical components. Besides, it will outline the adopted formalism, the flight modes and the implementation of the PX4 firmware.

- Chapter 3 will present a detailed description of the setpoint mapper system in terms of design and implementation. It will describe the architectural choices, the software used to develop the mapper state machine and the setpoint computation block, focusing on the mapping laws: for each flight mode, it will describe how the setpoint are mapped and computed.

- Chapter 4 is dedicated to the testing part of the setpoint mapper system. A formal testing procedure has been defined for the mapper state machine using a specific verifier tool. Furthermore, it will report three experimental results of in-flight testing.

- Chapter 5 will analyze the implemented system and will present some advantages of the proposed solution, together with the description of some possibilities to improve the system in future works.

Chapter 2

Multirotors and PX4

In this chapter we are going to introduce multicopters, how they work and how they are composed in terms of electrical and mechanical components. Besides, focus will be put on the PX4 autopilot and flight modes.

2.1 Introduction to multirotors and flight control

A multirotor or multicopter is an aerial vehicle with two or more rotors the motion of which is controlled by varying the relative speed of each rotor to change the thrust and torque produced by each. Multicopters have very fast attitude dynamics and require an on-board computer, the Flight Control Unit (FCU) for flight stabilization: the flight controller controls the angular speed of each motor, a task which would be overwhelming for human pilot.
The simplest type of multicopter is the quadcopter (or quadrotor) configuration, which is composed by four motors, four Electronic Speed Controls (ESCs) to regulate the speed of the motors and four propellers to generate thrust. This is the most common UAV configuration due to its simplicity (Figure 2.1).

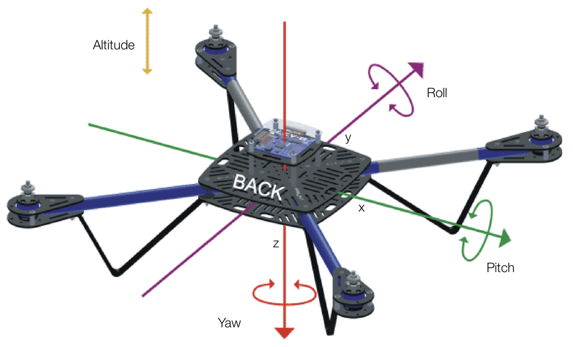


Figure 2.1: Multicopter in quadrotor configuration.

Each motor/propeller is spinning in the opposite direction from the two motors on either side of it. As a result two motors rotate in clock-wise direction while the other two in the counterclock-wise direction in order to balance the generated torque.
A quadcopter can control its roll and pitch attitude by speeding up two motors on one side and slowing down the other two. For example if the quadrotor wants to roll right, it would speed up motors on the left side of the frame and slow down the two on the right (Figure 2.2). Instead if the quadrotor wants to roll left, it would speed up motors on the right side of the frame and slow down the two on the left (Figure 2.3). Regarding the yaw, the copter can turn left or right by speeding up

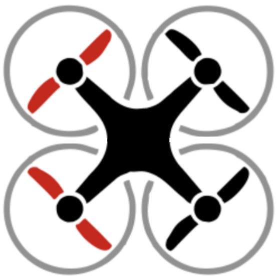


Figure 2.2: Right roll rotation.

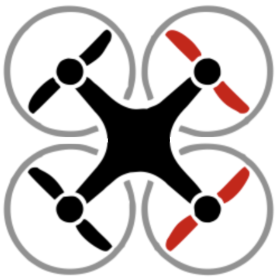


Figure 2.3: Left roll rotation.

two motors that are diagonally across from each other and slowing down the other two (Figure 2.4 and Figure 2.5). Altitude is controlled by speeding up or slowing

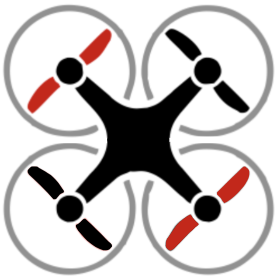


Figure 2.4: Left yaw rotation.

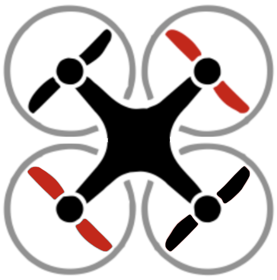


Figure 2.5: Right yaw rotation.

down all motors at the same time. There are many types of multirotor such as bi-copter, tricopter, quadcopter, pentacopter, hexacopter and octacopter, depending the number of rotors they have. Out of all these copters hexa and octa copters are considered to be the most stable drones and quadrotors have dynamically simple and strong structure, so they are used widely for experimental purposes.

2.1.1 Flight controller

The component that allows the flight is the flight controller, the real core of the multicopters. The Flight Control Unit is dedicated to control RPM (revolutions per minute) of each motor since the pilot is not able to control all motors at the same time to keep the quadrotor balanced. Selecting the right board depends on the physical constraints of the vehicle and the applications it is meant for. The Aerospace System and Control Laboratory (ASCL) at Politecnico di Milano has developed a number of multi-motor platforms dedicated to research and educational activity; for the experimental validation part of this thesis, the ANT-1 prototype has been considered (Figure 2.6).

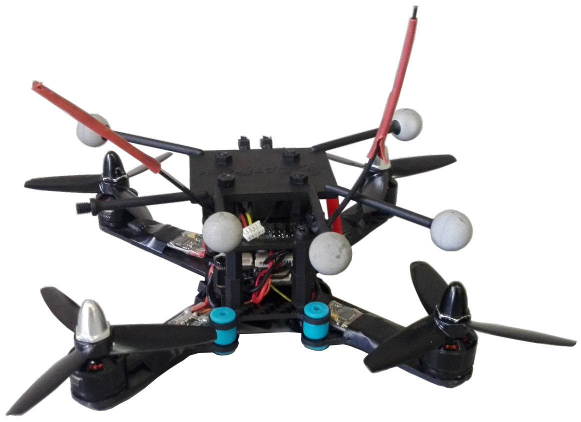


Figure 2.6: ANT-1.

The relevant parameters are reported in Table 2.1.

Variable	Value
Frame Config.	X
Propellers	Gemfan Bullnose 3055 3 blade
Arm lenght b	80 mm
Take-off weight m	230 g
Motors	QAV1306-3100kV brushless
ESC	ZTW Spider series 18A
Battery	Turnigy nano-tech 950mAh LIPO

Table 2.1: Main quadrotor parameters.

The quadrotor depicted in Figure 2.6 is a lightweight custom model with a distance of 160mm between opposite rotor axes and an overall take-off weight of about 230g.
Another UAV platform used to educational and research purposes, is the tiltrotor (Figure 2.7). The tiltrotor is a kind of quadrotor that has an over-actuated structure, in particular it has tilting rotor capabilities. Thanks to four servo-motors,

each arm with motor plus propeller group can be tilted in order to produce not only a vertical force, but also translational forces. This capability lets the tiltrotor to reach a full position/attitude decoupling: for example, it is able to hover keeping non-null roll/pitch angles. This kind of platform paves the way for more complex maneuvers and more operational scenarios, but also to more sophisticated control strategies able to exploit the eight actuators to fully control the six degrees of freedom of a rigid body in space [1]. A tiltrotor prototype has been designed and realized in Micheli [2].

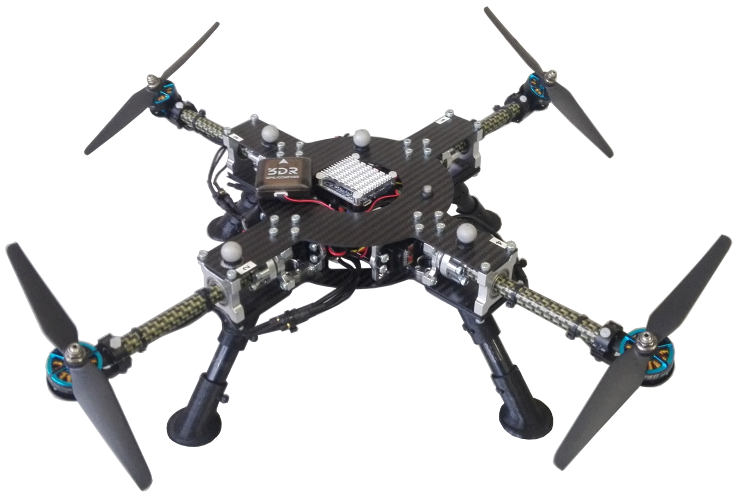


Figure 2.7: Tiltrotor.

The main parameters are reported in Table 2.2.

Dimensions	37x37x12 cm
Mass	1523 g
FCU	PixHawk mini

Table 2.2: Tiltrotor features.

The eletronic board on ANT-1 is the PixFalcon (board produced by Holybro, Figure 2.8), which features sensors, such as 3-axes accelerometer, 3-axes gyroscope, magnetometer and barometer, this board is based on the hardware standard[3].

Figure 2.8: PixFalcon FCU.

The features are listed in Table 2.3.

Dimensions	38x42x12 mm
Weight	15.8 g
Main Chipset	STM32F427
CPU	Cortex M4 core 168 MHz
RAM	256 KB SRAM (L1)

Table 2.3: PixFalcon FCU features.

The first version of the board was developed in 2008 by a team of students at the ETH university in Zurich. During the same time the team also created the MAVLink protocol, used for serial communication between Flight Control Unit and companion computer, the PX4 firmware, which is the firmware supported by PixFalcon and QGroundcontrol, the software used for PixFalcon configuration and real time information.

2.1.2 Companion computer

The companion computer (Figure 2.9) is the part of the drone system used to interface and communicate with PX4 on a PixFalcon using the MAVlink protocol. It enables a broad range of functionality such as the possibility to execute processes that require heavy CPU load. During the flight sessions arena, the companion computer is used to receive drone position information from the Ground Station, which is connected to a motion capture system, and to send the information received to the FCU through the serial communication.

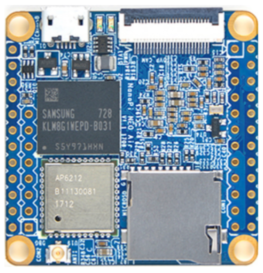


Figure 2.9: NanoPi NEO Air.

In Table 2.4 are reported the main parameters.

Name CPU	Quad-core Cortex-A7 1.2 GHz
RAM	512 MB
Wireless	2.4 GHz 802.11 b/g/n
Dimensions	40 x 40 mm
Weight	7.9 g
Power	5V-2A
Price	28 $

Table 2.4: NanoPi NEO Air features.

2.1.3 Radio controller

A RC system has a remote control unit that is used by the pilot to command the drone. The remote controller has physical controls that can be used to specify vehicle movements and to enable flight modes. On telemetry-enabled RC systems, the remote control unit can also receive and display information from the vehicle (e.g battery level, flight mode).
The remote control unit contains a radio module that communicates with a radio module on the vehicle. The radio module is connected to the flight controller and, based on the vehicle state and the current flight modes, drives the motor and the actuators in a proper way.
A remote controller used to control aircraft must have at least 4 channels (roll, pitch, yaw, thrust). An 8 or 16 channels transmitter has additional channels used to control other mechanisms or activate different flight modes. A popular remote controller for UAVs is FRrSky Taranis X9D. There are many possible configurations for the control sticks, switches, etc. The most common layout is "Mode" numbers. Mode 1 and Mode 2 differ only in the placement of the throttle: Mode 1 has the throttle on the left side (Figure 2.10) while in Mode 2 is placed on the right. A different configuration of the channels can be set at firmware level.

Figure 2.10: Taranis X9D radio.

2.1.4 Position measurement system

Normally in order to retrieve a 3D position in the space, the system requires GPS information, in the Aerospace System and Control laboratory at Politecnico of Milan there is a Motion Capture System (Mo-Cap) to achieve this task. The system is composed by 12 Infra-Red (IR) sensitive Opti-Track [4] cameras with incorporate IR flood lights. The cameras (Figure 2.11) are mounted on a closed arena and they are fixed at calibrated positions and orientation so that the measurement subject is within the field of view of multiple cameras. Thanks to markers (Figure 2.12) sensitive to infrared light mounted on top of the drone, it is possible to make it trackable and define its position into 3D space. To control the motion capture system, the Motive software (installed in Ground Station) is used. Motive is a software platform designed to control motion capture systems for various tracking applications. It not only allows the user to calibrate and configure the system, but it also provides interfaces for both capturing and processing of 3D data. The frequency with which the position information are sent to the drone (cameras rate), can be changed from a minimum of 30 Hz up to a maximum of 240 Hz through Motive software. The accuracy of the position estimated by the drone depends on the cameras frequency selected.

Figure 2.11: Infrared camera.

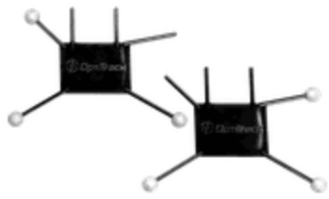


Figure 2.12: Infrared markers.

2.1.5 Ground Station

The Ground Station is the part of the system dedicated to send and retrieve information from the quadcopter during a flight session. The main task of the GS is to read the position information coming from motion capture system and send them to the drone, besides it is also possible to define a trajectory to be followed using specific waypoints and view telemetry data in real time.
The GS architecture is divided into two different OS's: the main OS is Windows 10 [5] in which Motive is installed. The second OS is Linux OS (more precisely Ubuntu 16.04 [6]) which is installed on a virtual machine and is used to execute ROS [7]. The GS architectural division was necessarily done because Motive is a Windows software while ROS integrates better in a Linux environment.

2.2 Formalism

In this section, some fundamental mathematical formalism used in the remainder of the thesis will be given.

2.2.1 Reference frames and axes

The motion of a body in space is described thanks to reference systems that need to be properly chosen. Many conventions are known in the literature. The NED convention is chosen for the inertial frame. The Earth fixed frame, assumed to be an inertial frame, is defined as $\mathcal{F}_E$ =$\{O_E, N, E, D\}$, where O_E is a point on the Earth surface. The N-axis and the E-axis are chosen to point respectively North and East and the D-axis completes the right-hand rule pointing downward. The second frame to be defined is the body frame $\mathcal{F}_B$ =$\{O_B, X_B, Y_B, Z_V\}$, where O_B corresponds to the body center of mass. The X-body axis lies in the plane of symmetry and generally points forward. The Y-body axis points in the right direction, normal to the plane of symmetry, and the Z-body axis points down.

2.2.2 Rotation matrices and Euler angles

A rotation matrix is a matrix $\in$ SO(3)

$$\mathrm{SO}(3) := \big\{R \in \mathbb{R}^{3\times 3} : RR^T = I_3,\, \det(R) = 1\big\}. \tag{2.1}$$

A rotation matrix could express three different meanings:

- the orientation of a frame with respect to another frame;
- the transformation that relates the coordinates of a point in two different frames;
- the rotation of a vector in a coordinate frame.

In order to rotate a vector around different axes many times, it is useful to define the rotations around a coordinate axis and in particular:

$$R_x(\alpha) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\alpha) & \sin(\alpha) \\ 0 & -\sin(\alpha) & \cos(\alpha) \end{bmatrix} \tag{2.2}$$

$$R_y(\beta) = \begin{bmatrix} \cos(\beta) & 0 & -\sin(\beta) \\ 0 & 1 & 0 \\ \sin(\beta) & 0 & \cos(\beta) \end{bmatrix} \tag{2.3}$$

$$R_z(\gamma) = \begin{bmatrix} \cos(\gamma) & \sin(\gamma) & 0 \\ -\sin(\gamma) & \cos(\gamma) & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{2.4}$$

The rotation direction is considered positive counterclockwise, following the right-hand rule. Since rotation matrices are orthonormal, the inverse rotation is obtained by transposing it. In fact:

$$R_x^{-1} = R_x^T \tag{2.5}$$
$$R_y^{-1} = R_y^T \tag{2.6}$$
$$R_z^{-1} = R_z^T. \tag{2.7}$$

Many consecutive rotations could be performed multiplying rotation matrices, noting that rotations performed in different order produce different results $(R_x(\alpha)R_y(\beta) \neq R_y(\beta)R_x(\alpha))$. When a vector or a frame is rotated an arbitrary number of times, the final attitude vector could be represented by a minimal representation, that consists in performing just three consecutive rotations.
A minimal representation is a parameterization of the attitude with respect to three parameters, called Euler angles $\Phi = [\phi \quad \theta \quad \psi]^T$ which represent the three angles of the rotations associated respectively with the matrices R_x, R_y, R_z, also called *roll, pitch, yaw.* The rotation from the earth frame to the body frame can be expressed in this form:

$$T_{BE}(\phi, \theta, \psi) = R_X(\phi)R_Y(\theta)R_Z(\psi), \tag{2.8}$$

where the subscripts B and E stand for "Body" and "Earth", respectively.
Matrix T_{BE} resolves an Earth-based vector to body axes. Expanding the indicated matrix multiplication, the T_{BE} matrix has these elements:

$$T_{BE}(\phi, \theta, \psi) = \begin{bmatrix} c_\theta c_\psi & c_\theta s_\psi & -s_\theta \\ s_\phi s_\theta c_\psi - c_\phi s_\psi & s_\phi s_\theta s_\psi + c_\phi c_{psi} & s_\phi c_\theta \\ c_\phi s_\theta c_\psi + s_\phi s_\psi & c_\phi s_\theta s_\psi - s_\phi c_{psi} & c_\phi c_\theta \end{bmatrix}, \tag{2.9}$$

where the matrix (2.9) reported a short notation has been reported, which is $c_\theta = \cos(\theta)$, $s_\theta = \sin(\theta)$ and $t_\theta = \tan(\theta)$.
On the other hand, it is also possible to obtain the T_{EB} (the rotation matrix from "Earth" to "Body") simply transposing T_{BE}, by obtaining:

$$T_{EB} = T_{BE}^T. \tag{2.10}$$

Quaternions

A quaternion is a four-dimensional representation of the orientation of a rigid body or a coordinate frame in three-dimensional space and are generally represented in the form:

$$q = \begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \end{bmatrix} \tag{2.11}$$

where q_0, q_1, q_2, q_3 are real numbers such that $q^T q = 1$. The term q_0 represents the scalar part of the quaternion, while $[q_1, q_2, q_3]^T$ constitutes the vectorial part. An alternative quaternion notation exists, in which the scalar term is located at the bottom of the quaternion (*i.e.*, $q = [q_1, q_2, q_3, q_0]^T$). In the remainder of this work, the first notation shall be employed (scalar term first). The quaternion elements generate the following coordinate transformation:

$$T_{BE} = \begin{bmatrix} q_0^2 + q_1^2 - q_2^2 - q_3^2 & 2(q_1q_2 + q_0q_3) & 2(q_1q_3 - q_0q_2) \\ 2(q_1q_2 - q_0q_3) & q_0^2 - q_1^2 + q_2^2 - q_3^2 & 2(q_2q_3 + q_0q_1) \\ 2(q_1q_3 + q_0q_2) & 2(q_2q_3 - q_0q_1) & q_0^2 - q_1^2 - q_2^2 + q_3^2 \end{bmatrix}. \tag{2.12}$$

The Euler angles representation of q_{BE} is defined by:

$$\phi = \tan^{-1}\left(\frac{T_{BE}(2,3)}{T_{BE}(3,3)}\right), \tag{2.13}$$

$$\theta = \sin^{-1}\big(T_{BE}(1,3)\big), \tag{2.14}$$

$$\psi = \tan^{-1}\left(\frac{T_{BE}(1,2)}{T_{BE}(1,1)}\right). \tag{2.15}$$

The quaternion conjugate, denoted by *, can be used to swap the relative frames described by an orientation. For example, the attitude of frame B relative to frame A can be represented by the quaternion q_{AB}, and its conjugate q_{AB}^* describes the orientation of frame A relative to frame B (q_{BA}). The conjugate of q_{AB} can be described by the following equation:

$$q_{AB}^* = q_{BA} = \begin{bmatrix} q_0 \\ -q_1 \\ -q_2 \\ -q_3 \end{bmatrix}. \tag{2.16}$$

The quaternion product, denoted by $\otimes$, can be used to define compound orientations. For example, for two orientations described by q_{AB} and q_{BC}, the compounded orientation q_{AC} can be obtained in this way:

$$q_{AC} = q_{BC} \otimes q_{AB}. \tag{2.17}$$

For two quaternions, a and b, the quaternion product can be determined using the Hamilton rule:

$$a \otimes b = \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} \otimes \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} a_0b_0 - a_1b_1 - a_2b_2 - a_3b_3 \\ a_0b_1 + a_1b_0 + a_2b_3 - a_3b_2 \\ a_0b_2 - a_1b_3 + a_2b_0 + a_3b_1 \\ a_0b_3 + a_1b_2 - a_2b_1 + a_3b_0 \end{bmatrix}. \tag{2.18}$$

The quaternion product is not commutative, so $a \otimes b \neq b \otimes a$.

2.3 PX4 autopilot architecture

PX4 is a professional autopilot developed by a world-class from industry and academia, supported by a world wide community [8]. It is designed to drive a large number of types of autonomous vehicles, including ground and aerial platforms.
PX4 consists in two main layers: the flight stack and the middleware. The flight stack is a collection of guidance, navigation and control algorithms for autonomous drones, the middleware consists in device drivers for embedded sensors, communication with the external world and the uORB publish-subscribe message bus. The detailed overview of the building blocks of the architecture of PX4 is shown in Figure 2.13. The top part contains the middleware layer while the bottom the components of the flight stack. The arrows show the most important connections between the modules, there are however other arrows that are not shown (e.g the parameters module is accessed by most of modules).
Modules communicates each others with a publishing/subscribing policy through bus messages called uORB. The uORB scheme assures that:

- The system is reactive - it is asynchronous and will update instantly when new data is available.
- All operations and communication are fully parallelized.
- A system component can consume data from anywhere in a thread-safe fashion.

Modules use the information contained in the messages through data structure defined internally: for each message they subscribe to, whenever an update occurs, they copy the information in their own internal structs, then elaborate it and publish the results. Typically the drivers define how fast a module updates. Most of the IMU drivers sample the data at 1kHz, integrate it and publish at 250Hz.
The PX4 architecture uses an other protocol to manage the communication between PX4 and the ground station (Section 2.1.5) that is MAVLink. It is a very lightweight messaging protocol that has been designed for the drone ecosystem. Besides the ground station, MAVLink is used as the integration mechanism for connecting to drone components outside of the flight controller (e.g companion computers, MAVLink enabled cameras).

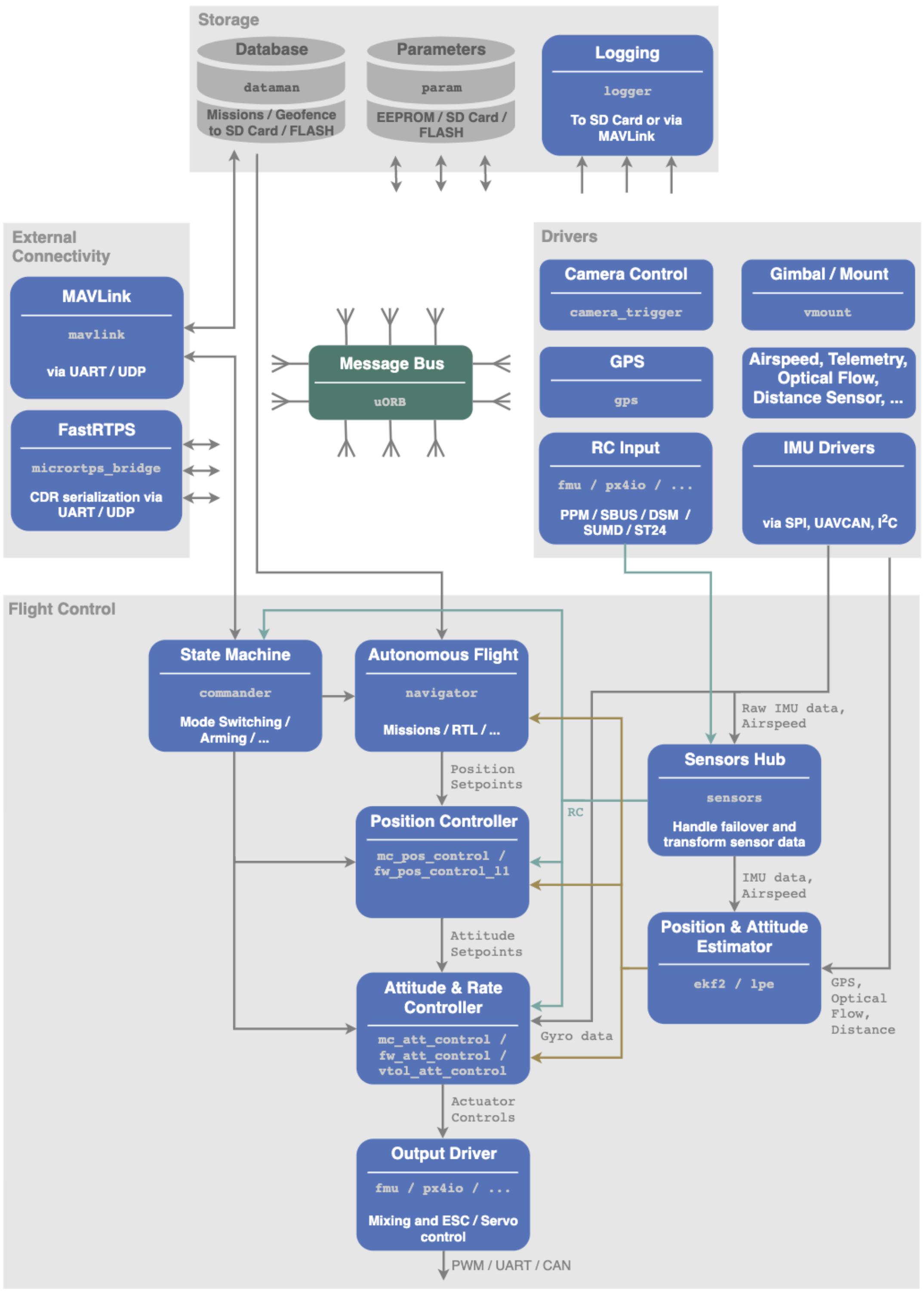


Figure 2.13: PX4 architecture overview.

2.3.1 Commander, position controller and attitude controller

An overview of the core modules of the flight stack will be given.

- `commander`
- `mc_pos_control`
- `mc_att_control`

The commander is the core of the PX4 firmware, it is a very complex module that manages many tasks, starting from what the vehicle should do to security checks. The commander has an internal state machine to manage the information coming from the Radio Controller: depending from the pilot commands it sets its state and several flags for autopilot purposes. The role of the state machine is not only regarding flight modes, but also for managing the arming phase of the vehicle: the commander can reject a flight mode or an arming state if some safety conditions are not satified. The initial state of the state machine is the manual flight mode (refers to Section 2.4 for details). Moreover, it manages the status of the leds, checks and resets (if needed) the position-velocity validity. Due to its complexity, it is currently in a refactoring process.
The position controller module has a twofold functionality: it runs the position controller law and acts as a setpoint generator based on the flight mode. For what concerns the position control law, it features a standard cascaded proportional loop for the position error and a PID loop for velocity error. Depending on the mode, the outer (position) loop can be bypassed. The position loop is only used when holding position or when the requested velocity in an axis is null.
The attitude controller module has two loops, a P loop for angular error and PD loop for angular rate error. It subscribes to many messages of the position controller. Besides the common subscriptions, the attitude controller subscribes to many sensors messages and its publications goes directly to the mixer.
The mixer is the component in the PX4 architecture (in the flight stack layer) that takes force commands (e.g. turn right) and translates them into individual motor commands, while ensuring that some limits are not exceeded. This translation is specific for a vehicle type and depends on various factors, such as the motor arrangements with respect to the center of gravity, or the vehicle's rotational inertia.

2.3.2 Flight controller module

The flight controller module is a custom module the Aerospace System and Control Laboratory (ASCL) at Politecnico of Milan has developed for research and educational activity. Its role is to replace the position and attitude controllers implemented in the PX4 firmware and to allow the execution of custom control laws. Table 2.5 shows the interface of the module.

Parameters	Description
$q_A^0(\phi^0, \theta^0, \psi^0)$	Desired attitude quaternion
p^0	roll angular rate (rad/s)
q^0	pitch angular rate (rad/s)
r^0	yaw angular rate (rad/s)
$\dot{p}^0$	roll angular acceleration (rad/s^2)
$\dot{q}^0$	pitch angular acceleration (rad/s^2)
$\dot{r}^0$	yaw angular acceleration (rad/s^2)
x^0	X position (m)
y^0	Y position (m)
z^0	Z position (m)
v_x^0	X velocity (m/s)
v_y^0	Y velocity (m/s)
v_z^0	Z velocity (m/s)
a_x^0	X acceleration (m/s^2)
a_y^0	Y acceleration (m/s^2)
a_z^0	Z acceleration (m/s^2)
$\dot{a}_x^0$	X jerk (m/s^3)
$\dot{a}_y^0$	Y jerk (m/s^3)
$\dot{a}_z^0$	Z jerk (m/s^3)
$\ddot{a}_x^0$	X snap (m/s^4)
$\ddot{a}_y^0$	Y snap (m/s^4)
$\ddot{a}_z^0$	Z snap (m/s^4)

Table 2.5: Flight controller module interface.

In Section 2.6, the issues related to its implementation are described.

2.3.3 uORB graphs

The following uORB graph describes all the topics which the multicopter position controller subscribes or publishes:

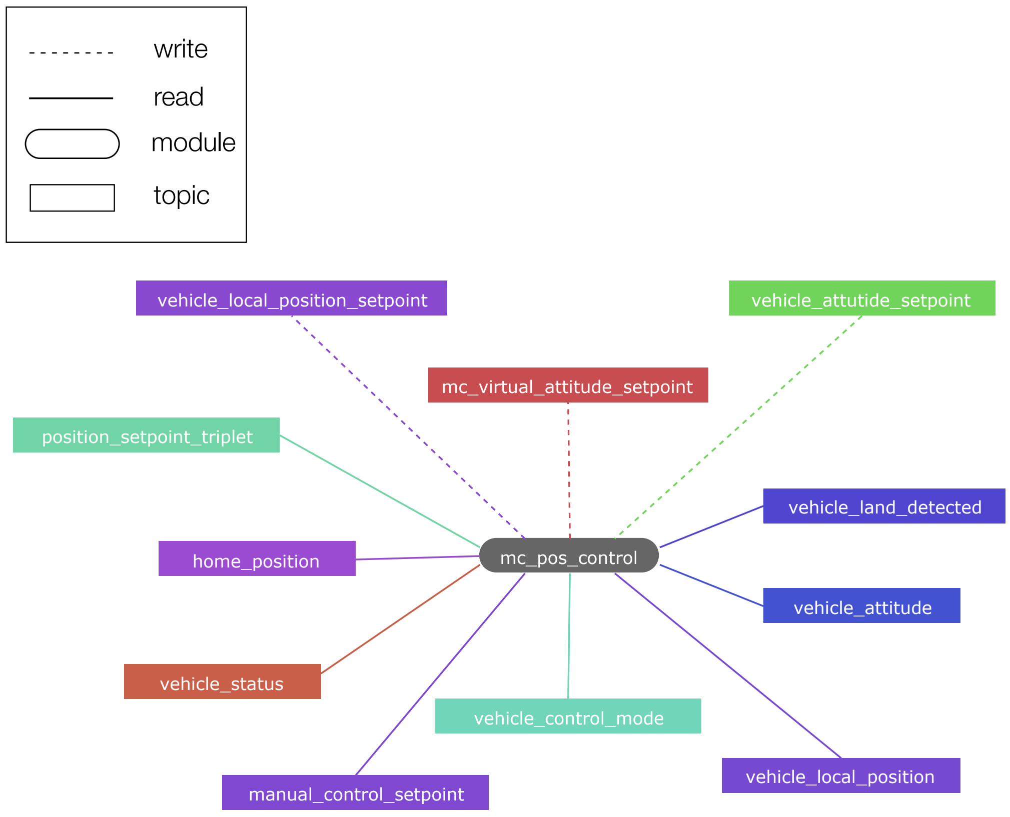


Figure 2.14: Multicopter position controller uORB graph.

The gray rounded corner box is the position controller module while the other coloured rectangular boxes are the topics. The continuous lines represents the subscribing process and the dashed lines the publishing one.
A detailed list of all subscribing topics used by the position controller module to reach its double task follows:

- `vehicle_land_detected` is published by land detector module, it contains some information related if vehicle is currently landed on the ground, if it is in free-fall, if it has ground contact but is not landed and the maximum altitude in [m] that can be reached.
- `vehicle_attitude` contains the quaternion rotation from NED earth frame to body frame.
- `vehicle_local_position` contains the information related to positions and velocities in local NED frame.

- `vehicle_control_mode` is published by the commander and it contains a list of flags representing the autopilot activity, in particular which controller should be activated with respect to the flight mode.
- `manual_control_setpoint` contains the input stick commands of the radio controller, basically the pilot's input, and the channels mapping.
- `vehicle_status` is published by commander and represents the state of the commander's state machine, in term of arming and navigation state. Moreover, it describes the status and the types of the vehicle.
- `position_setpoint_triplet` contains global position setpoint triplet in WGS84 coordinates, it is used for offboard modes purposes.

The position controller uses these topics to correctly mapping the pilot's input, computes the setpoint and publishes its control action; the uORBs published are:

- `vehicle_attitude_setpoint` contains the desired quaternion for quaternion control, the thrust and body angle in NED frame.
- `vehicle_local_position_setpoint` contains the position, velocity and acceleration resulting from the controller action.
- `mc_virtual_attitude_setpoint` contains the same information of `vehicle_attitude_setpoint` but it is used for VTOL (vertical take off and landing).

The following uORB graph shows the topics which the multicopter attitude controller subscribes or publishes.

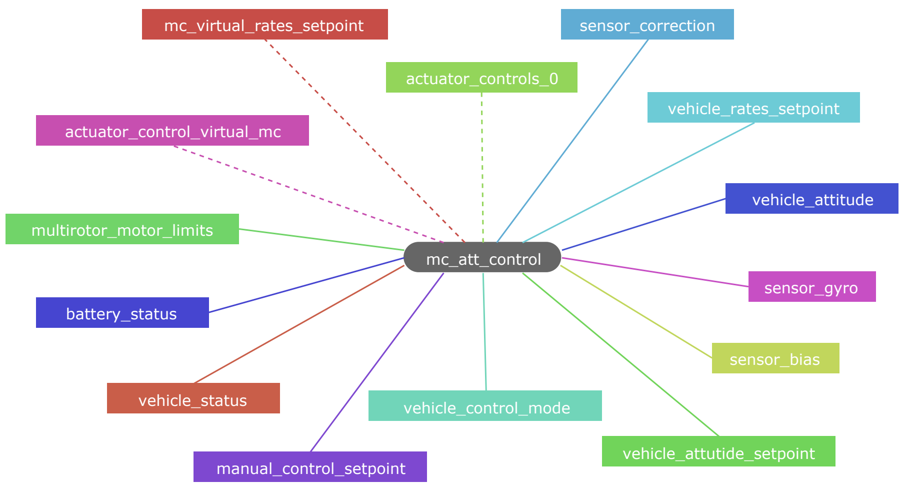


Figure 2.15: Multicopter attitude controller uORB graph.

2.4 Flight modes

Flight Modes define how the autopilot responds to user input and controls vehicle movement. They are grouped into manual, assisted and auto modes, based on the level/type of control provided by the autopilot. The pilot changes flight mode using switches on the remote control or with a ground control station.
Not all flight modes are available on all vehicle types, and some modes behave differently on different vehicle types. Finally, some flight modes make sense only under specific pre-flight and in-flight conditions (e.g. GPS lock, airspeed sensor, vehicle attitude sensing along an axis).
The system will not allow transitions to those modes until the right conditions are met. For thesis purposes and for the research context in which the Aerospace System and Control Laboratory works, focus has been put only on some flight modes of the multirotors.

2.4.1 Manual/Stabilized mode

In manual mode the user has direct control over the vehicle via the RC controller. When under manual control the roll and pitch sticks control the angle of the vehicle (attitude) around the respective axes, the yaw stick controls the rate of rotation above the horizontal plane, and the throttle controls altitude/speed.
The multicopter will level out and stop once the roll and pitch sticks are centered. The vehicle will then hover in place/maintain altitude - provided it is properly balanced, throttle is set appropriately, and no external forces are applied (e.g. wind). The craft will drift in the direction of any wind and the pilot has to control the throttle to hold altitude.

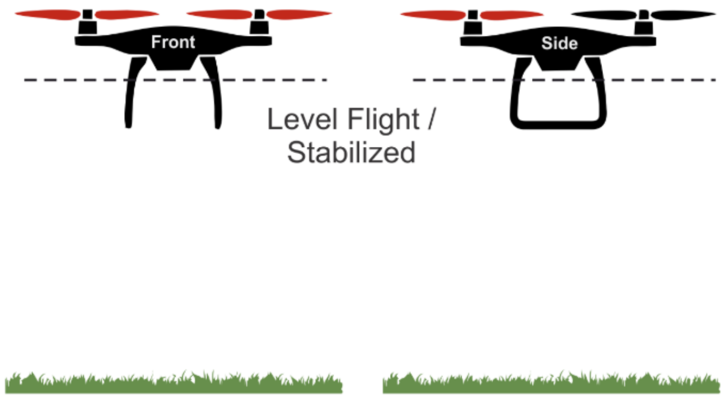


Figure 2.16: Manual mode.

The pilot's inputs are passed as roll and pitch angle commands and a yaw rate command. Throttle is rescaled and passed directly to the output mixer. The autopilot controls the attitude, meaning it regulates the roll and pitch angles to zero when the RC sticks are centered inside the controller deadzone. From a PX4 firmware point of view, the commander performs some checks and if the state related to the flight mode is not rejected, it sets its state in the internal state machine. Besides, the commander sets another parameter that is the navigation state (meaning what should the vehicle do, also in this case there is an enumeration associated). Based on the information provided from the navigation state parameter, it defines how the autopilot responds to user input, activating some controllers. For the manual/stabilized mode, the altitude controller and the position controller are disabled while the attitude controller is enabled.

2.4.2 Altitude mode

Altitude mode is a relatively easy-to-fly RC mode in which roll and pitch sticks control vehicle movement in the left-right and forward-back directions (relative to the “front” of the vehicle), yaw stick controls rate of rotation over the horizontal plane, and throttle controls speed of ascent-descent. When the sticks are released/centered the vehicle will level and maintain the current altitude. If the wind blows the aircraft will drift in the direction of the wind. Altitude mode is just like Manual mode but additionally locks the vehicle altitude when the sticks are released.

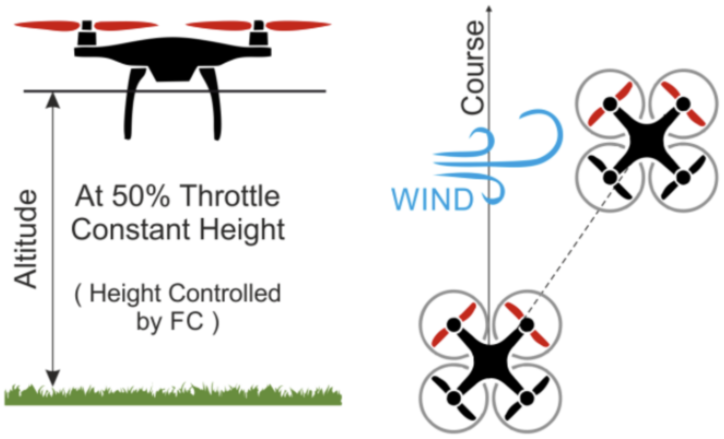


Figure 2.17: Altitude mode.

When the sticks are centered (within the deadband):

- Roll, Pitch, Yaw sticks levels vehicle.
- Throttle (about 50 percent) holds current altitude steady against wind.

Outside the center:

- Roll/Pitch sticks control tilt angle in respective orientations, resulting in corresponding left-right and forward-back movement.

- Throttle stick controls up/down speed with a predetermined maximum rate (and movement speed in other axes).
- Yaw stick controls rate of angular rotation above the horizontal plane.

The procedure to set the altitude mode in the PX4 firmware follows the description given in the manual mode section. Regarding the autopilot, altitude and attitude controller are both enabled while position controller is disabled.

2.4.3 Position mode

Position is an easy-to-fly RC mode in which roll and pitch sticks control speed over ground in the left-right and forward-back directions (relative to the “front” of the vehicle), and throttle controls speed of ascent-descent. When the sticks are released/centered the vehicle will actively brake, level, and be locked to a position in 3D space, compensating for wind and other forces. Position mode is the safest mode and unlike Altitude and Manual/Stabilized modes the vehicle will stop when the sticks are centered rather than continuing until slowed by wind resistance.

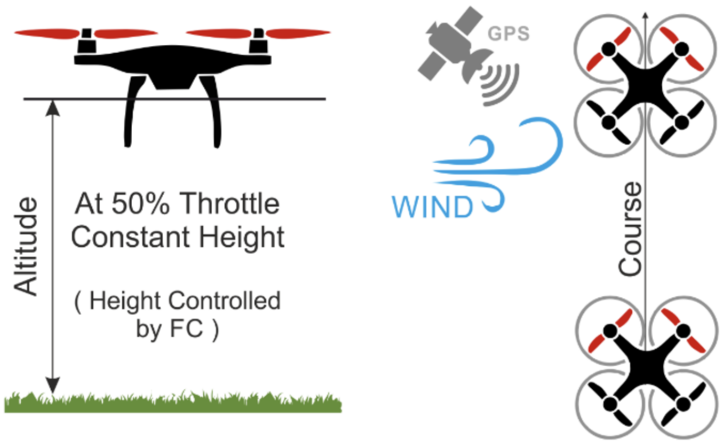


Figure 2.18: Position mode.

Roll, Pitch, Throttle sticks control speed in corresponding directions. Centered sticks level vehicle and hold it to fixed position and altitude against wind. When the sticks are centered (within the deadband):

- Hold x, y, z position steady against any wind and levels attitude.

Outside the center:

- Roll/Pitch sticks control speed over ground in left-right and forward-back directions (respectively) relative to the “front” of the vehicle.
- Throttle stick controls speed of ascent-descent.

- Yaw stick controls rate of angular rotation above the horizontal plane.

In the PX4 firmware context, the procedure to set position mode follows the description given in the previous section. Regarding the autopilot, altitude, attitude and position controllers are enabled. The position measurement is performed by a Motion Capture system, described in Section 2.1.4.

2.4.4 Offboard mode

The vehicle obeys a position, velocity or attitude setpoint provided over MAVLink. The setpoint may be provided by a MAVLink API running on a companion computer (and usually connected via serial cable or wifi).

- This mode requires position or pose/attitude information (e.g. GPS, optical flow, visual-inertial odometry, mocap, etc).
- This mode is automatic (RC control is disabled by default except to change modes).
- The vehicle must be already be receiving a stream of target setpoints before this mode can be engaged.
- The vehicle will exit the mode if target setpoints are not received at a rate larger than 2Hz.

Offboard mode is primarily used for controlling vehicle movement and attitude, and supports only a very limited set of MAVLink commands (more may be supported in future). It can be used to:

- Control vehicle position, velocity, or thrust.
- Control vehicle attitude/orientation.
- Control vehicle pose.

An additional offboard pose setpoint type has been implemented, to control position and attitude for platform in non-coventional configurations (e.g over-actuated).

2.4.5 Commander state value

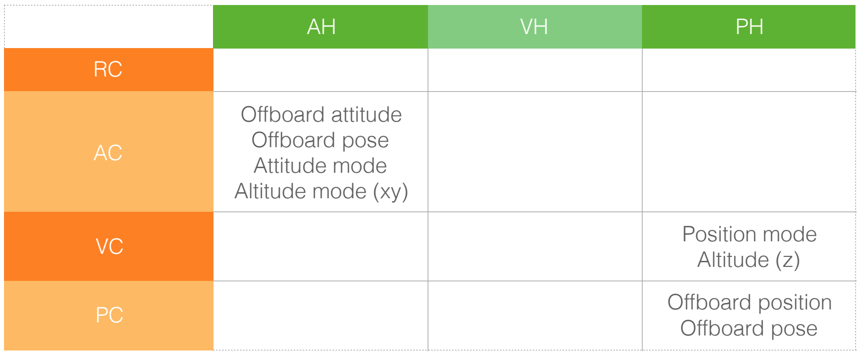
In the previous sections all the main flight modes have been presented, Table 2.6 describes how the `commander_state` has been set with respect to the current flight mode.

Shorthand	Flight mode	`commander_state`
`MAN`	manual	0
`ALT`	altitude	1
`POS`	position	2
`OFF`	offboard	7

Table 2.6: Commander state value table.

2.5 Command style

For each flight mode explained in the previous section (attitude mode coincides with the Manual/Stabilized mode), a combination of command style and hold capability has been defined.



	AH	VH	PH
RC			
AC	Offboard attitude Offboard pose Attitude mode Altitude mode (xy)		
VC			Position mode Altitude (z)
PC			Offboard position Offboard pose

Figure 2.19: Command style/hold capability table.

The *command style* defines the way in which the vehicle responds to a displacement of the pilot controls from the trim position. The *hold capability* is defined as the ability of the system to return to its equilibrium value following a pulse input; thus, it characterizes the capability of the system to reject external disturbance (e.g wind gusts). In Figure 2.19 vertical axis represents the command style:

- RC, Rate Command
- AC, Attitude Command
- VC, Velocity Command

- PC, Position Command

While horizontal axis represents the hold capabilty:

- AH, Attitude Hold
- VH, Velocity Hold
- PH, Position Hold

2.6 Issues related to the current firmware implementation

This section is going to describe which are the problems identified in the PX4 firmware with reference to the subject of this thesis, and the limits of the current system.

2.6.1 Comparison with firmware development status

PX4 firmware is constantly under development by an active worldwide community and the customizability and ease of use of the PX4 firmware make it very popular in the research field. In the Aerospace System and Control Laboratory (ASCL) at Politecnico of Milan a customized version of the firwmare based on the 1.7.3 version is currently in use (the same release used and presented in this thesis), so the last updates are not considered in this thesis (at the time of writing, the current stable version is v.1.9.1). In particular, the flight task architecture introduced in the latest releases is not considered. This feature aims at handling the generation of setpoints which are then fed to the position controller itself.
The development of such a feature was motivated by very similar considerations to the ones that are at the origin of this thesis (see the remainder of the chapter). Regarding flight modes, a new feature has been implemented in the custom firmware, concerning the offboard flight mode: the offboard pose mode (see Section 2.4.4 and Section 2.5). Besides, the custom firmware has a flight controller module containing the implementation of the controllers.

2.6.2 Modules issues

This section shows the identified problems related to position controller, commander and flight controller modules in terms of implementation and functionalities. In Sections 2.3.1 and 2.3.3 the position controller has been described in term of its twofold role of running the effective position control laws and the task of setpoint generation and which are the topics it subscribes to and publishes. The management of the setpoint generation based on the flight mode should be kept

out of the controller because it exceeds its scope. The position controller module subscribes to several topics that the commander publishes with the aim to compute the setpoint based on the information provided from commander internal state machine and the vehicle status. Among all the topics that subscribes, the only information which needs in order to run its control law is related to which flight mode is enabled and depending on this, it runs the position control law. For reasons of modularity and separation of functions, the controller should receive the setpoint and the generation of the setpoint should be managed externally. The position controller without the internal setpoint mapping and computation behaves in the same way of the attitude controller: it receives the setpoint and runs its control law if is requested by the autopilot.
Regarding the commander module (Section 2.3.1), it is extremely complicated to parse and to understand all the tasks that it covers, moreover the absence of an official documentation make it harder to debug according to discussions in the community forums [9]. This module should be refactored also to increase the scalabily of the parts of the code related to the multicopter, fixed wing and VTOL (vertical take off and landing) that are managed jointly. The current implementation of the flight controller module allows only the management of the offboard flight modes since both position and attitude controllers are always enabled. Currently this module has not the information about which flight mode the multicopter performs and so which controllers to be enabled.

Chapter 3

Setpoint mapper design

The first step in the design and development of a new system is the formalization of all the architectural choices that will concur in shaping the final system, in order to provide a general idea on how we intend to tackle the identified problems and solve them. In the following chapter the overview of the general architecture of the system will be described and then focus will be put on the custom setpoint mapper. Subsequently, a more in depth analysis of the components of the system will be presented by defining the implementation details, the role they cover and the software used to implement them.

3.1 High-level design decisions

Since the problems identified in the previous chapter are related to the commander and the position controller modules, two approaches have been considered:

- Approach 1: refactor both commander and position controller modules;
- Approach 2: preserve the commander module and refactor the position controller module;

Among the two approaches, it has been decided to use the second one. The reason coming from the fact that is more safe to maintain the commander since it performs several security checks. In addition, building the entire commander module from scratch is a hard work.
The following step concerns the re-implementation of the position controller module. Also in this case two approaches have been considered:

- Approach 2A: adopt a centralized architecture, keeping the setpoint mapper and the flight controller module in a unique block;
- Approach 2B: adopt a de-centralized architecture, dividing the setpoint mapper and the flight controller into two different modules;

In the first approach, the resulting system should be able to compute the setpoint and run the control laws based on the flight modes, this means to manage two different tasks in the same system and integrate the flight control to the generation of the setpoint. On the other hand, using a de-centralized architecture would guarantee that the two tasks are managed in different system components and the flight controller as it is implemented should be modified without any major changes. The choice among the presented approaches is straightforward. The resulting de-centralized architecture will have two blocks: one block for the setpoint generation and management of the flight mode and one block for control task.

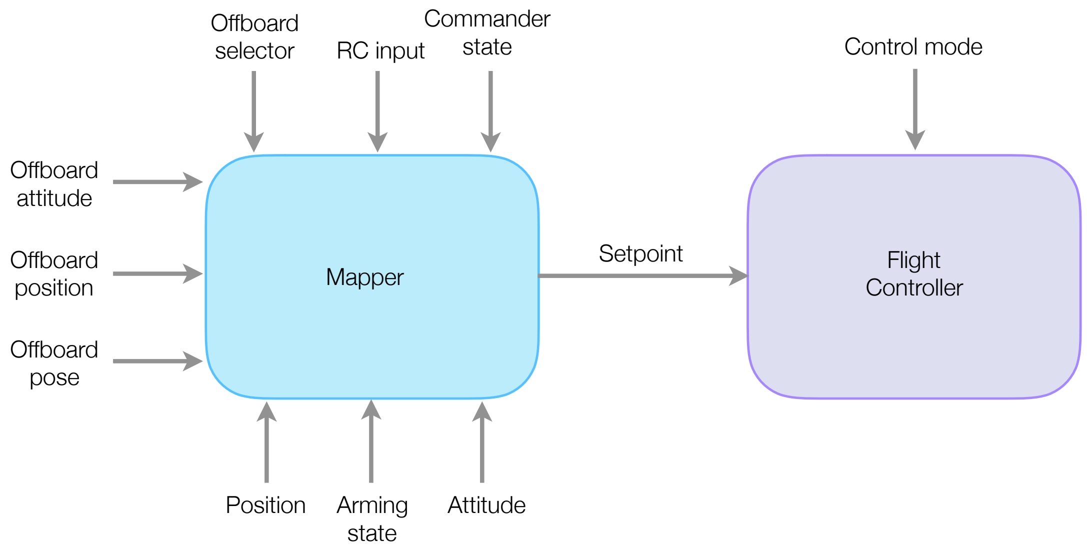


Figure 3.1: De-centralized system architecture.

In Figure 3.1 the whole system architecture is shown. The mapper block receives the information in order to map and compute the setpoint properly with respect to the enabled flight mode and it outputs the resulting setpoint values to the flight controller block. The flight controller receives the setpoint and runs the position and attitude control laws depending on the control mode enabled.

3.2 Setpoint mapper architecture

The aim of the setpoint mapper is a mapping procedure of the setpoint with respect to the response mode (each state of the state machine represents a different response mode, see Figure 3.4). From an architectural point of view: a two blocks system has been considered (Figure 3.2). The first block manages the switching among the flight modes with a state machine while the second block receives the response mode and generates the setpoints for that particular mode.
While in Figure 3.2 an high-level architectural model is shown, Figure 3.3 shows the implemented Simulink [10] model.

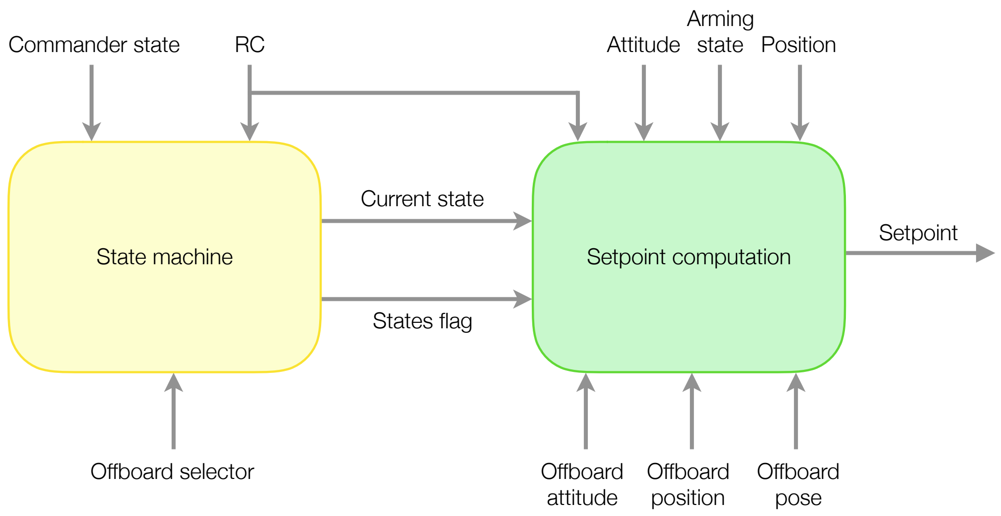


Figure 3.2: Setpoint mapper architecture.

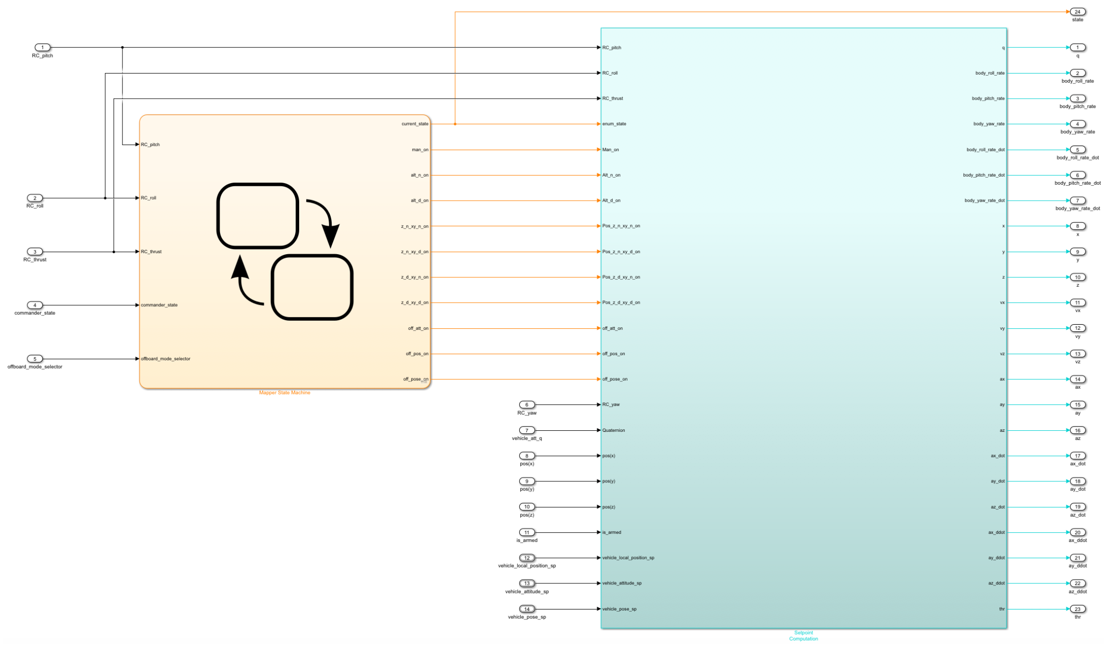


Figure 3.3: Screenshot of setpoint mapper Simulink model.

The following sections will explain in detail the two system components.

3.3 Mapper state machine

The mapper state machine has the purpose to set the current flight mode based on the information provided from the commander, the radio controller sticks and the offboard mode selector. Each state of the state machine corresponds to a different flight mode and each transition is managed by a commander action, a movement of the roll, pitch, thrust stick or a command from the ground station.

3.3.1 Stateflow

For the purpose of this thesis Stateflow [11] is used to model and simulate the mapper state machine. Stateflow is an add-on product to Simulink and it provides a graphical language that includes state transition diagrams, flow charts, state transition tables, and truth tables. It can be used to describe how MATLAB algorithms and Simulink models react to input signals, events, and time-based conditions. It is possible to model combinatorial and sequential decision logic that can be simulated as a block within a Simulink model or executed as an object in MATLAB. Graphical animation feature enables the analysis and debug of the implemented logic while it is executing. Stateflow is complemented with a set of tools for formal verification.

3.3.2 Mapper state machine states

As we can see in Figure 3.4 there are four macro-states that represent the main flight modes presented in Section 2.4 and then some sub-states that represent the response modes. States are summarized in Table 3.1.

State	Description	Enumeration
`MAN`	Manual mode	0
`ALT_N`	Altitude mode thrust neutral	1
`ALT_D`	Altitude mode thrust deflected	2
`Z_N_XY_N`	Position mode thrust neutral, roll/pitch neutral	3
`Z_N_XY_D`	Position mode thrust neutral, roll/pitch deflected	4
`Z_D_XY_N`	Position mode thrust deflected, roll/pitch neutral	5
`Z_D_XY_D`	Position mode thrust deflected, roll/pitch deflected	6
`POS_OFF`	Offboard position mode	7
`ATT_OFF`	Offboard attitude mode	8
`POSE_OFF`	Offboard pose mode	9

Table 3.1: Mapper state machine states table.

The suffixes "`_N`" and "`_D`" in the states name in Table 3.1 mean respectively the neutrality or deflection of a particular stick. To be considered neutral, the stick should be inside the deadzone otherwise it is deflected (see Section 3.3.3).

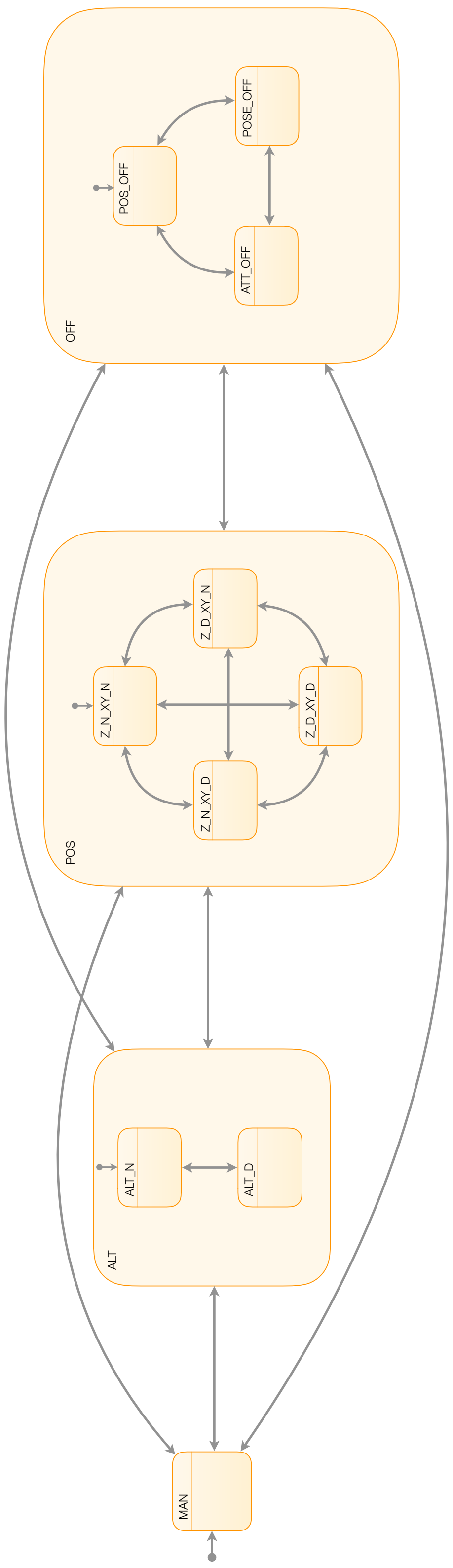


Figure 3.4: Setpoint mapper state machine.

Each state sets its own enumeration in the `current_state` parameter and a boolean representing its child activity, the information related if it is active or not. For example in Figure 3.5 it is reported the `POS_OFF` state:

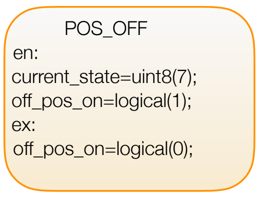


Figure 3.5: `POS_OFF` state.

As soon as the state becomes active, the current state enumeration is set to 7 and its activity to 1 (entry condition), when an outgoing transition happens, the flag `off_pos_on` will be set to 0 (exit condition). The parameter related to the enumeration provides the information about which response mode is enabled, the other parameter is used for reset purposes when a new response mode is activated.

3.3.3 Mapper state parameters and functions

This section is going to describe all the input/output parameters and the useful functions the mapper state machine has in order to fulfill its goal.

Parameters	Description	Type
`RC_pitch`	Forward stick displacement	input
`RC_roll`	Lateral stick displacement	input
`RC_thrust`	Thrust stick position	input
`commander_state`	State of the commander state machine	input
`offboard_mode_selector`	Selector of offboard mode	input
`current_state`	Mapper state	output
`man_on`	Bool: manual state is active	output
`alt_n_on`	Bool: alt neutral state is active	output
`alt_d_on`	Bool: alt deflected state is active	output
`z_n_xy_n_on`	Bool: pos neutral-neutral state is active	output
`z_n_xy_d_on`	Bool: pos neutral-deflected state is active	output
`z_d_xy_n_on`	Bool: pos deflected-neutral state is active	output
`z_d_xy_d_on`	Bool: pos deflected-deflected state is active	output
`off_pos_on`	Bool: offboard position is active	output
`off_att_on`	Bool: offboard attitude is active	output
`off_pose_on`	Bool: offboard pose is active	output

Table 3.2: Mapper state machine input/output table.

The input parameters shown in Table 3.2 come directly from the PX4 firmware.

The mapper state machine will subscribe to the topics containing the information which needs. In order to do that, once the overall system has been developed and the Simulink Embedded Coder (see Section 3.7) has been run, the generated code will be integrated in the firmware in a custom setpoint mapper module. This module will subscribe to the following topics:

- `manual_control_setpoint.msg`, it contains the value of the RC sticks;
- `commander_state.msg`, it contains the value of the current state of the commander state machine;

Besides the parameters, the mapper state machine has also two MATLAB function to take into account the deflections of the radio controller sticks, the `RC_yaw` stick is not considered since it is not useful in the mapping procedure. The first function takes as input parameter the thrust of the radio controller and outputs a boolean value to represent the neutrality or deflection of the stick. The second function takes as input two parameters, the pitch and the roll radio controller sticks and outputs a boolean value to represent the neutrality or deflection of one of them of both. To summarize:

$$\Delta_z = 0.05$$

$$\Delta_{xy} = 0.05$$

$$f_N^z(z) = \begin{cases} 1 & |z - 0.5| \leq \Delta_z \\ 0 & \text{otherwise} \end{cases} \tag{3.1}$$

$$f_N^{xy}(x, y) = \begin{cases} 1 & |x| \leq \Delta_{xy} \wedge |y| \leq \Delta_{xy} \\ 0 & \text{otherwise} \end{cases} \tag{3.2}$$

The Δ_z and Δ_{xy} are set by default to 0.05, these values are not hardcoded and they can be modified.

3.3.4 Mapper state machine transitions

The switching between states is managed by five input variables:

- `commander_state`
- `RC_pitch`
- `RC_roll`
- `RC_thrust`
- `offboard_mode_selector`

For sake of simplicity, some defines have been used:

- X=`RC_pitch` as the forward stick displacement;
- Y=`RC_roll` as the lateral stick displacement;
- Z=`RC_thrust` as the throttle stick displacement;
- R=`RC_yaw` as the directional stick displacement;

The commander manages the transitions between the macro-states (so between the main flight modes) based on the current state of its internal state machine, the thrust stick manages the transitions between response modes when altitude mode is enabled, the roll and pitch sticks together with the thrust stick manage the transitions among response modes when the position mode is enabled, depending on the neutrality of deflection with respect to the deadzone.
Finally the offboard mode selector manages the transitions between the response modes of the offboard flight mode depending on the type of the MAVLink message coming from the ground station related to the specific offboard mode.
In Tables 3.3, 3.4 and 3.5 all the possible transitions between the states and the consequent action condition are listed. The last column represents the evalution order from a state to the next one.

Current state	Next state	Conditions	Evaluation order
`ALT_N`	`ALT_D`	NOT($f_N^z(Z)$)	-
`ALT_D`	`ALT_N`	$f_N^z(Z)$	-
`Z_N_XY_N`	`Z_D_XY_D`	NOT($f_N^z(Z)$) AND NOT($f_N^{xy}(X,Y)$)	1
`Z_N_XY_N`	`Z_D_XY_N`	NOT($f_N^z(Z)$) AND $f_N^{xy}(X,Y)$	2
`Z_N_XY_N`	`Z_N_XY_D`	$f_N^z(Z)$ AND NOT($f_N^{xy}(X,Y)$)	3
`Z_N_XY_D`	`Z_D_XY_D`	NOT($f_N^z(Z)$) AND NOT($f_N^{xy}(X,Y)$)	1
`Z_N_XY_D`	`Z_N_XY_N`	$f_N^z(Z)$ AND $f_N^{xy}(X,Y)$	2
`Z_N_XY_D`	`Z_D_XY_N`	NOT($f_N^z(Z)$) AND $f_N^{xy}(X,Y)$	3
`Z_D_XY_N`	`Z_N_XY_N`	$f_N^z(Z)$ AND $f_N^{xy}(X,Y)$	1
`Z_D_XY_N`	`Z_D_XY_D`	NOT($f_N^z(Z)$) AND NOT($f_N^{xy}(X,Y)$)	2
`Z_D_XY_N`	`Z_N_XY_D`	$f_N^z(Z)$ AND NOT($f_N^{xy}(X,Y)$)	3
`Z_D_XY_D`	`Z_N_XY_N`	$f_N^z(Z)$ AND $f_N^{xy}(X,Y)$	1
`Z_D_XY_D`	`Z_N_XY_D`	$f_N^z(Z)$ AND NOT($f_N^{xy}(X,Y)$)	2
`Z_D_XY_D`	`Z_D_XY_N`	NOT($f_N^z(Z)$) AND $f_N^{xy}(X,Y)$	3

Table 3.3: Transitions managed by the RC sticks.

Current macro-state	Next macro-state	Conditions	Evaluation order
MAN	ALT	commander_state==1	1
MAN	POS	commander_state==2	2
MAN	OFF	commander_state==7	3
ALT	MAN	commander_state==0	1
ALT	OFF	commander_state==7	2
ALT	POS	commander_state==2	3
POS	MAN	commander_state==0	1
POS	ALT	commander_state==1	2
POS	OFF	commander_state==7	3
OFF	MAN	commander_state==0	1
OFF	ALT	commander_state==1	2
OFF	POS	commander_state==2	3

Table 3.4: Transitions managed by the commander.

Current state	Next state	Condition	Evaluation order
POS_OFF	ATT_OFF	offboard_mode_selector==1	1
POS_OFF	POSE_OFF	offboard_mode_selector==2	2
ATT_OFF	POSE_OFF	offboard_mode_selector==2	1
ATT_OFF	POS_OFF	offboard_mode_selector==0	2
POSE_OFF	POS_OFF	offboard_mode_selector==0	1
POSE_OFF	ATT_OFF	offboard_mode_selector==1	2

Table 3.5: Transitions managed by the offboard mode selector.

3.3.5 Offboard mode selector

The offboard mode selector has been implemented in the PX4 firmware for take into account the type of the MAVLink messages coming from the ground control station. In Table 3.6 `offboard_mode_selector` values are shown.

Shorthand	Flight mode	offboard_mode_selector
POS_OFF	offboard position	0
POS_ATT	offboard altitude	1
POS_POSE	offboard pose	2

Table 3.6: Offboard mode selector value table.

3.4 Setpoint computation

The setpoint computation has the purpose to generate the setpoint according to the response mode. The implementation of this component follows a switch/switch-case/merge logic: it receives the information related to which response mode is active from the mapper state machine (the value of the `current_state`) and it selects the corresponding switch-case block (the setpoint computation has a switch-case block for each state of the mapper state machine that corresponds to a specific response mode). Within the selected subsystem, the setpoint calculation is performed; the merge block selects the setpoint computed by the currently active state and routes it to the output of the system.

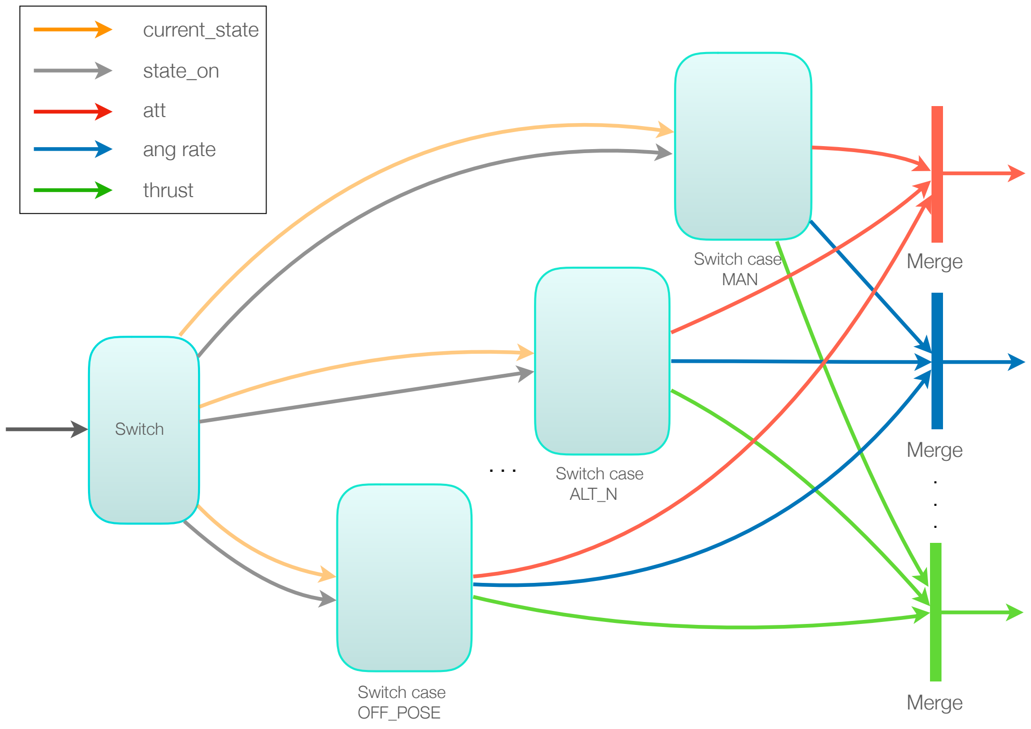


Figure 3.6: Setpoint computation block diagram.

Figure 3.6 shows the idea behind the block diagram implementation in a simple case with three states and the three setpoint variables of attitude, angular rate and the thrust for sake of simplicity.

3.4.1 Simulink

For the aim of this thesis, Simulink is used to model the setpoint computation block. Simulink is an add-on product to MATLAB, it is a graphical programming environment for modeling, simulating and analyzing multi-domain dynamical systems and it is developed by MathWorks. Thanks to a set of customizable block

libraries, it enables rapid construction of virtual prototypes to explore design concepts at any level of detail, with minimal effort. For the modeling part, Simulink provides a GUI for building models as block diagrams simply using the drag-and-drop operations. An interesting feature of Simulink is the possibility to generate code in C, C++ and other languages starting from a dynamic model. The generated code can be integrated like a source code, static libraries or dynamic libraries into application running outside of MATLAB and Simulink environment.
The idea is use this Simulink feature and integrate the generated code of the setpoint mapper in the custom firmware.

3.4.2 Input variables

Table 3.7 shows all the signals and parameters the setpoint computation uses in order to map and compute the setpoint properly.

Parameters	Description
`RC_pitch`	Forward stick displacement
`RC_roll`	Lateral stick displacement
`RC_thrust`	Thrust stick displacement
`RC_yaw`	Yaw stick displacement
`current_state`	Mapper state
`man_on`	Bool: manual state is active
`alt_n_on`	Bool: alt neutral state is active
`alt_d_on`	Bool: alt deflected state is active
`z_n_xy_n_on`	Bool: pos neutral-neutral state is active
`z_n_xy_d_on`	Bool: pos neutral-deflected state is active
`z_d_xy_n_on`	Bool: pos deflected-neutral state is active
`z_d_xy_d_on`	Bool: pos deflected-deflected state is active
`off_pos_on`	Bool: offboard position is active
`off_att_on`	Bool: offboard attitude is active
`off_pose_on`	Bool: offboard pose is active
`pos(x)`	North position in NED earth-fixed frame
`pos(y)`	East position in NED earth-fixed frame
`pos(z)`	Down position in NED earth-fixed frame
`is_armed`	Bool: current arming state is armed
`vehicle_att_q`	Attitude quaternion from NED to body frame
`vehicle_local_position_sp`	Local position setpoint in NED frame
`vehicle_attitude_sp`	Quaternion-based attitude setpoint
`vehicle_pose_sp`	Pose setpoint for fully-actuated UAV

Table 3.7: Setpoint computation block input parameters.

The last three parameters are send through offboard commands. For what concerns the other parameters, they come from the output of the state machine and from the PX4 firmware. Section 3.3.3 explained how the state machine subscribes to the uORB messages and the setpoint computation follows the same idea. In order to retrieve the information which needs, it can subscribe to:

- `manual_control_setpoint.msg`, it contains the value of the RC sticks;
- `vehicle_local_position.msg`, it contains the position and velocity of the vehicle, this is estimated based on information coming from the Motion-Capture system (see Section 2.1.4);
- `vehicle_status.msg`, it contains the arming state information;
- `vehicle_local_position_setpoint.msg`, it contains the setpoint for the offboard position mode;
- `vehicle_attitude_setpoint.msg`, it contains the setpoint for the offboard attitude mode;
- `vehicle_pose_setpoint.msg`, it contains the setpoint of the offboard pose mode;

3.4.3 Output variables

Table 3.8 shows the output variables of the setpoint computation block, that are the setpoint signals which feed the flight controller.

Parameters	Description
$q_A^0(\phi^0, \theta^0, \psi^0)$	Desired attitude quaternion
p^0	roll angular rate (rad/s)
q^0	pitch angular rate (rad/s)
r^0	yaw angular rate (rad/s)
$\dot{p}^0$	roll angular acceleration (rad/s^2)
$\dot{q}^0$	pitch angular acceleration (rad/s^2)
$\dot{r}^0$	yaw angular acceleration (rad/s^2)
x^0	X position (m)
y^0	Y position (m)
z^0	Z position (m)
v_x^0	X velocity (m/s)
v_y^0	Y velocity (m/s)
v_z^0	Z velocity (m/s)
a_x^0	X acceleration (m/s^2)
a_y^0	Y acceleration (m/s^2)
a_z^0	Z acceleration (m/s^2)
$\dot{a}_x^0$	X jerk (m/s^3)
$\dot{a}_y^0$	Y jerk (m/s^3)
$\dot{a}_z^0$	Z jerk (m/s^3)
$\ddot{a}_x^0$	X snap (m/s^4)
$\ddot{a}_y^0$	Y snap (m/s^4)
$\ddot{a}_z^0$	Z snap (m/s^4)
T^0	Thrust

Table 3.8: Setpoint computation output variables.

The output of the setpoint computation block, together with the information related to the `state` of the mapper state machine, constitutes the output of the setpoint mapper system (the information related to the mapper state has been used for testing purposes and it is not useful for the flight controller).
In order to feed the flight controller module, it has been created a uORB topic which contains all the computed setpoint: `flight_controller_setpoint.msg`.

3.4.4 Mapping laws

The role of the setpoint computation block is mapping the pilot and offboard commands to the setpoint that will feed the flight controller. This mapping depends on the current response mode and Table 3.9 explains how the radio controller sticks have been mapped for the computation of the setpoint.

Current state	X	Y	Z	R
`MAN`	θ^0	ϕ^0	T^0	ψ^0
`ALT_N`	θ^0	ϕ^0	z^0	ψ^0
`ALT_D`	θ^0	ϕ^0	v_z^0	ψ^0
`POS_Z_N_XY_N`	x^0	y^0	z^0	ψ^0
`POS_Z_N_XY_D`	v_x^0	v_y^0	z^0	ψ^0
`POS_Z_D_XY_N`	x^0	y^0	v_z^0	ψ^0
`POS_Z_D_XY_D`	v_x^0	v_y^0	v_z^0	ψ^0
`OFF_POS`	-	-	-	-
`OFF_ATT`	-	-	-	-
`OFF_POSE`	-	-	-	-

Table 3.9: Mapping laws table for RC modes.

The vertical axis represents the response mode while the horizontal axis the radio controller sticks defined in Section 3.3.4. This mapping is consistent with the command style/hold capability associated with the flight modes, reported in Section 2.5.

3.4.5 Radio setpoint computation

In the previous section it was presented how the sticks are mapped for the setpoint computation, this section is going to describe how the setpoint has been computed. Parameters used in the computation are defined in Section 3.5.

- longitudinal stick $X \in [-1, 1]$ (positive up)
- lateral stick $Y \in [-1, 1]$ (positive right)
- throttle stick $Z \in [0, 1]$ (positive up)
- directional stick $R \in [-1, 1]$ (positive right)

Thrust - T^0

$$T' = \begin{cases} 2\bar{T}Z & Z < \bar{Z} \\ \bar{T} + 2(1-\bar{T})(Z-\bar{Z}) & Z \geq \bar{Z} \end{cases} \tag{3.3}$$

where $\bar{T}$ represents the normalized thrust needed to maintain the UAV in hover. The idea is that the Z stick should stay in the middle position $\bar{Z} = 0.5$ when hovering; this is done by remapping the Z stick to thrust as a piecewise linear function. In case $\bar{T} = 0.5$, the expression simplifies to $T' = Z$. Afterwards, a re-scaling procedure follows.

$$T'' = \begin{cases} T' & T' < \texttt{MPC\_MANTHR\_MAX} \\ \texttt{MPC\_MANTHR\_MAX} & T' \geq \texttt{MPC\_MANTHR\_MAX} \end{cases} \tag{3.4}$$

$$T^0 = \begin{cases} \texttt{MPC\_MANTHR\_MIN} & T'' < \texttt{MPC\_MANTHR\_MIN} \\ T'' & T'' \geq \texttt{MPC\_MANTHR\_MIN} \end{cases} \tag{3.5}$$

Pitch angle - θ^0

(positive nose up)

$$\theta^0 = -\texttt{MPC\_MAN\_TILT\_MAX} \cdot X \tag{3.6}$$

Roll angle - ϕ^0

(positive right wing down)

$$\phi^0 = \texttt{MPC\_MAN\_TILT\_MAX} \cdot Y \tag{3.7}$$

Yaw angle - ψ^0

(positive clockwise from above)

$$r^0 = \texttt{MPC\_MAN\_Y\_MAX} \cdot R \tag{3.8}$$

$$\psi^0(t) = \psi^0(t_0) + \int_{t_0}^{t} r^0(\tau)d\tau \qquad t \geq t_0 \tag{3.9}$$

$$\psi^0(t_0) = \psi(t_0) \tag{3.10}$$

Vertical velocity - v_z^0

First, the throttle stick is remapped from [0,1] to up and down command [-1,1].

$$Z' = (Z - 0.5) * 2 \tag{3.11}$$

$$f(x) = \begin{cases} 0 & -\Delta \leq x \leq \Delta \\ \dfrac{x-\Delta}{1-\Delta} & \Delta < x \leq 1 \\ \dfrac{x+\Delta}{1-\Delta} & -1 \leq x < \Delta \end{cases} \tag{3.12}$$

$$g(x) = (1-e)x + x^3 \tag{3.13}$$

where

$$\Delta = \frac{\texttt{MPC\_HOLD\_DZ}}{2}$$
$$e = \texttt{MPC\_Z\_MAN\_EXPO}$$

$$v_z^0 = -(g(f(Z'))) \tag{3.14}$$

In-plane velocity - v_x^0, v_y^0

$$v_x^{B,0} = V_{XY}^{MAX} X \tag{3.15}$$
$$v_y^{B,0} = V_{XY}^{MAX} Y \tag{3.16}$$

where the B superscript indicates velocity is expressed in body frame. The body frame velocity setpoint is then rotated in inertial NED frame.

$$\begin{bmatrix} \bar{v}_x^0 \\ \bar{v}_y^0 \end{bmatrix} = R(\psi) \begin{bmatrix} v_x^{B,0} \\ v_y^{B,0} \end{bmatrix} \tag{3.17}$$

where

$$R(\psi) = \begin{bmatrix} \cos(\psi) & -\sin(\psi) \\ \sin(\psi) & \cos(\psi) \end{bmatrix} \tag{3.18}$$

$$v^0 = \begin{bmatrix} g(f(\bar{v}_x^0)) \\ g(f(\bar{v}_y^0)) \end{bmatrix} \tag{3.19}$$

$$h(v) = \begin{cases} v & |v| \leq 1 \\ \dfrac{v}{|v|} & |v| > 1 \end{cases} \qquad v \in \mathbb{R}^2 \tag{3.20}$$

$$h(v^0) = \begin{bmatrix} v_x^0 \\ v_y^0 \end{bmatrix} \tag{3.21}$$

Position - x^0, y^0, z^0

(same for x^0, y^0, z^0)

$$x^0(t) = x^0(t_0) \qquad t \geq t_0 \tag{3.22}$$

$$x^0(t_0) = x(t_0) \tag{3.23}$$

which means that the position measurement at time instant t_0 is used as a constant position setpoint from that time on.

Figure 3.7 shows the switch-case setpoint computation block for the manual mode as an example, the other subsystems follow the same structure.

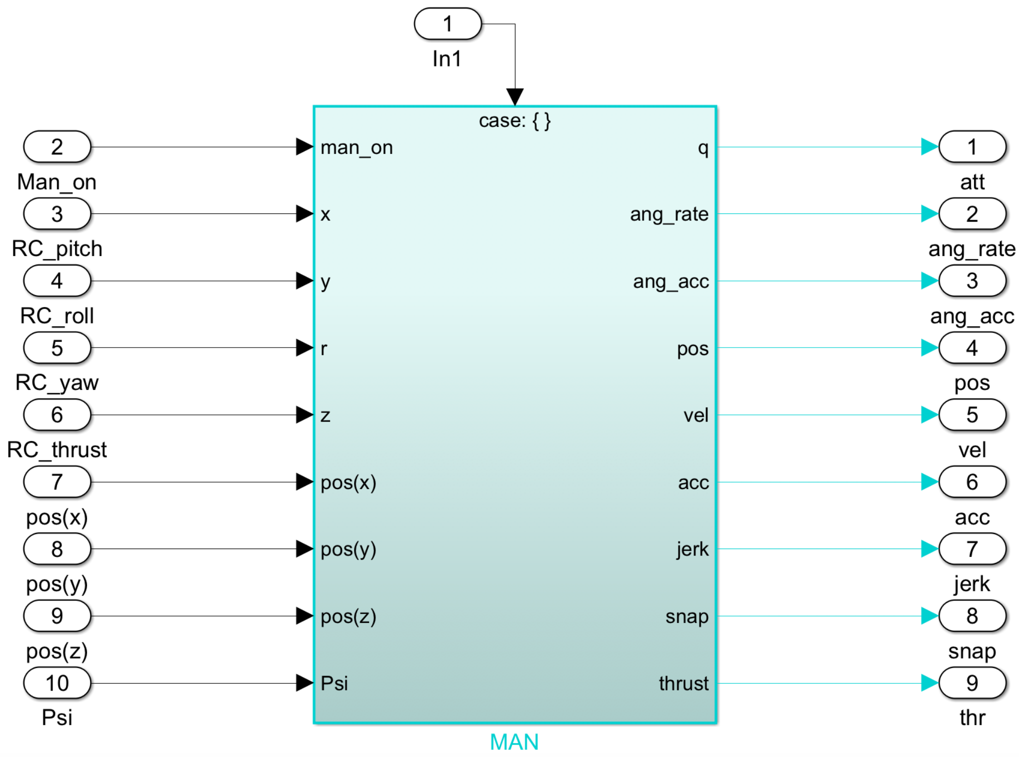


Figure 3.7: Switch case: manual mode.

The block receives the information which needs to perfom the computation and outputs the setpoint that are going to feed the flight controller.
The setpoint computation output of all the switch-case blocks is summarized in Table 3.10 (the `Att` setpoint follows the formalism used in the other tables: $q_A^0(\phi^0, \theta^0, \psi^0)$).
For sake of simplicity, the setpoints which are not relevant or applicable to a particular mode are set to zero.
Selecting the current signal value of a signal as a setpoint is equivalent to opening the feedback loop on that particular signals (i.e. the tracking error is set to zero) e.g. `Z_D_XY_D`.
In order to compute a generic value at time t_0 a custom Simulink component named “Hold on rising” has been implemented that acts as follows: as soon as a state becomes active and the flag related of its activity rises, the hold on rising block has a detect rising block in order to perceive the rising activity.
Then, thanks to a memory block, it is able to read the value at time t_0 and hold the output at that value. The implementation is depicted in Figure 3.8.

Setpoint	MAN	ALT_N	ALT_D	Z_N_XY_N	Z_N_XY_D	Z_D_XY_N	Z_D_XY_D
Att	ϕ^0 θ^0 ψ^0	ϕ^0 θ^0 ψ^0	ϕ^0 θ^0 ψ^0	ϕ^0=0 θ^0=0 ψ^0	ϕ^0=0 θ^0=0 ψ^0	ϕ^0=0 θ^0=0 ψ^0	ϕ^0=0 θ^0=0 ψ^0
Ang rate	0	0	0	0	0	0	0
Ang acc	0	0	0	0	0	0	0
Pos	$x(t_0)$ $y(t_0)$ $z(t_0)$	$x(t_0)$ $y(t_0)$ $z(t_0)$	$x(t_0)$ $y(t_0)$ $z(t)$	$x(t_0)$ $y(t_0)$ $z(t_0)$	$x(t)$ $y(t)$ $z(t_0)$	$x(t_0)$ $y(t_0)$ $z(t)$	$x(t)$ $y(t)$ $z(t)$
Vel	0	0	v_x^0=0 v_y^0=0 v_z^0	0	v_x^0 v_y^0 v_z^0=0	v_x^0=0 v_y^0=0 v_z^0	v_x^0 v_y^0 v_z^0
Acc	0	0	0	0	0	0	0
Jerk	0	0	0	0	0	0	0
Snap	0	0	0	0	0	0	0
Thrust	T^0	0	0	0	0	0	0

Table 3.10: Setpoint computation output table for RC modes.

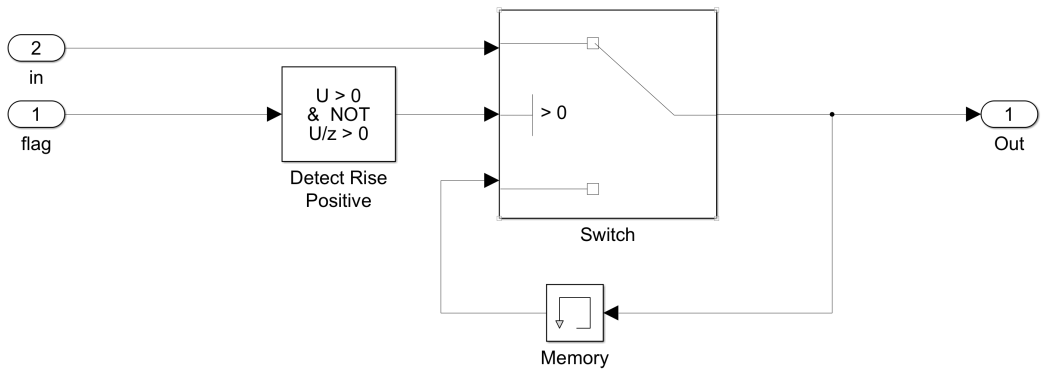


Figure 3.8: "Hold on rising" Simulink block.

3.4.6 Offboard setpoint computation

Regarding the offboard modes, the computation of the setpoints has been managed in a different way respect to RC modes. Basically in order to map these setpoint, the setpoint computation should be able to read the content of the uORB messages containing the setpoint of the offboard modes and pass them through to the flight controller.
The topics related to the offboard setpoint are:

- `vehicle_local_position_setpoint.msg`
- `vehicle_attitude_setpoint.msg`
- `vehicle_pose_setpoint.msg`

Table 3.11 shows how the offboard modes setpoint are mapped to the output setpoint variables.

Setpoint	`OFF_POS`	`OFF_ATT`	`OFF_POSE`
`Att`	$q_A^0(0, 0, \psi^0)$	$q_A^0(\phi^0, \theta^0, \psi^0)$	$q_A^0(\phi^0, \theta^0, \psi^0)$
`Ang rate`	0	0	p^0 q^0 r^0
`Ang acc`	0	0	$\dot{p}^0$ $\dot{q}^0$ $\dot{r}^0$
`Pos`	x^0 y^0 z^0	$x(t_0)$ $y(t_0)$ $z(t_0)$	x^0 y^0 z^0
`Vel`	v_x^0 v_y^0 v_z^0	0	v_x^0 v_y^0 v_z^0
`Acc`	a_x^0 a_y^0 a_z^0	0	a_x^0 a_y^0 a_z^0
`Jerk`	0	0	$\dot{a}_x^0$ $\dot{a}_y^0$ $\dot{a}_z^0$
`Snap`	0	0	$\ddot{a}_x^0$ $\ddot{a}_y^0$ $\ddot{a}_z^0$
`Thrust`	0	T^0	0

Table 3.11: Offboard setpoint computation table.

Within all the switch-case offboard blocks a security check has been implemented in order to manage the new messages coming from the MAVLink. Thanks to this implementation, the vehicle is able to listen to new offboard messages only if it is armed, starting from the time instant when entering the current state.
The Simulink block interfaces with the uORB messages coming from the firmware through bus structures. A bus object has been created for each message reflecting the structure of the corresponding uORB message and thanks to a bus selector block, the information the block needs to generate the setpoint has been extracted.

3.5 PX4 firmware parameters and constants

In order to compute the setpoint properly, the setpoint computation block uses some PX4 parameters and constants. These parameters can also be found in PX4 firmware in the `mc_pos_control` module, precisely in `mc_pos_control_params.c`. They are defined externally from the Simulink model in a MATLAB file, named Parameters.m, that is loaded whenever the Simulink block runs. Parameters are shown in Table 3.12.

PX4 parameter	Description
`MPC_MAN_TILT_MAX`	Max tilt angle in manual or altitude mode
`MPC_MAN_Y_MAX`	Max manual yaw rate
`MPC_THR_HOVER`	Hover thrust
`MPC_MANTHR_MIN`	Minimum manual thrust
`MPC_Z_MAN_EXPO`	Manual control stick vertical exponential curve
`MPC_XY_MAN_EXPO`	Manual position control stick exponential curve sensitivity
`MPC_HOLD_DZ`	Deadzone sticks where position hold is enabled

Table 3.12: PX4 firmware parameters table.

Table 3.13 shows the minimum and maximum value the parameters can assume and the default value set.

PX4 parameter	Min	Max	Default
`MPC_MAN_TILT_MAX`	0.0	90.0	35.0
`MPC_MAN_Y_MAX`	0.0	400.0	200.0
`MPC_THR_HOVER`	0.1	0.8	0.5
`MPC_MANTHR_MIN`	0.0	1.0	0.08
`MPC_MANTHR_MAX`	0.0	1.0	0.9
`MPC_Z_MAN_EXPO`	0	1	0.0
`MPC_XY_MAN_EXPO`	0	1	0.0
`MPC_HOLD_DZ`	0.0	1.0	0.1

Table 3.13: PX4 parameters min/max table.

Other internally defined parameters are shown in Table 3.14.
In the Table 3.14 an interesting parameter is the `sample_time` that represents the sampling time of the system that is 250 Hz.
All these parameters are not hardcoded and they can be modified, for example the hovering thrust of ANT-1 (see Section 2.6) is about 0.3.

Constant	Description
`DEG_TO_RAD`	conversion grad to rad
`RAD_TO_DEG`	conversion rad to grad
`THRESH_ARM_Z`	threshold on thrust stick lower value during arming
`THRESH_ARM_R`	threshold on yaw stick upper value during arming
`sample_time`	system sampling time

Table 3.14: Constants table.

3.6 Flight controller interface

In order to solve the issues related to the flight controller presented in Section 2.6, the interface of the flight controller module has been modified: in addition to the setpoint listed in Table 2.5 the throttle setpoint has been included, so the resulting interface will concide with the setpoint computation output. To retrieve the information related to which controllers are to be enabled, the topic `vehicle_control_mode.msg` has been considered. The flight controller module subscribes to this topic published by the commander that contains the information about which controllers to be enabled. These modification allow to expand the management also to other flight modes.

3.7 PX4 firmware integration

This section describes how the integration to the custom PX4 firmware has been performed. Once the Simulink model of the setpoint mapper system has been developed and all the requirements has been defined, C++ code is automatically generated by the Simulink Embedded Coder and the autogenerated code is ready to be injected in the PX4 firmware. Once defined the I/O interface of the Simulink setpoint mapper, the next step is to implement it also in the PX4 firmware. As mentioned previously, PX4 is composed by different modules which correspond to specific task in the firmware.
In this case the creation of a new module will be essential to handle the new mapping system that will be imported from Simulink. The tasks of the new module will be to subscribe to the needed topics (those listed in Section 3.3.3 and 3.4.2) and to publish a new topic for the flight controller module (Section 3.4.3). The setpoint mapper PX4 module is provided with an interface consistent with the one defined for the setpoint mapper Simulink model.

Chapter 4

Verification and Testing

This chapter will focus on testing setpoint mapper. A formal testing procedure has been defined for the mapper state machine. The setpoint computation block takes inspiration from the setpoint generation feature implemented in the PX4 position controller module; the more interesting part involves the test of the state machine in order to verify all the possible transitions combination.
Besides the setpoint mapper system has been intensively tested in flight environment with the ANT-1 multirotor UAV (Figure 2.6) and in this section some experimental results are reported. Furthermore, a couple of experiments have been conducted to show the new features involving tiltrotor (Figure 2.7) flight modes.

4.1 Test harness

The first approach to test the mapper state machine was the test harness. A test harness is a model block diagram that can be used to test, edit, or debug a Simulink model. In the main model, a test harness can be executed with a model component or the top-level model. It contains a separate model workspace and configuration set; it is associated with the main model and can be accessed via the model canvas. The test harness model is built around the component under test and it can be generated for:

- A model component, such as a subsystem, library block, or model block. The test harness isolates the component in a separate simulation environment.
- A top-level model, the component under test is a model block referencing the main model.

For the purpose of this thesis the first option has been used: it has been isolated the mapper state machine subsystem.
The test harness can be saved internally as a part of the model SLX or externally in a SLX file. In order to test all the possible transitions five test harnesses have

been defined and they are kept externally to the main model.

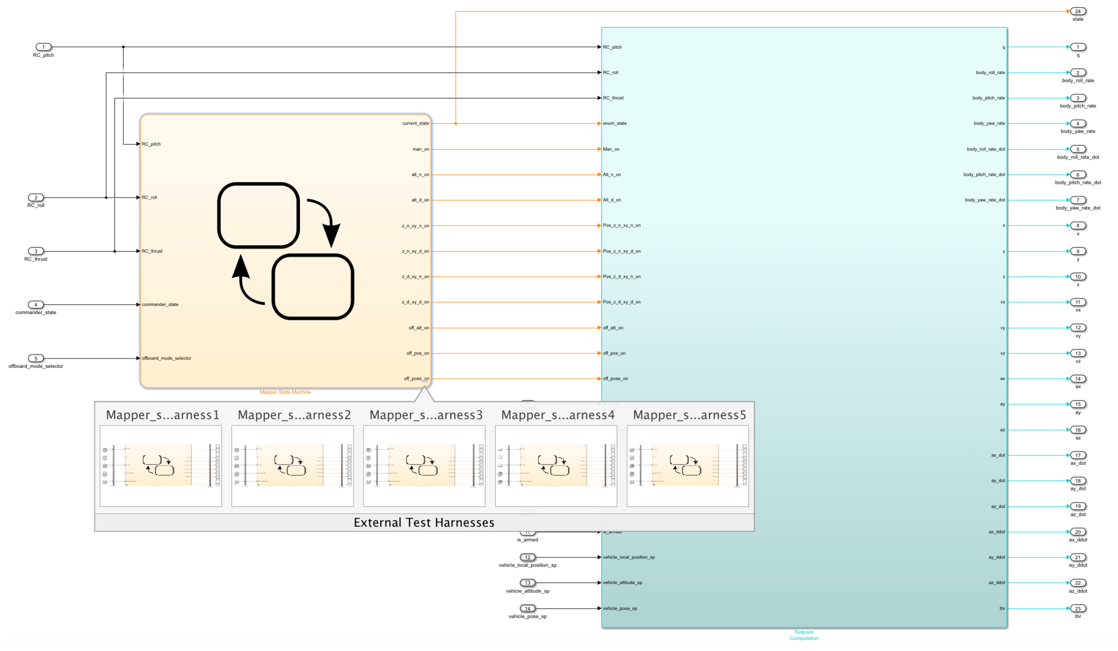


Figure 4.1: External test harnesses.

These test cases cover every possible transition of the state machine, in particular:

- the transitions involving the states within `MAN`, `ALT` and `POS` macro-states.
- the transitions involving the states within the `OFF` macro-state.
- the transitions involving the macro-states.

Figure 4.2 shows one of the three tests regarding the main flight modes, precisely the transitions between the response modes (Section 3.2) of the `POS` macro-state. The purpose of this test is verify, based on the sticks of the radio contoller and the information of the commander state, that all the transitions between position response modes are feasible. In order to build the test, a repeating sequence stairs has been considered as input signal for `RC_pitch`, `RC_roll`, `RC_thrust` and `commander_state`. In Table 4.1 are shown all the parameters used for this test.

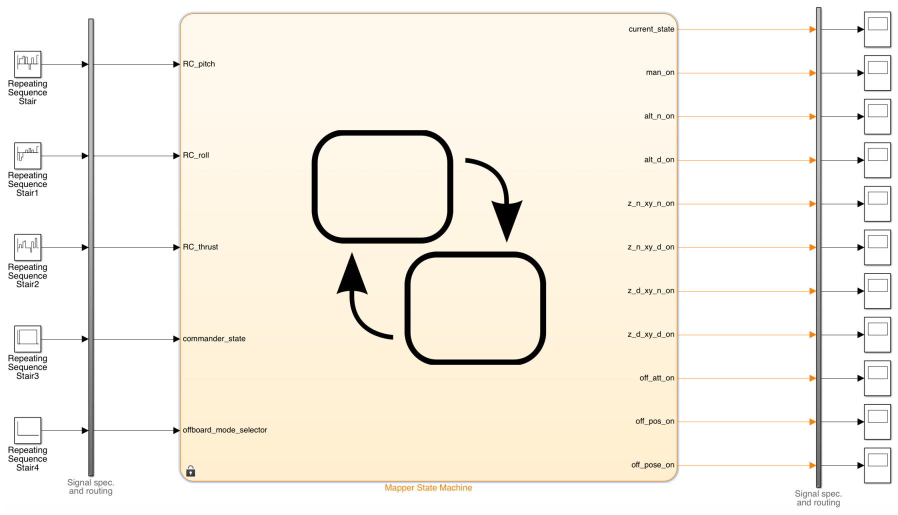


Figure 4.2: Test harness example.

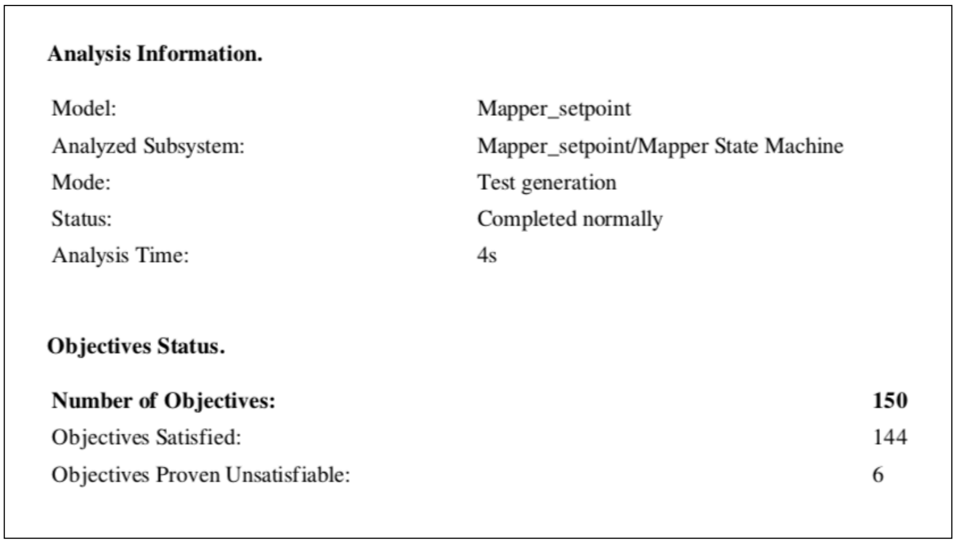
`RC_pitch`	`RC_roll`	`RC_thrust`	`commander_state`	time	`current_state`
0	0	0.5	0	0	0
0	0	0.5	2	0.004	3
0.35	-0.4	0.5	2	0.008	4
0.55	0	0.75	2	0.012	6
0	0	0.8	2	0.016	5
0.5	0.35	0.5	2	0.020	4
0	0	0.5	2	0.024	3
-0.4	0.45	0.5	2	0.028	4
0	0	0.35	2	0.032	5
0.4	0.35	0.75	2	0.036	6
0	0	0.5	2	0.040	3
0	0	0.9	2	0.044	5
0	0	0.5	2	0.048	3
0.49	0.52	0.5	0	0.052	0
0	0.01	0.5	2	0.056	3
0	0.5	0.8	2	0.060	6
0	0.7	0.5	2	0.064	4
0	0	0.5	2	0.068	3
0	0	0.7	0	0.072	0

Table 4.1: Test harness parameters table.

For this test purpose, the `offboard_mode_selector` is useless and it is trivially set to 0. The results of this test are consistent with the enumeration defined in Section 3.3.2.

4.2 Simulink design verifier

The second approach to test the mapper state machine was based on the Simulink Design Verifier tool [12]. This tool uses formal methods to identify hidden design errors in models. It detects blocks in the model that result in integer overflow, dead logic, array access violations and division by zero. It can formally verify that the design meets functional requirements. For each design error or requirements violation, it generates a simulation test case for debugging.
Simulink Design Verifier generates test cases for model coverage and custom objectives to extend existing requirements-based test cases. These test cases drive the model to satisfy condition, decision and custom coverage objectives.
The Design Verifier has been run for the mapper state machine and in Figure 4.3 the summary of the analysis has been reported.



Analysis Information.

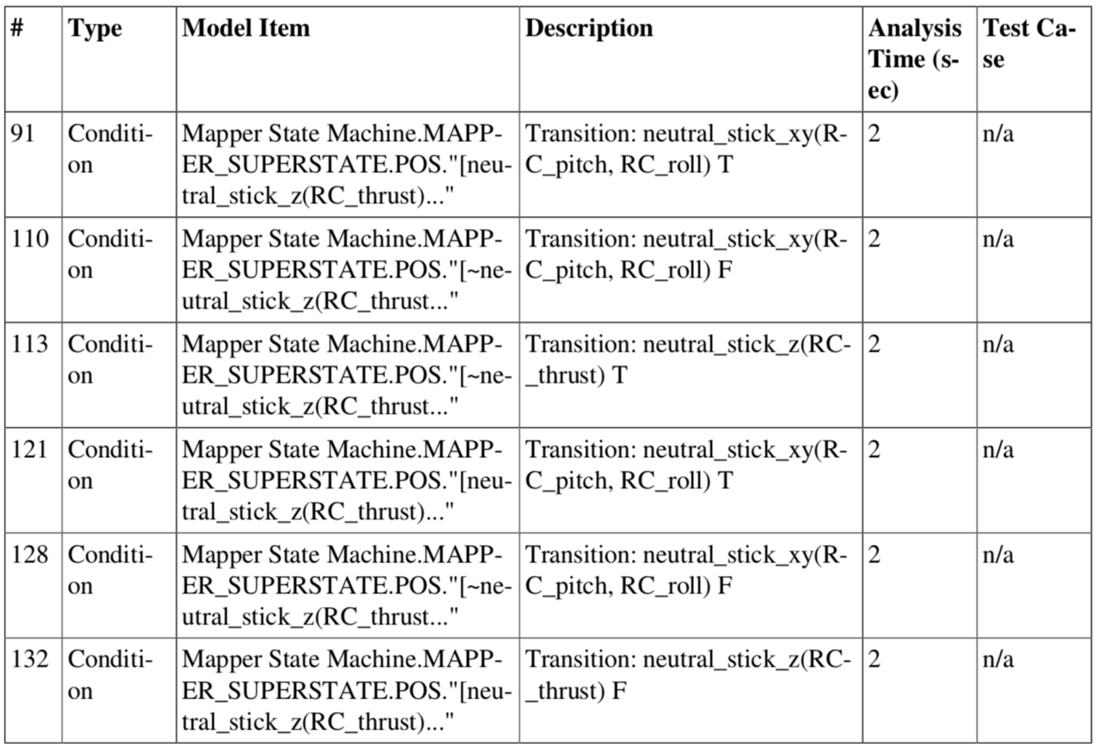
Model:	Mapper_setpoint
Analyzed Subsystem:	Mapper_setpoint/Mapper State Machine
Mode:	Test generation
Status:	Completed normally
Analysis Time:	4s

Objectives Status.

Number of Objectives:	**150**
Objectives Satisfied:	144
Objectives Proven Unsatisfiable:	6

Figure 4.3: Summary of the Design Verifier analysis.

The results of the analysis show that there are six objectives proven unsatisfiable: Simulink Design Verifier proved that there does not exist any test case exercising for these test objectives. This often indicates the presence of dead-logic in the model. Other possible reasons can be inactive blocks in the model due to parameter configuration or test constraints.
Figure 4.4 reports the six elements proven unsatisfiable. In fact a redundancy has been identified in the transition conditions within the `POS` macro-state. This redundancy comes to the fact that Stateflow assigns a priority for each transition arrow (Section 3.3.4 and Table 3.3) and there are some conditions that are never considered during the analysis because they are always true or false due to the evaluating order of the state machine. However, in the case under exam it was verified that, whenever a state transition condition is highlighted by the verification



#	Type	Model Item	Description	Analysis Time (s-ec)	Test Ca-se
91	Conditi-on	Mapper State Machine.MAPP-ER_SUPERSTATE.POS."[neu-tral_stick_z(RC_thrust)..."	Transition: neutral_stick_xy(R-C_pitch, RC_roll) T	2	n/a
110	Conditi-on	Mapper State Machine.MAPP-ER_SUPERSTATE.POS."[~ne-utral_stick_z(RC_thrust..."	Transition: neutral_stick_xy(R-C_pitch, RC_roll) F	2	n/a
113	Conditi-on	Mapper State Machine.MAPP-ER_SUPERSTATE.POS."[~ne-utral_stick_z(RC_thrust..."	Transition: neutral_stick_z(RC-_thrust) T	2	n/a
121	Conditi-on	Mapper State Machine.MAPP-ER_SUPERSTATE.POS."[neu-tral_stick_z(RC_thrust)..."	Transition: neutral_stick_xy(R-C_pitch, RC_roll) T	2	n/a
128	Conditi-on	Mapper State Machine.MAPP-ER_SUPERSTATE.POS."[~ne-utral_stick_z(RC_thrust..."	Transition: neutral_stick_xy(R-C_pitch, RC_roll) F	2	n/a
132	Conditi-on	Mapper State Machine.MAPP-ER_SUPERSTATE.POS."[neu-tral_stick_z(RC_thrust)..."	Transition: neutral_stick_z(RC-_thrust) F	2	n/a

Figure 4.4: Objectives proven unsatisfiable.

tool, the redundancies involve only a part of the state transition condition, but the overall condition is still guaranteed not to be affected by dead logic paths; in particular, the expressions related to such conditions could be rewritten in order to eliminate the redundancy and improve the efficiency of the system, but do not jeopardize the coverage of the state machine graph.
As a consequence of the result of this analysis it has been decided not to remove this redundancy in order to maintain clarity of the state transition condition expressions.

4.3 Flight testing

In order to test the setpoint mapper, two different machines have been used: ANT-1 and tiltrotor.

4.3.1 Test on ANT-1

Three different experiments were designed to test the management of the flight modes and the consequent setpoint generation based on the current response mode enabled. Figure 4.5 shows the different mapping of the throttle stick with respect to the manual and altitude modes. As long as the manual mode is enabled (the enumeration associated to the `mapper_state` is 0), the thrust is mapped as T^0 that corresponds to a re-scaling of the input thrust stick value.
As soon as the altitude mode is enabled, specifically a deflection of the thrust stick corresponds to the `ALT_D` response mode of the mapper state machine (`mapper_state` is equal to 2), the thrust stick is mapped as a velocity command in the vertical direction.

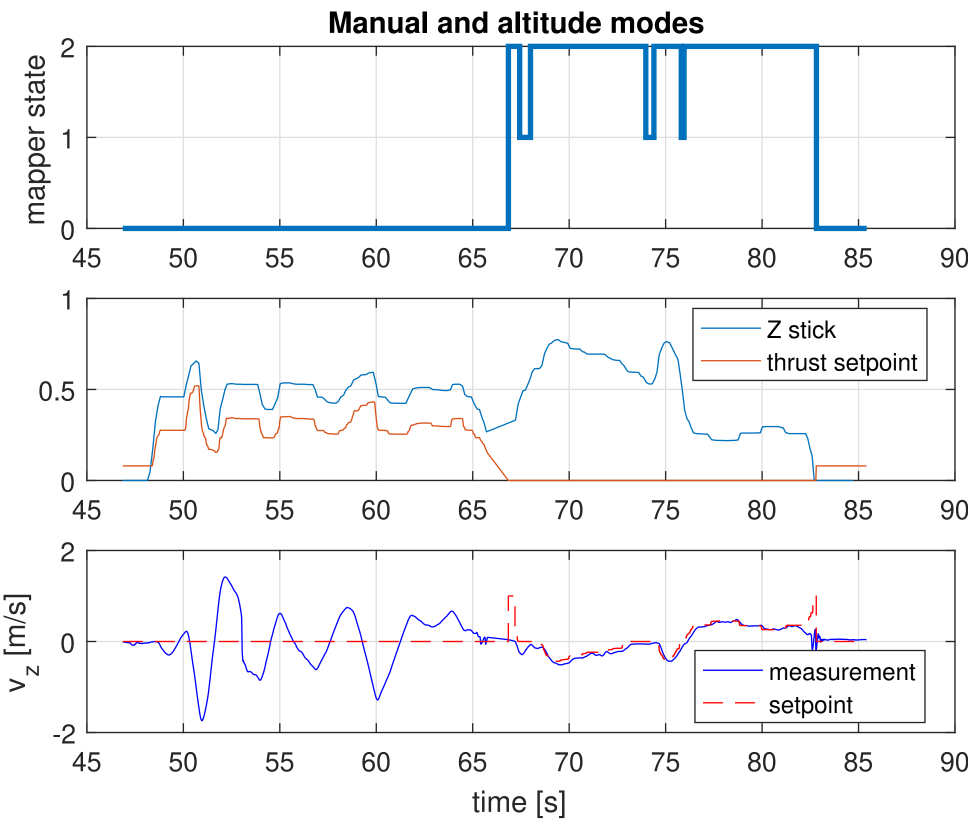


Figure 4.5: Experimental results ANT-1 test 1.

The `mapper_state` value equal to 1 corresponds to the `ALT_N` response mode which means the thrust stick within the deadband: v_z^0=0.

Figure 4.6 shows the switching between the `Z_N_XY_N` and `Z_N_XY_D` response modes within the `POS` macro-state focusing on the lateral stick mapping and the position hold capability. When the `Z_N_XY_D` response mode is enabled (the lateral stick is deflected and the `mapper_state` value is set to 4), the stick value is mapped as velocity command in the lateral direction.
As soon as the `Z_N_XY_N` response mode is enabled (the lateral stick within the deadband and the enumeration associated of the `mapper_state` is 3), the multicopter has the capability to hold the position at the time instant of the switching to the current mode (`Z_N_XY_N`). The position measurement at that time instant, is kept as position setpoint y^0.

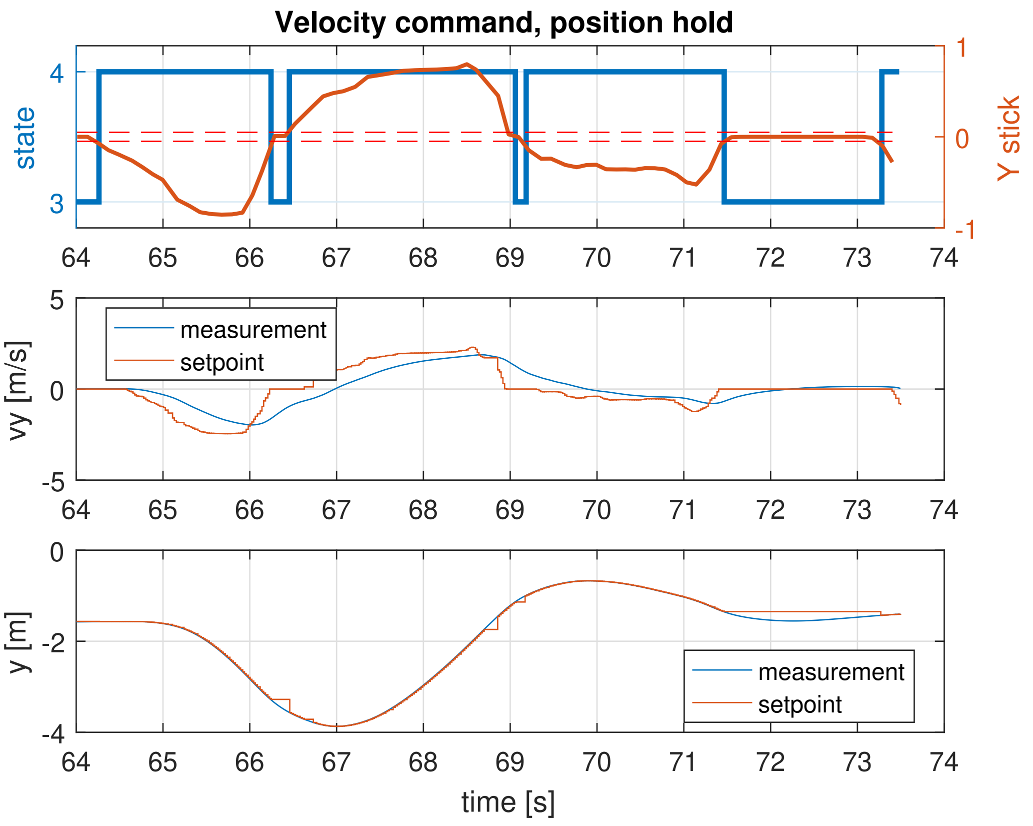


Figure 4.6: Experimental results ANT-1 test 2A.

Figure 4.7 describes a further experiment conducted to highlight the position hold capability.

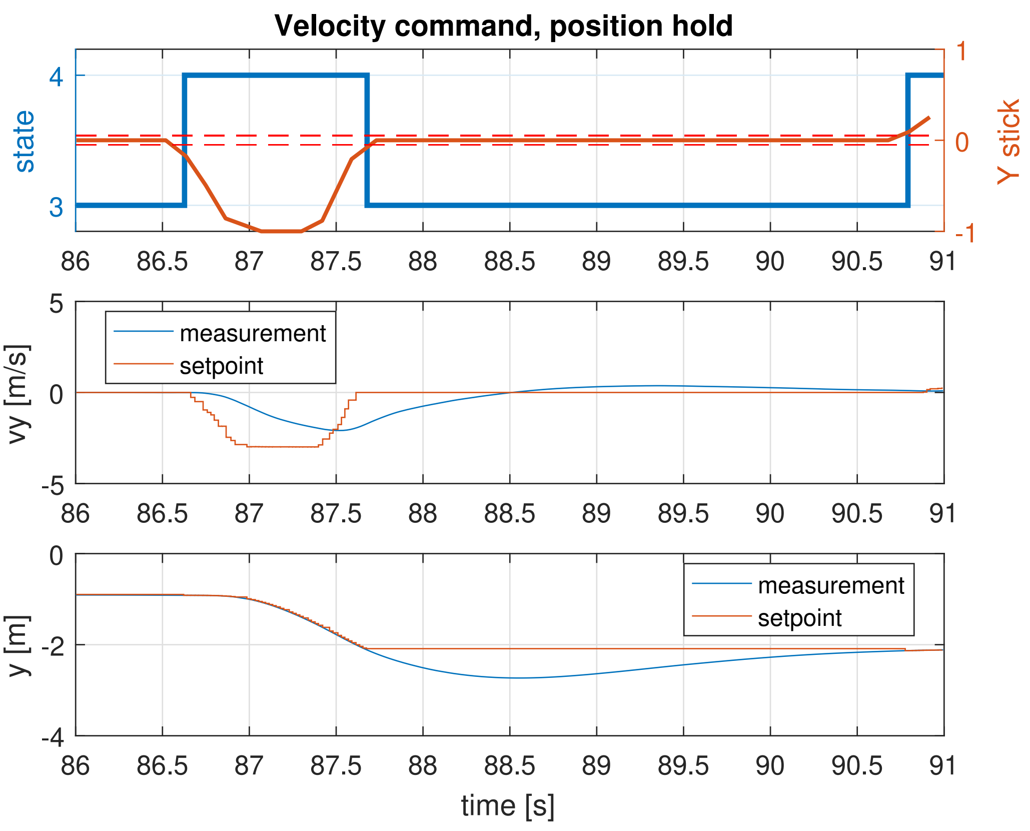


Figure 4.7: Experimental results ANT-1 test 2B.

When the lateral stick is centered, the v_y^0 is not considered anymore and the position hold capability is enabled. In fact the position measurement at the time instant of the switching to the Z_N_XY_N response mode (∼87.7s) is kept as a position setpoint y^0 till a deflection of the lateral stick occurs.

Figure 4.8 shows the experiment involving the switching between the offboard mode and position mode and vice versa (precisely `OFF_POS` and the `Z_N_XY_N` response modes). At time instant ∼43s the multicopter is in offboard position mode (the enumeration associated to the `mapper_state` is 7) which the position hold capability is enabled. After few seconds a series of position setpoints representing a trajectory is sent from the ground station and the vehicle according to the setpoint contained in the message, tracks the setpoint. Afterwards, a switching to a position flight mode occurs and the `Z_N_XY_N` response mode is enabled. As soon as this new mode is enabled, the position measurement is kept as a position setpoint y^0 corresponding to the measured position at that time instant since the position hold capability is enabled. Then, the offboard mode is enabled again. The vehicle also in this case holds the setpoint corresponding to the position measurement at the time instant of switching to the `OFF_POS` response mode as a position setpoint till a new series of setpoint commands is sent.

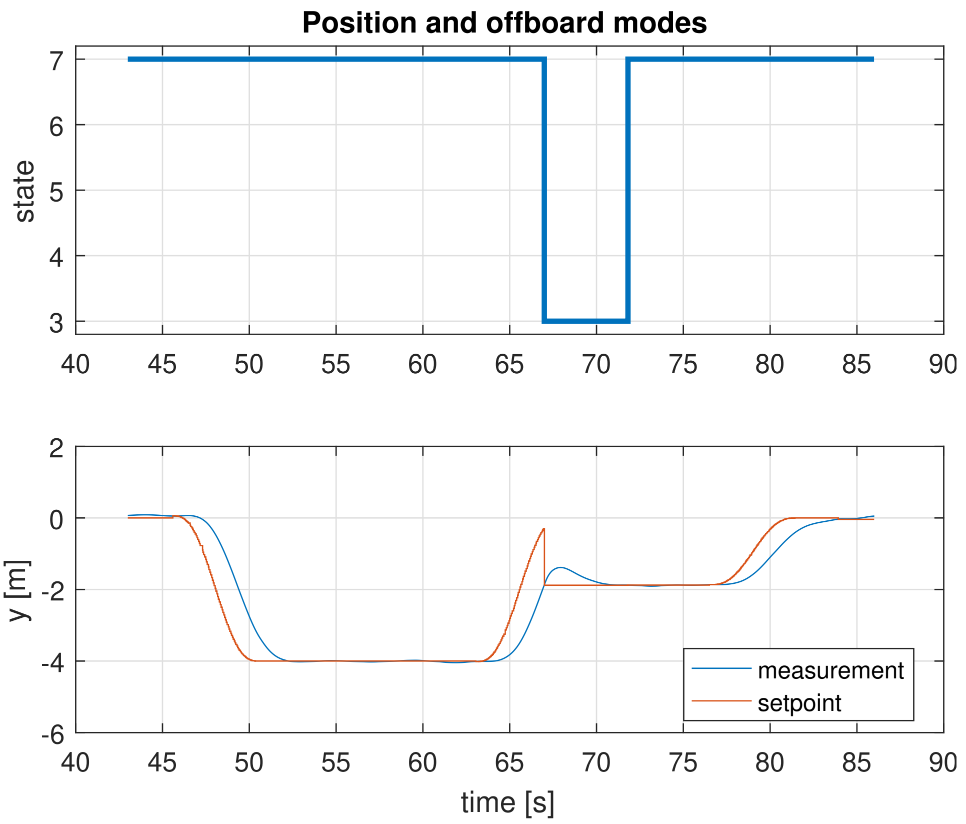


Figure 4.8: Experimental results ANT-1 test 3.

4.3.2 Test on tiltrotor

The setpoint mapper system was tested on tiltrotor. The developed system has allowed to implement new features in terms of flight modes it can perform. In fact, the tiltrotor was only able to perform trajectory based on the setpoint in offboard pose mode sent by the ground control station. In addition of the offboard modes, the setpoint mapper, together with a modification in the flight controller module, has enabled the management of the main flight modes: manual, altitude and position. The tiltrotor has an over-actuated structure and it is able to achieve maneuvers that simple quadrotors are not able to perform. The next experiments show the setpoint mapper system tested on the tiltrotor.

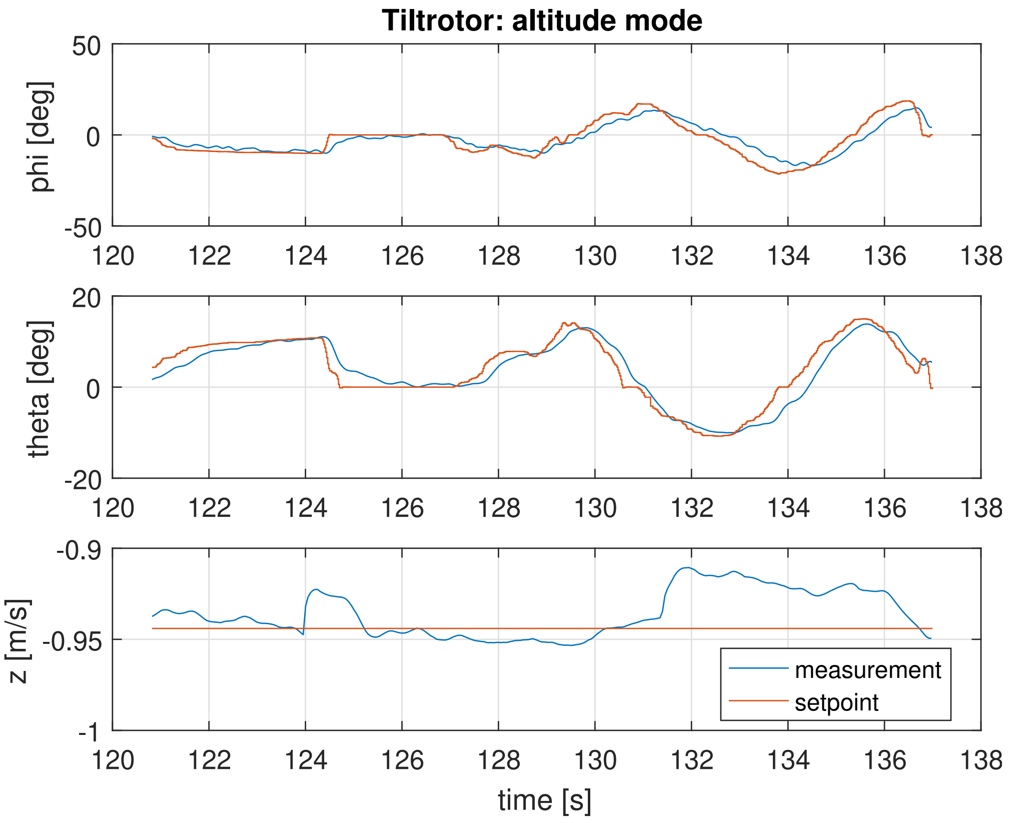


Figure 4.9: Experimental results tiltrotor test 1.

Figure 4.9 shows the altitude mode that corresponds to hovering, keeping non-zero roll/pitch angles (the tiltrotor has the capability to reach a full position/attitude decoupling). Depending on the deflection of the X, Y sticks of the Radio Controller, the tiltrotor performs its tilting capability; the ϕ and θ measurements follow the setpoints.

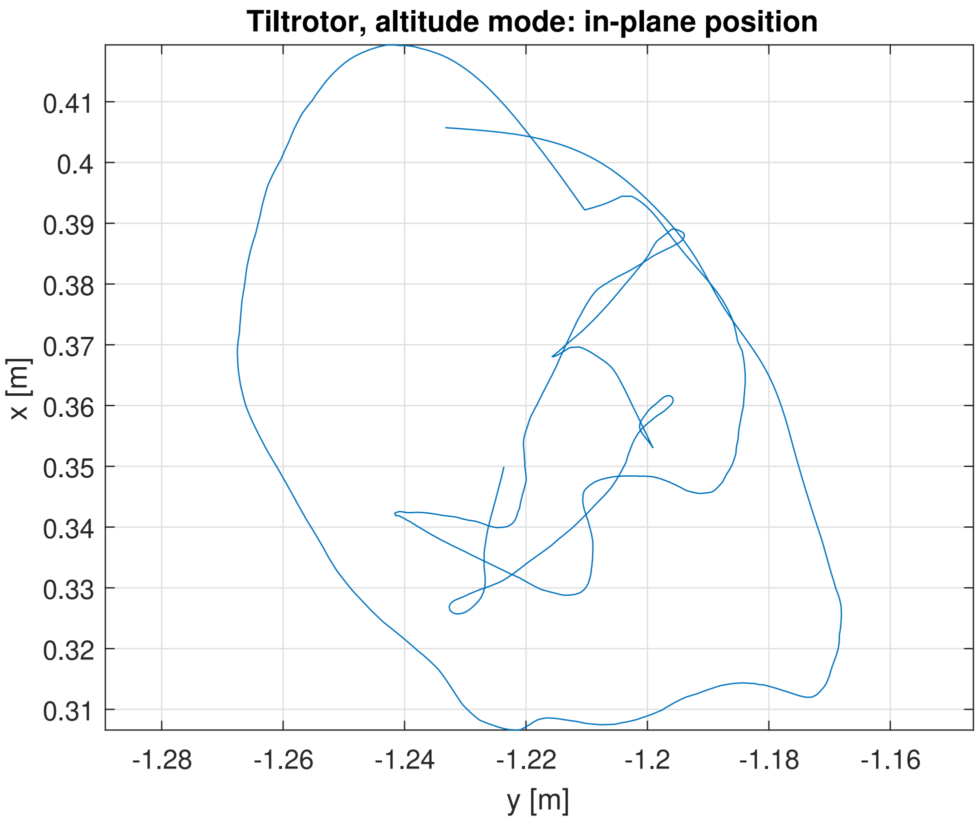


Figure 4.10: Experimental results in-plane tiltrotor test 1.

In Figure 4.10 is shown the in-plane position that tiltrotor performs according to the experimental results presented in Figure 4.9.

Figure 4.11 shows the switching between the `Z_N_XY_N` and `Z_N_XY_D` response modes within the `POS` macro-state focusing on the lateral stick. When the `Z_N_XY_D` response mode is enabled (the lateral stick is deflected and the `mapper_state` value is set to 4), the stick value is mapped as velocity command in the lateral direction and thanks to its tilting capability, the tiltrotor is able to maintain a zero roll angle even when maneuvering at high speed. It can be noticed that the roll angle reaches relatively large values in correspondence of braking maneuvers, when the velocity setpoint suddenly goes to zero: the flight controller could be improved to avoid this aggressive response. As soon as the `Z_N_XY_N` response mode is enabled (the lateral stick within the deadband and the enumeration associated of the `mapper_state` is 3), the setpoint velocity v_y^0 is set to 0.

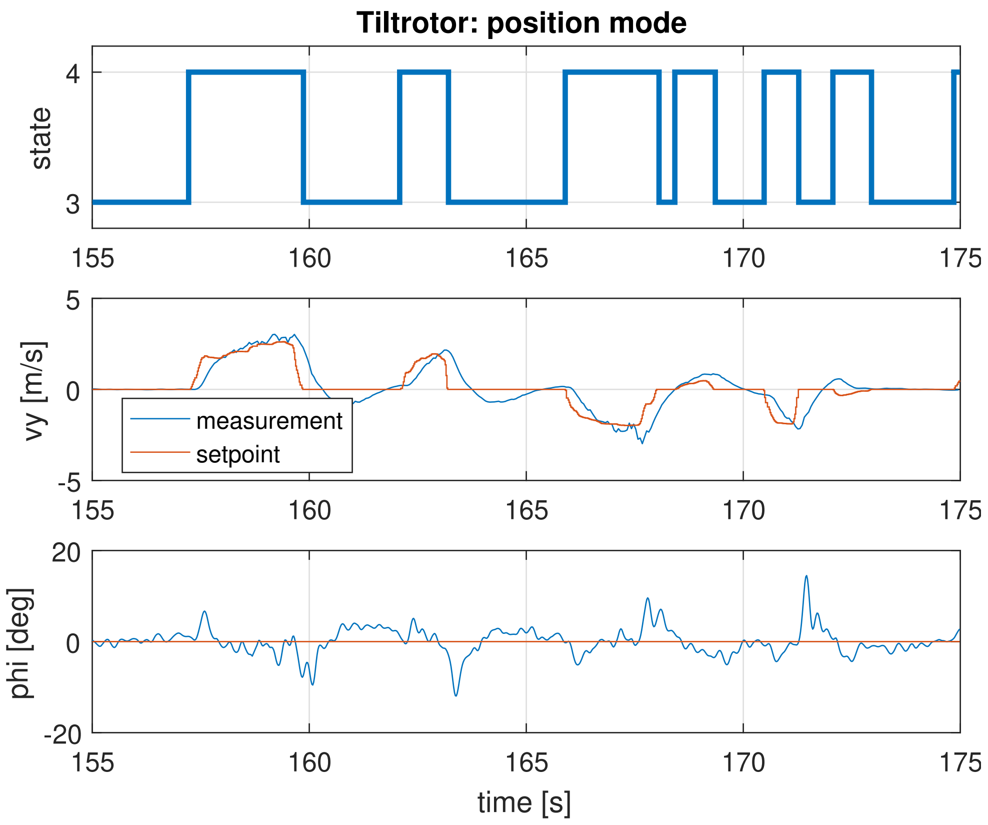


Figure 4.11: Experimental results tiltrotor test 2.

Chapter 5

Conclusions

This thesis has involved the design and implementation of a flight mode management and setpoint generation system and also the modification of the custom flight controller module.
The main objectives have been, on the one hand, to re-implement the setpoint generation system managed by the position controller module in the PX4 firmware in order to separate the setpoint generation from core PID control, on the other hand, to modify the interface of the custom flight controller module in order to extend the management of the main flight modes.
The developed system handles the setpoint generation based on the current flight mode enabled; it is also important to notice that it facilitates the integration of new flight modes.
In fact, from an architectural point of view, the choice of using a de-centralized architecture related to the setpoint mapper and the flight controller, makes the task of the integration of a new flight mode more straightforward.
Some possible developments are the following:

- refactor the setpoint computation block using a centralized architecture to manage the computation of the setpoint within the mapping of the current flight mode.

- improve the flight controller to avoid an aggressive response whenever the roll angle reaches relatively large values in correspondence of braking maneuvers, when the velocity setpoint suddenly goes to zero.

Bibliography

[1] P. Gattazzo. Nonlinear control of a tilt-arm quadrotor UAV. Master thesis, 2017.

[2] Micheli. Design, identification and control of a tiltrotor quadcopter uav. Master thesis, 2016.

[3] L. Meier. PixHawk Autopilot. `http://www.pixhawk.org`, 2008.

[4] Motive. Optitrack. `http://optitrack.com/products/motive`.

[5] Microsoft. Windows 10 Pro. `https://www.microsoftstore.com/store/mseea/it_IT/pdp/Windows-10-Pro/productID.320433200`.

[6] Canonical. Ubuntu 16.04 lts. `https://www.ubuntu.com`.

[7] Open Source Robotic Foundation OSRF. ROS. `http://www.ros.org`.

[8] L. Meier. Px4 Firmware. `https://github.com/PX4/Firmware`, 2008.

[9] GitHub.com. GitHub Px4 issues. `https://github.com/PX4/Firmware/issues`.

[10] MathWorks. Simulink. `https://www.mathworks.com/products/simulink.html`.

[11] MathWorks. Stateflow. `https://mathworks.com/products/stateflow.html`.

[12] MathWorks. Simulink design verifier. `https://mathworks.com/products/sldesignverifier.html`.