Politecnico di Milano
SCHOOL OF INDUSTRIAL AND INFORMATION ENGINEERING
Automation and Control Engineering

Quadcopter HOSM Control on PX4 Firmware Architecture

Supervisor
Luca Bascetta
Co-Supervisor
Gian Paolo Incremona

MSc Student
Stefano MONTEMURRO – 10560265

Academic Year 2020 – 2021

“Intelligence without ambition is a bird without wings”.

Salvador Dalì

Acknowledgment

I would like to thank Professor Bascetta and Professor Incremona for their complete supply at each step of this work, and for all their teaching lessons.

I am thankful to my mother and my sister for their support and to always encourage me toward the entire university course.

I am truly grateful to my girlfriend for her patience and her kindness.

A last though goes to my father, no more here, but always by my side.

Stefano Montemurro

Sommario

L'obiettivo di questa tesi è quello di implementare il controllo Higher-Order-Sliding-Mode (**HOSM**) utilizzando l'architettura **PX4**, open-source, che permette di simulare il volo di un **quadricottero** in un ambiente che replica quello reale, in particolare apportando alcune modifiche al suo firmware.
Questo tipo di azione di controllo richiede che il modello sia linearizzato mediante trasformazioni matematiche e trasformato nel modello canonico di *Brunovski*.
Le trasformazioni utilizzate sono la linearizzazione in **FeedBack** e in **FeedForward**, la prima linearizza il sistema rispetto ai suoi stati, mentre la seconda lo linearizza rispetto al riferimento che gli si vuol dare.
L'architettura che abbiamo deciso di utilizzare è PX4, (basata su *ardupilot* [1] e *dronecode* [2]), questa risulta a primo impatto complessa, ma il suo approccio **modulare** facilita le modifiche e l'aggiunta di estensioni volte all'introduzione di nuove funzionalità nel software senza influenzare il comportamento del sistema esistente.
La parte più difficile è stata capire come modificare la struttura di controllo poiché include tutti i comportamenti di sicurezza che non desideriamo modificare.
In questa tesi viene mostrata per prima una panoramica dell'architettura generale, in seguito la struttura interna di controllo (analizzata nel dettaglio implementativo), una panoramica del firmware, ed infine (in appendice) un breve e pratico tutorial su come apportare modifiche sul codice. Tale guida risulterà necessaria per permettere facilmente di comprendere come la nuova architettura di controllo è stata implementata nel codice.
Scendendo in maggiori dettagli il firmware utilizzato è basato sul sistema operativo a tempo reale (RTOS) **Nuttx**, completamente open-source.
Una delle caratteristiche del sistema PX4 è la "Ground Control Station" (**QGC**), la quale non è che un' interfaccia di controllo che permette di dare comandi da remoto, o in simulazione, e di analizzare i dati di volo.
I middleware che PX4 utilizza sono quelli **MAVLINK**, per la comunicazione offboard, mentre la messaggistica interna è quella **uORB**, questa alloca memoria globale sulla quale i diversi topic (messaggi) sono creati e poi condivisi tra i vari moduli del sistema.
Ultima peculiarità del sistema, è la possibilità di poter simulare il tutto con veicoli diversi e in vari ambienti di simulazione come **Gazebo** e **jmavsim**. Entrambi i simulatori consentono di avere un ambiente realistico in grado di riprodurre disturbi, dinamica dei sensori e dei motori, ed inoltre grazie alla tecnica di **lockstep**, consentono una completa sincronizzazione tra ambiente firmware e di simulazione.
La seconda parte è incentrata sulla implementazione effettiva del controllo HOSM, la linearizzazione del modello necessaria all'utilizzo di tale controllore, test di simulazione e risultati raggiunti.

Nell'analisi dei risultati viene mostrato il controllo non lineare HOSM, paragonato al controllore interno (**PID**) del sistema.
La nuova tecnica di controllo, a meno di un transitorio iniziale, ottiene performance migliori della tecnica già implementata da PX4, il che lascia la possibilità di continuare su questa strada facendo ulteriori test con algoritmi diversi di Sliding Mode e passando ad applicazioni reali.

Parole chiave: PX4, quadricottero, HOSM, FeedBack, FeedForward, Nuttx, QGC, MAVLINK, uORB, Gazebo, jmavsim, lockstep, PID.

Abstract

The goal of this thesis is to implement a Higher-Order-Sliding-Mode (**HOSM**) controller with model linearization strategy, on a **quadcopter**, using **PX4** architecture, an open-source firmware able to reproduce a real application environment in simulation.
This type of control action requires the model to be linearized by mathematical transformations that transforms the model into a *Brunowvki* canonical one. On this state space structure the "sliding mode" control is then applied.
The transformations used are the linearization in **FeedBack** and in **FeedForward**, the first linearizes the system with respect to the its states, while the second linearizes it with respect to the reference to be given.
The architecture used is PX4, (based on *ardupilot* [1] and *dronecode* [2]). The architecture, albeit seemingly complex at first, is designed to be modular, which turn eases modifications and the introduction of extensions in the firmware without affecting the entire system behaviour.
The most difficult part was to understand how to modify the control architecture since it includes all the failsafe behaviours that we do not desired to modify.
In this thesis is first showed an overview of the general architecture, then the control strategy used by PX4, and its firmware description. Lastly the appendix a brief and practical tutorial on how to make code modifications is showed, it should be useful to easy understanding of how the new control architecture is implemented in the code.
Going into more details, the firmware is based on the real-time operating system (RTOS) **Nuttx**, completely open-source.
One of PX4 features is the Ground Control Station (**QGC**), which is nothing more than a control interface that allows to give offboard commands, and analyze the flight data.
The middleware that PX4 uses are those **MAVLINK**, used for offboard communication, and the internal messaging system is **uORB**, this allocates global memory on which different topics (messages) are created and then shared between the various system modules.
The last peculiarity of the system is the possibility to simulate the whole architecture with different vehicles and in various simulation environments such as **Gazebo** and **jmavsim**. Both the simulators allows to have a realistic environment able to reproduce disturbaces, sensors and motors dynamic, and another feauture is the **lockstep** simulation technique, that allows to completely synchronize the firmware and the simulator.
The second part is focused on on the mentioned control mode implementation, simulation tests and achieved results. In the analysis of results the nonlinear HOSM control

compared to the internal controller (**PID**) of the system is shown.
The new control technique, apart from the initial transient, has better performances than the one already implemented by PX4, which leaves the possibility to continue on this path by making further tests with different sliding mode algorithms and passing to real applications.

Key words: PX4, quadcopter, HOSM, FeedBack, FeedForward, Nuttx, QGC, MAVLINK, uORB, Gazebo, jmavsim, lockestep, PID.

Contents

List of Figures

List of source codes

List of Tables

Chapter 1

Introduction

Unmanned and small aerial vehicles have started to change the world. Their applications are the most various ones such as: aerial photography, entertainment, geographic mapping, shipping and delivery [4], military use, etc...[5], [6], [7].
An example of a quadrotor can be visualized in Figure 1.1.

Figure 1.1. Dji Mavic Mini [3]

The market has grown a lot in the last years and drone applications vary in number of rotors, dimensions, and cost (from the cheapest one around 30$ to the most expensive that cost thousands of dollars).
The different models are for example helicopter (1 only rotor), multicopter (quadrotor, hexarotor etc...), fixed-wing drones, micro drones [8] and others [9]. The growing use of this type of vehicles requires every day a better flight control strategy.
For purposes like simple entertainment it is not required to be so precise, but for military use, geographic mapping, aerial photography (for example if a small drone has to pass through narrow spaces like in a cave) the performances of the control strategy have to be very high.
The most used control strategy is based on simple PIDs structures, often used in cascade architecture [10], it is the oldest and easiest way to control every type of vehicle, but other approaches exist like optimal control LQG (Linear Quadratic Gaussian) [11], [12], and non-linear control strategies (SMC: Sliding Mode Control).
This last type of controller is the newest one and hopefully it will bring a strong improvement on the control performances, but the tests done until today are not so many.

Objective of this thesis

It is known that sliding mode has the so called “chattering” problem, highlighted for fast dynamics systems, i.e. the use of this control technique causes the actuators to continuously go “up and down” trying to bring the “sliding surface” error to zero.
The controller used in this work will be the HOSM (Higher Order Sliding Mode) controller, based on the SMC (Sliding Mode Controller) control theory, the main difference respect the classic SMC is the idea to bring the “sliding surfaces” on higher order derivatives, this will keep the control action smooth and should reduce the chattering.
Some tests were already be done in simple simulations based on Matlab/Simulink environment, where very good results were achieved. This encouraged to continue on this direction.
The interest of this work was based on the idea of having a controller able to perfectly follow whatever trajectory, this controller could is be the HOSM, but it was needed to test it in an environment that can well reproduce a realistic one.
In order to do so the controller was reproduced in “C++” code (also known as cpp) and then implemented on the PX4 firmware, designed for professional drone application and code development. Moreover it gives the possibility to test the code in simulation environments such as Gazebo, jmavSim.
Gazebo it is able to generate an ambient that considers most of the non-linearities presented in the real world, and to reproduce motor and vehicle dynamics (considering for example not only the body inertia but also the one of motors and sensors mounted on the vehicle.
Summarizing the steps of this thesis will be:

- The study of the firmware to be modified, its features, its control architecture, its middleware etc...
- Create a short and brief tutorial that explains how to add new “message, module and task” to the firmware
- Reproduce the control strategy designed in Matlab environment in a one compatible with the firmware (c++ language)
- Implement the controller, interfacing it with the existing PX4 one, and keeping intact all the already existing features
- Test the code in Gazebo simulation environment and verify its performance

The achieved results are promising for future work, in particular can be seen that the new control technique, compared with the one used by default from PX4, performs better. This will open new possibilities for future works with the aim of fine tuning the control strategy and make new tests on real world with a real drone.

1.1 Organization of the thesis

The work of this thesis is organized in seven chapters, here briefly summarized.

Chapter 1: an introduction about the drone development, applications and the main perspective in the future.

Chapter 2: an general overview on the PX4 world, its main functionalities, various *Flight Modes* available, the compatible hardware and the operating system.

Chapter 3: a description of the PX4 firmware, how it is structured, and the main modules description. Then a description of the *middleware* used by PX4 for topic communication inside the firmware and from the firmware to the external components, lastly an introduction of the simulation environment.

Chapter 4: a detailed description of whole flight control structure, and the main parts of its implementation into the firmware.

Chapter 5: description of the $HOSM$ control and the linearization technique, their implementation into the Firmware and some features added to the controller in order to interface it with the already existing flight control structure.

Chapter 6: description and results of the testing phase highlighting the obtained performances and a comparison with the default PX4 control strategy.

Chapter 7: a short summary of the obtained results with some suggestions for further development.

- **Appendix A**: a low level tutorial, necessary if the reader wants to have an idea of how the controller was implemented in the PX4 firmware.

Chapter 2

PX4 Overview

This chapter contains a general overview of the whole architecture (Figure 2.1), most of its contents are treated from an high perspective spared for the ones used in the thesis.
The aim is give to the reader the capability to better understand the next chapters and to have a general background of the architecture.

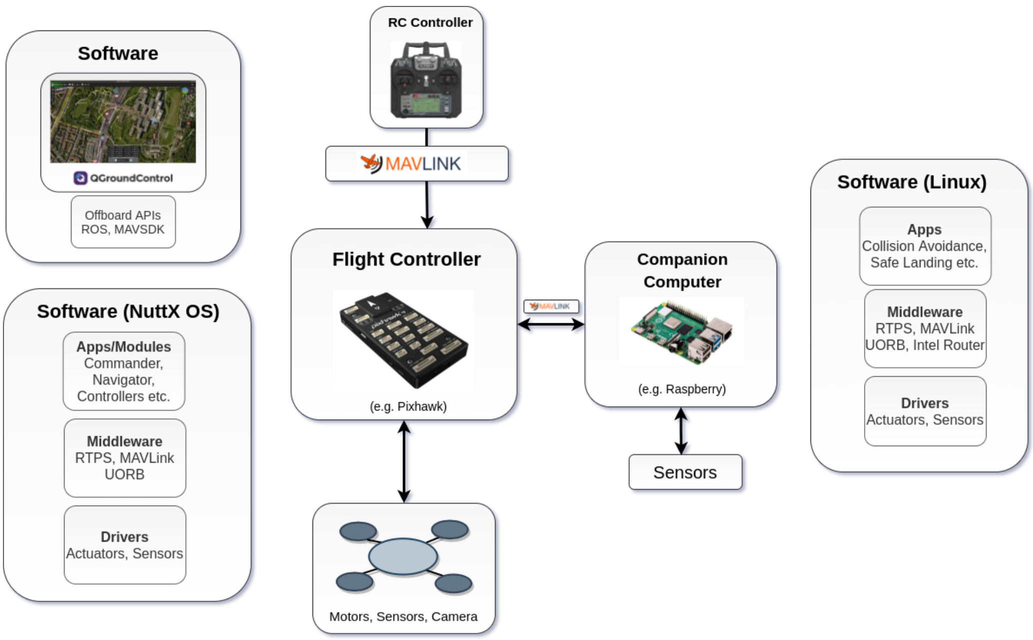


Figure 2.1. PX4 Architecture General Scheme

2.1 System description

Any autonomous vehicle project (in this work a quadcopter) is essentially composed by the following parts:

- **Hardware**: combination of sensors, controllers and output devices allowing the drone to sense the external environment and to decide what to do according to the current scenario. It essentially consists of all physical components the drone is made of.

- **Firmware**: the code running on the controller which guarantees reliability, stability, and provides features leveraging inputs from the sensors, elaborating them through a processing unit, and sending outputs to other peripherals such as *ESCs*, motors, servos, camera, etc. The firmware allows to integrate additional hardware components implementing their drivers.

- **Software**: it is the interface to the controller often called Ground Control Station (*GCS*). Software can run both on PCs and/or mobile devices. A *GCS* allows users to set-up, configure, test, and finely tune the vehicle setup, acting on the firmware in simple and straight-forward fashion. Advanced packages or add-ons allow autonomous mission planning, operation, and post-mission analysis. This is generally used by the operator both to program the mission and to analyze log files.

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

2.2 Hardware

Pixhawk 4 (Figure 2.2) is an advanced autopilot designed and made in collaboration with Holybro [13] and the PX4 team. It is optimized to run PX4 v1.7 and later, and it is suitable for academic and commercial developers.
It is based on the Pixhawk-project FMUv5 open hardware design and runs PX4 firmware on the NuttX OS (see Section 2.3).

Figure 2.2. PX4 Hardware

This hardware was used in order to make tests in *HITL* (see Section 3.6.2) and it will be probably used for real application in future works.
Some more details can be found in the online developer guide [14].

2.3 NuttX

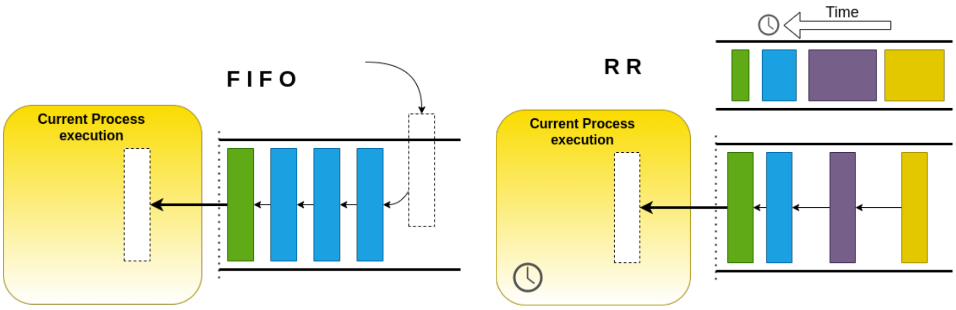
NuttX is a real-time operating system compact and efficient for embedded applications; it supports different architectures from 8 to 64 bit, the main programming language used is C/ C++, the reference it implements are POSIX and ANSI standards (these standards are used for API (Application Programming Interface) definitions), lastly it is open-source (BSD license)[15].
NuttX is the primary *RTOS* (Real-Time-Operating-System) for running PX4 on a flight-control board, serving as a collection of various features packed as a library. It is executed in two conditions only: when the application is asking a piece of the NuttX library code, and when an interrupt occurs.
Its most relevant features are:

- Fully preemptible
- FIFO, Round-Robin scheduling
- Realtime, deterministic, with support for priority inheritance
- System logging
- Over 30 supported platforms
- Around 20 supported file systems

All RTOSs support the notion of 'tasks' as NuttX does; a task is the RTOS's moral equivalent of a process. Each task is represented by a particular data structure called task control block or TCB, which is unique. Nevertheless, all tasks share a single address space [16].
Moreover, each task/thread has a fixed-size stack, and there is a periodic task which checks whether all stacks have enough free space left. The stack is that portion of the memory used to manage the automatic allocation of memory; the stack may contain the CPU registers during context switch (see Section 2.3.1) or other useful data.

2.3.1 Scheduling Policy

In order to be a real-time OS, the OS must support strict priority scheduling: the highest priority thread runs (i.e. given to the CPU). Tasks of equal priority are scheduled with SCHED_FIFO. FIFO stands for "First-In-First-Out", also known as "First-Come-First-Served" (FCFS), it is the simplest scheduling algorithm, where processes are added to a queue and executed with respect to the entry order.
As it is possible to read from the official NuttX website [15], NuttX supports one additional real-time scheduling policy: SCHED_RR, where RR stands for "Round-Robins". Here the scheduler assigns a fixed time unit per process, and cycles through them. If a process completes within that time-slice it gets terminated, otherwise it is rescheduled after giving a chance to all other processes. Thus, NuttX supports time-slicing: if a task with SCHED_RR scheduling policy is running, then when each time slice elapses, it will give up the CPU to the next task that is at the same priority. Both the scheduling policies are shown in Figure 2.3.



(a) FIFO (b) RR

Figure 2.3. Scheduling policies

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

2.4 PX4 Opens-Source firmware

Autonomous drones are gaining a lot of interest both in private companies and in universities. This lead to a subdivision of the market in two different categories of systems: open-source and private. This thesis will be based on an open-source system, focusing on an open-source firmware which is the already mentioned PX4 one; its key features are:

- It is able to control many different vehicle frames/types, including: aircraft (multicopters, fixed wing aircraft and VTOLs (Vertical Take-Off and Landing)), ground vehicles and underwater vehicles.
- Great choice of hardware for vehicle controller, sensors and other peripherals.
- Flexible and powerful flight modes and safety features.
- Optimised APIs and SDKs (Software Development Kit) for developers working with integrations and customizations.
- Designed to be deeply coupled with embedded microcontrollers, in particular concerning vision for autonomous capabilities.
- The main feature is that all the airframes share a single codebase (including other robotic systems like rovers, submarines, boats etc.) [17].

PX4 is a core part of a broader drone platform that includes the *QGroundControl* [18] ground station, Pixhawk hardware, *MAVSDK* [19] for integration with companion computers [20] (like a Raspberry Pi which could provide advanced features like object avoidance and collision prevention), cameras and other hardware using the *Mavlink* [21] protocol. The whole project is supported by *dronecode* [2] .

2.5 QGroundControl

QGroudControl (*QGC*) is the dronecode control station (Figure 2.4). It can be used to flash the firmware into the hardware, setup the vehicle, change parameters, get real-time flight information and create and execute fully autonomous missions.

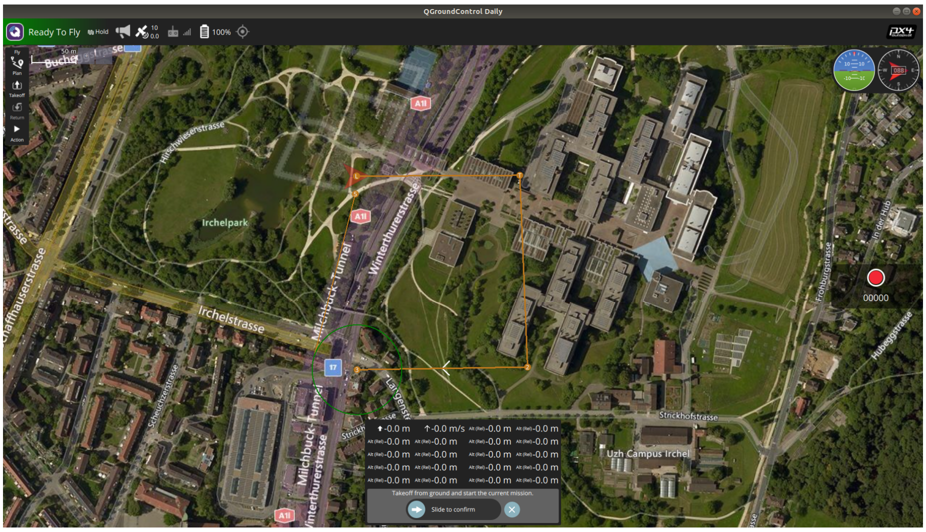


Figure 2.4. QGroundControl mission planning

The control station also allows to give manual commands from a console shell, to log and plot flight data online during the mission, then once the flight is ended, allows to collect the data for further analysis.

2.6 PX4 Flight Modes Overview

Flight modes define how the autopilot responds to remote control inputs, and how it manages vehicle movement during fully autonomous flight (Figure 2.5).
The modes provide different types/levels of autopilot assistance to the user (pilot), ranging from automation of common tasks like take off and landing, to mechanisms that make it easier to regain level flight, guide the vehicle to a fixed path or to a position, etc. [22], [23]. The main feature of flight mode is that an advanced user can easily add new modes and tasks (see Section 4.4) to the vehicle, adding for example a customized trajectory like an helical one.

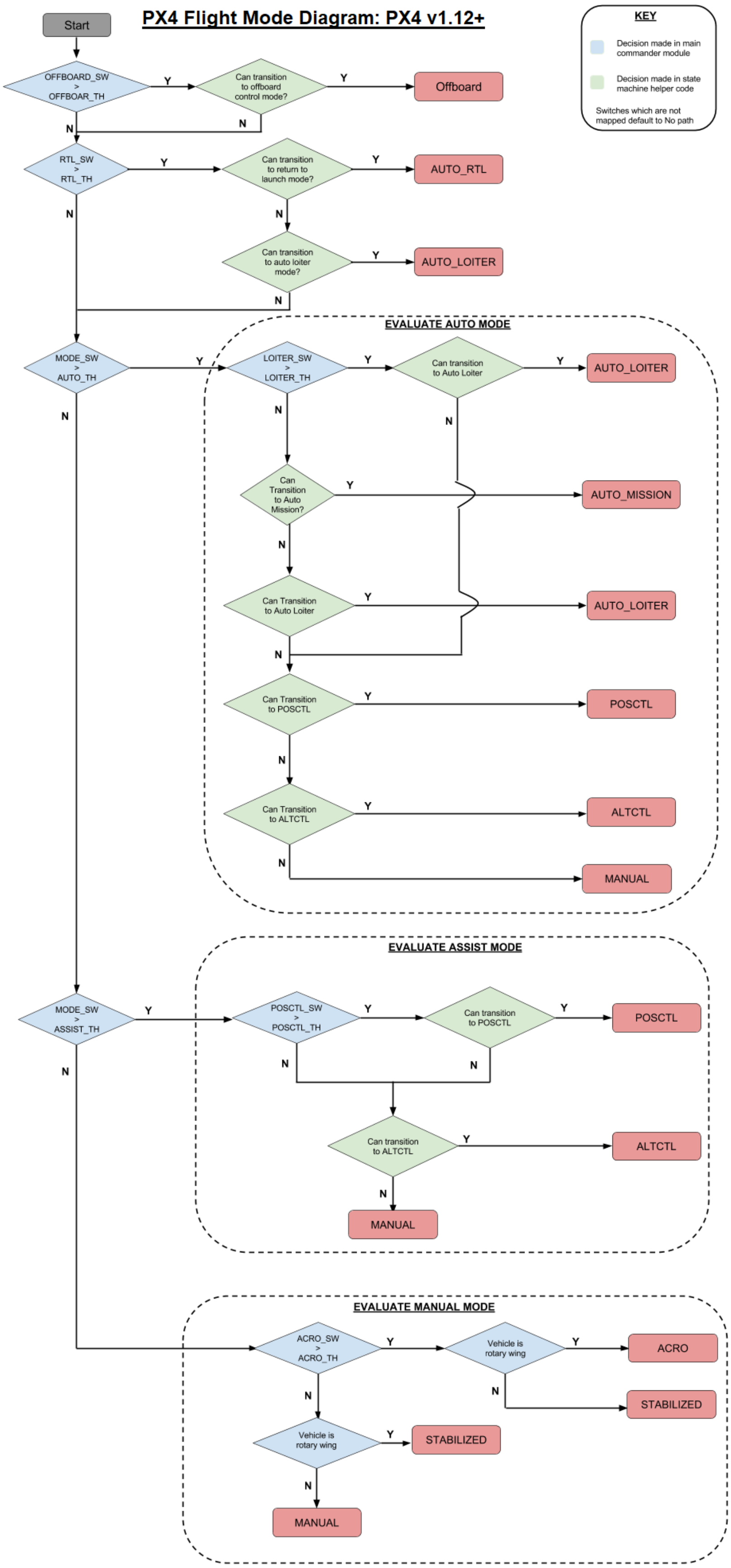


Figure 2.5. Flight Mode switching

The next subsections describe some of the available flight modes, in particular the ones related to multicopters.

2.6.1 Manual modes

"Manual" modes are those where the user has direct control over the vehicle via RC (Radio Commander), (or joystick). Vehicle movement always follows stick movement, but the level/type of response changes depending on the mode. Some modes provide direct passthrough of stick positions to actuators, while others (for beginners pilots) are less responsive to sudden stick-position changes [23]:

- **MANUAL/STABILIZED:** throttle is passed directly to the output mixer (mixer will be explained in Section 4.8). Pilot's inputs are passed as roll and pitch angle commands and a yaw rate command. In this mode the autopilot controls the attitude, regulating the roll and pitch angles to zero when the RC sticks are centered. Nonetheless, the autopilot cannot control the vehicle position, therefore drone could shift position due to wind forces [22].

- **ACRO:** as in manual mode, the throttle is directly passed to the output mixer (see Section 4.8). Contrary to the previous mode, pilot inputs are introduced as roll, pitch, and yaw rate commands to the autopilot, while in the previous case roll and pitch were introduced as angle commands. Angular rates can be controlled by the autopilot, but not the attitude. As a consequence, if the RC sticks are centred, the vehicle will not level-out. This allows the multirotor to perform acrobatic moves such as a complete inversion [22].

2.6.2 Assisted modes

"Assisted" modes are an improvement with respect to manual modes offering "automatic" assistance in specific situations, like holding position/direction against the wind. One of the advantages of assisted modes is the capability of restoring controlled flight [23]:

- **Altitude Control (ALTCTL):** roll, pitch and yaw inputs are analogous to the STABILIZED mode. Throttle inputs indicate going up or down at a predetermined maximum speed. Centred Throttle holds altitude steady. The autopilot only controls altitude, hence wind can change the vehicle position [22].

- **Position Control:** roll controls left-right speed, pitch controls front-back speed relative to the ground and yaw controls the yaw rate similarly to MANUAL mode. Throttle handles climbing/descending rate as in ALTCTL mode. As a consequence vehicle position is held steady by the autopilot against any wind disturbances [22].

2.6.3 Autonomous modes

"Auto" modes are those where the controller requires little to no user input. Autonomous flight mode includes "submodes" like: takeoff, land, hold, return and others [23]:

- **AUTO LOITER:** the multirotor hovers maintaining the current position and altitude [22].
- **AUTO RTL (return to land):** the multirotor returns following a straight line either on the current altitude or at a pre-defined one [22].
- **AUTO MISSION:** the aircraft performs the programmed mission sent by the GCS. If no mission is received, aircraft will LOITER at current position, unless battery drops; in that case, depending on the failsafe mode, it will either return to the home position or simply land wherever it is [22].

Note: The system autonomously knows when to switch to a certain flight mode based on commands received by the user, and measurements reads from sensors. For example when a planned mission is launched from QGC, PX4 control architecture knows that has to use the autonomous flight mode and will enable the relative internal control strategy (see Chapter 4).

2.7 FastRTPS

The PX4-FastRTPS Bridge adds a Real Time Publish Subscribe (RTPS) interface to PX4, enabling the exchange of uORB messages between PX4 components and (offboard) Fast RTPS applications (including those built using the ROS2/ROS frameworks) [24].
RTPS should be considered when real-time or time-critical informations need to be exchanged between the flight controller and the off-board components (perception computer) in a reliable manner.

Chapter 3

PX4 Firmware description, Middleware, Communication System & Simulation Environment

This chapter will briefly describe the main firmware components present on PX4 architecture, how they works, the purpose of each of them, the scheduling of the system, etc.
Then will be also described the middleware, the communication system, and the simulation environment.

Note: a detailed low level code development tutorial, and relative online documentation can be find in the appendix at the end of this thesis (see Appendix A).

3.1 PX4 Software Architecture

The diagram in Figure 3.1 provides an overview of the building blocks of PX4. The top part of the diagram contains middleware (see Section 3.2.4) blocks, while the lower section shows the components of the flight stack (see Chapter 4).
Modules communicate with each other through a publish-subscribe message bus named uORB (see Section 3.4). The use of the publish-subscribe scheme means that:

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

3.1.1 Runtime Environment

PX4 runs on many different operating systems providing POSIX-API (NuttX, macOS, Linux or QuRT). For our application Linux will be the used OS. The inter-module (see Section 3.2.3) communication relies on uORB and it is based on shared memory. A single address space is used to run the entire PX4 middleware, in fact memory

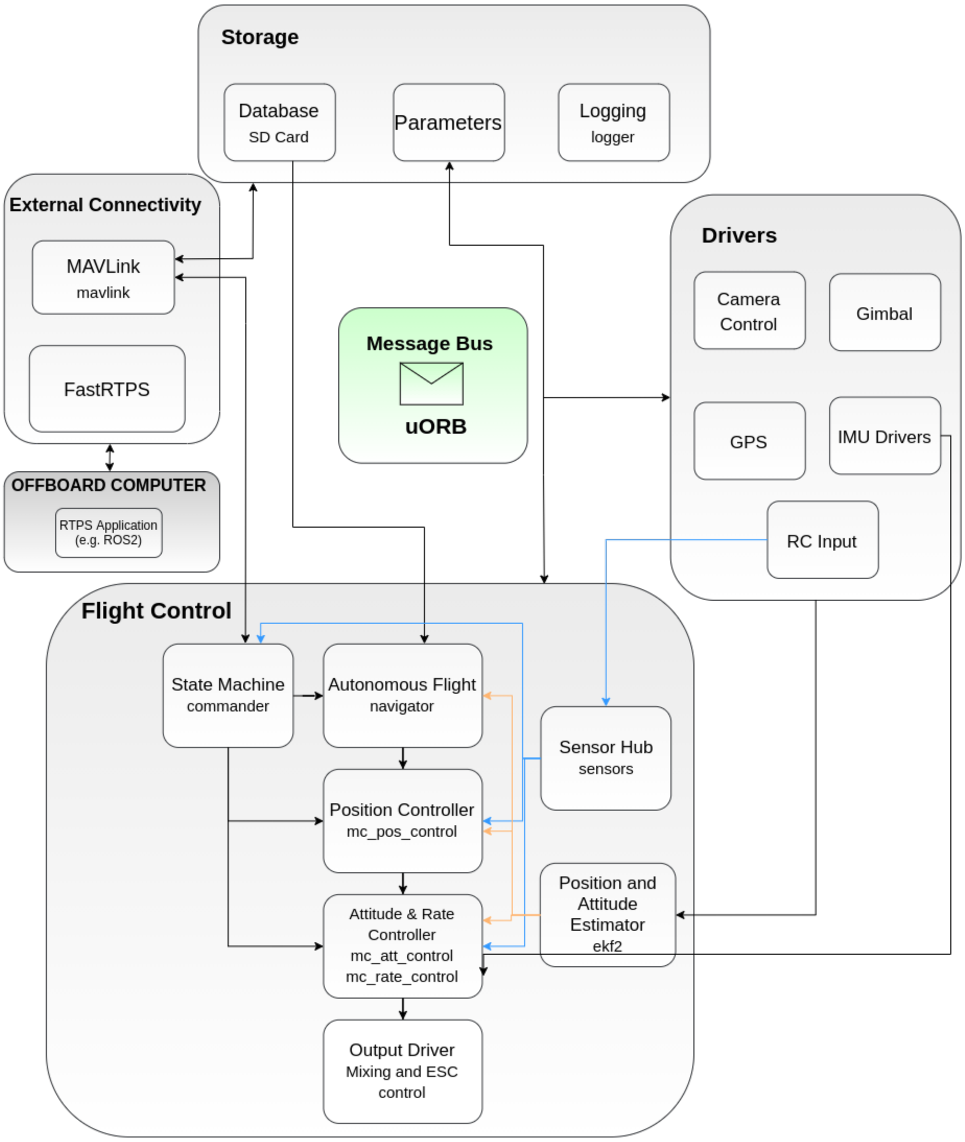


Figure 3.1. Firmware Architecture scheme

is shared between all modules. Nevertheless, the system is designed such that with minimal changes it would be possible to execute each module/application in a private address space [25].
Linux also have some form of real-time scheduling (e.g. FIFO as seen in Section 2.3.1). All the tasks must behave co-operatively as they cannot interrupt each other. Multiple work queue tasks can run on a queue, and there can be multiple queues. A work queue task is scheduled by specifying a fixed time for loop cycle, or via uORB topic update callback, meaning that the task will be triggered by new incoming data, “data driven”.

3.1.2 Firmware components

This subsection introduces the main PX4 modules (see Section 3.2.3) and their functionality, the only aim is to give some general knowledge of them before entering in details in next Chapter (see Chapter 4). The main modules are:

- **commander**, it contains the state machine for mode switching and failsafe behaviors.
- **navigator**, it is responsible for autonomous flight modes. This includes missions (read from dataman), takeoff and RTL. It is also responsible for geofence violation checking.
- **flight_mode_manager**, it implements the setpoint generation for all modes. It takes the current mode state of the vehicle as input and outputs setpoints for controllers.
- **mc_controller**, it comprehends three different modules: position, attitude and rate. The whole architecture is responsible of trajectory following for autonomous flight and for stabilizing the manual one.
- **mixer**, it's the interface between the control action given by the mc_controller and the actuation values, it remaps the control commands to each rotor according to the used vehicle geometry.
- **sensors**, it is central to the whole system. It takes low-level output from drivers, turns it into a more usable form, and publishes it for the rest of the system.
- **EKF2**, it estimates attitude and position using an Extended Kalman Filter. It is used for multirotors and fixed-wing vehicles.
- **drivers**, they handle the interface between sensor hardwares and PX4 firmware architecture.
- **mavlink**, it implements the MAVLink (communication middleware, see Section 3.3) protocol and communicates with the system via uORB. Streams are used to send periodic messages with a specific rate, such as the vehicle attitude. When starting the mavlink instance, a mode can be specified, which defines the set of enabled streams with their rates.
- **simulator**, it interfaces software in the loop simulator (SITL) such as jMAVSim or Gazebo with the firmware, via MAVLink messaging.

Other informations can be found on the online user guide [26].

3.2 Firmware Structure

The Firmware structure is organized in different folders that keeps the structure modular, next the main folders used for this work are described:

- **cmake**: contains the cmake files (add_flags, add_library, add_module, etc.)
- **msg**: contains uORB msg templates, the uORB msg headers are generated from this folder

- **boards/px4**: contains the different cmake files for each board (they are the file on which you set the used platform, drivers, modules, etc.)
- **ROMFS**: contains the startup scripts, airframes configuration files and mixer definition
- **src/drivers**: contains all the sensors drivers (pwm, gyro, gps, etc.)
- **src/examples**: contains some simple examples which helps to understand the code and system functionalities
- **src/modules**: containts the main code blocks as estimators, commander, controllers, mavlink, vmount etc.
- **src/lib**: contains all the libraries used by modules like mixer, matrix arithmetic, PID default structures etc., it also contains the mixer module
- **Tools/sitl_gazebo**: contains vehicles models, sensors plugins, gazebo worlds and the gazebo starting script

3.2.1 Topic message

A topic message its where the various modules of the system speaks interchanging data, and takes decisions with respect to some flags set on the topics. A list of the most used ones will be show in Section 3.4.1.
PX4 allows to create new messages in a simple way as described in the appendix, Section A.1.

3.2.2 Application

An application is the simplest function that can be implemented on PX4, it can be used to check something during the flight from the command shell, for example it could print the vehicle acceleration when called, or the current flight mode, etc.
For more information and a short tutorial see Section A.3.

3.2.3 Modules

Modules are the fundamental blocks of PX4 architecture, they are parts of the code that runs iteratively following a particular scheduling. Modules (commander, mc controller, estimators, navigator, etc) interchange information through topics, and based on sensor measurements and commands given from QGC or the RC, decide the action to do, such as the control mode, failsafes activation and the control outputs to send to the motors.
An application (Section 3.2.2) can be written to run as either a task (a module with its own stack and process priority) or as a work queue task (a module that runs on a work queue thread, sharing the stack and thread priority with other tasks on the work queue). In most cases a work queue task can be used, as this minimizes resource usage.
The differences between the two are:

- **Work queue task**: the module runs on a shared priority queue, meaning that it does not own a proprietary stack (this is the most used case). In other words more then one task can run with the same priority on the same queue, the scheduler will run the queue with the highest priority first, and on that queue, the first task (by entry order) will be run. The main advantage in using such approach is that it requires less RAM, even though the task is not allowed to sleep or poll on messages (pause execution for a specific period, or check availability of a datum before to do something). Multiple tasks run on the same stack with single priority per work queue. Work queues are essentially used for periodic tasks, such as sensor drivers or the land detector and in particular for control loops (see Section 4.5.1).
- **Task**: the module runs on its own, with its own stack and process priority so it is independent from any queues. The main problem is that it requires a lot of computational resources, so it is the less used module type.

For more information and a short tutorial see Section A.4.

3.2.4 Module Context

When creating a new module, its methods are called in a specific order like showed in Figure 3.2.

The methods used by most of the modules are:

- **task_spawn()**, allocates the new module on the stack, setting its priority.
- **instantiate()**, used for **task** modules, creates a new module instance.
- **init()**, used for **work queue** modules, initializes it and sets the scheduling frequency.
- **Constructor()**, standard constructor method for classes.
- **Run()**, cyclic function.
- **custom_command()**, used to add called functions.
- **print_usage()**, called from the command shell, prints a module description.
- **print_status()**, called from the command shell, prints the module status (e.g. running or not).

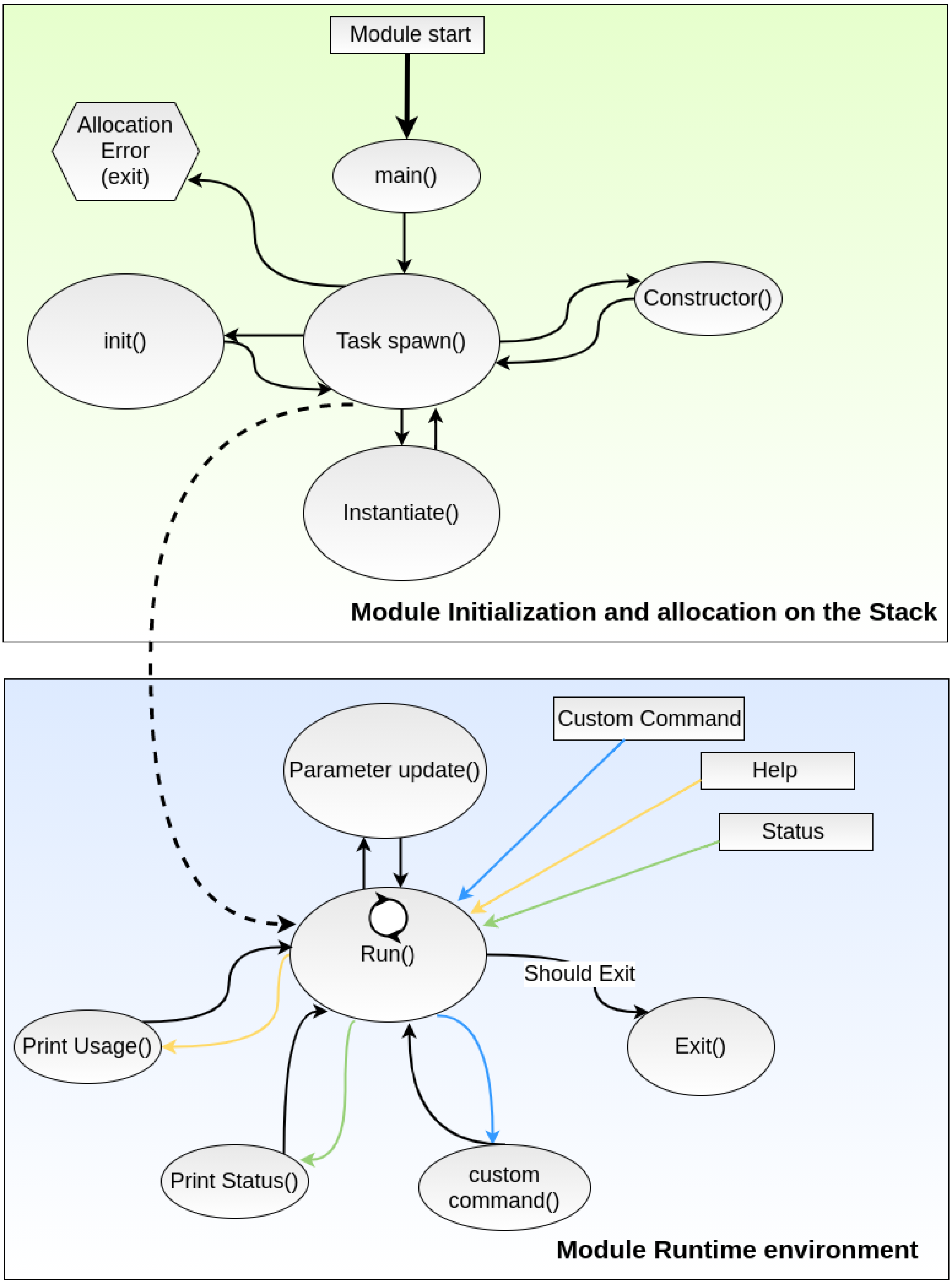


Figure 3.2. Module Context

Note: a detailed method explanation and the actual code can be seen in Appendix 4.

Middleware

PX4 is made of two main layers: the **flight stack** that is responsible of measure estimation and flight control, and the **middleware** that is a general robotics layer providing hardware integration and internal/external communications [25].
Simulators for example needs to speak with PX4 using its middleware communicating system as is shown in Figure 3.3.

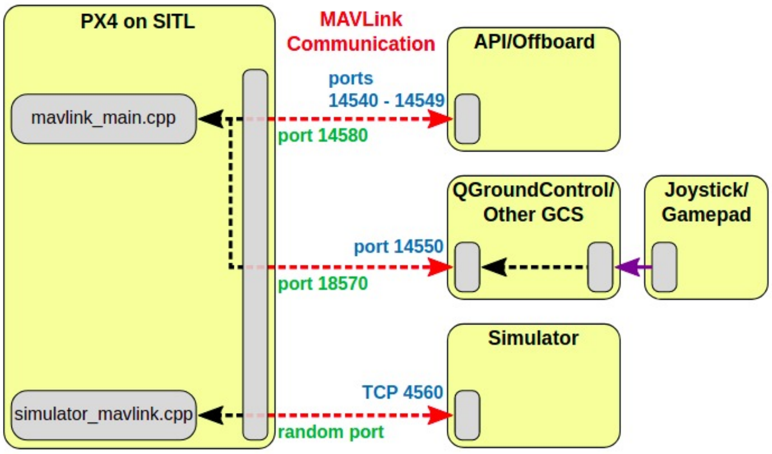


Figure 3.3. SITL communication system

Source: PX4 Developer Guide

The middleware consists primarily of device drivers for embedded sensors, communication with the external world (companion computer, GCS, etc.) and the message bus. The Firmware is made of modules/programs communicating to each other through a publish-subscribe message bus named uORB [27]. While the flight stack should let the drone fly in a reliable manner, the middleware is that part of the firmware necessary to add functionalities, integrate sensors and change the behaviour of already existing ones.
Moreover, the middleware incorporates a simulation layer allowing PX4 to run on a desktop OS and control a modelled vehicle in a simulated environment.

3.3 MAVLINK

MAVLink is the messaging protocol that has been designed for the entire drone ecosystem (the drone-ground control station pair). In particular, PX4 uses Mavlink to communicate with QGroundControl, and as the integration mechanism for connecting to drone components outside of the flight controller: companion computers, Mavlink enabled cameras, etc. [28].
MAVLink protocol defines both the message structure and the serialization criterion at the application layer (it basically defines how information should be passed through the network).
It can be transmitted both through serial telemetry low bandwidth channels, operating

in the MHz range, namely 433 MHz, 868 MHz or 915 MHz, or through WiFi and Ethernet (TCP/IP networks).
Both the TCP and UDP connection protocols can be used, depending on the required reliability of the application [29].

3.3.1 MAVSDK

MAVSDK is a collection of libraries for various programming languages to interface with MAVLink systems such as drones, cameras or ground systems.
The libraries provides a simple API for managing one or more vehicles, which enables programmatic access to vehicle information and telemetry, control over missions, movement and other operations.
The libraries can be used onboard in a drone, on a companion computer, on a ground station or a mobile device.
MAVSDK is cross-platform: Linux, macOS, Windows, Android and iOS [19].

3.4 uORB

The uORB is an asynchronous publish()/subscribe() messaging API used for inter-thread/inter-process communication. Publish and Subscribe to a topic can be done from anywhere in the system [27].
New uORB topics can be added either within the main PX4-Autopilot repository, or can be added in an out-of-tree message definitions.
The list of the built-in topics (more than 100, see Figure 3.4) can be found online on github [30], or within the directory PX4-Autopilot/msg on the development machine where PX4 firmware is downloaded.

Figure 3.4. uORB graph (old version)

Source: PX4 developer guide

Note: a new version (updated periodically) of the uORB graph (Figure 3.4) can be found on the online developer guide.
To use the topic in the code, it is enough to include the header of the message we want to listen to or publish at as shown in Figure 3.5:

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

% e.g. include <uORB/topics/vehicle_attitude.h>

```

Figure 3.5. uORB include

3.4.1 uORB main topics

In this section we are going to see the main topics used by PX4 flight stack, and the ones used for this work, their characteristic, what they are used for, and which module mainly use them (Figure 3.6).

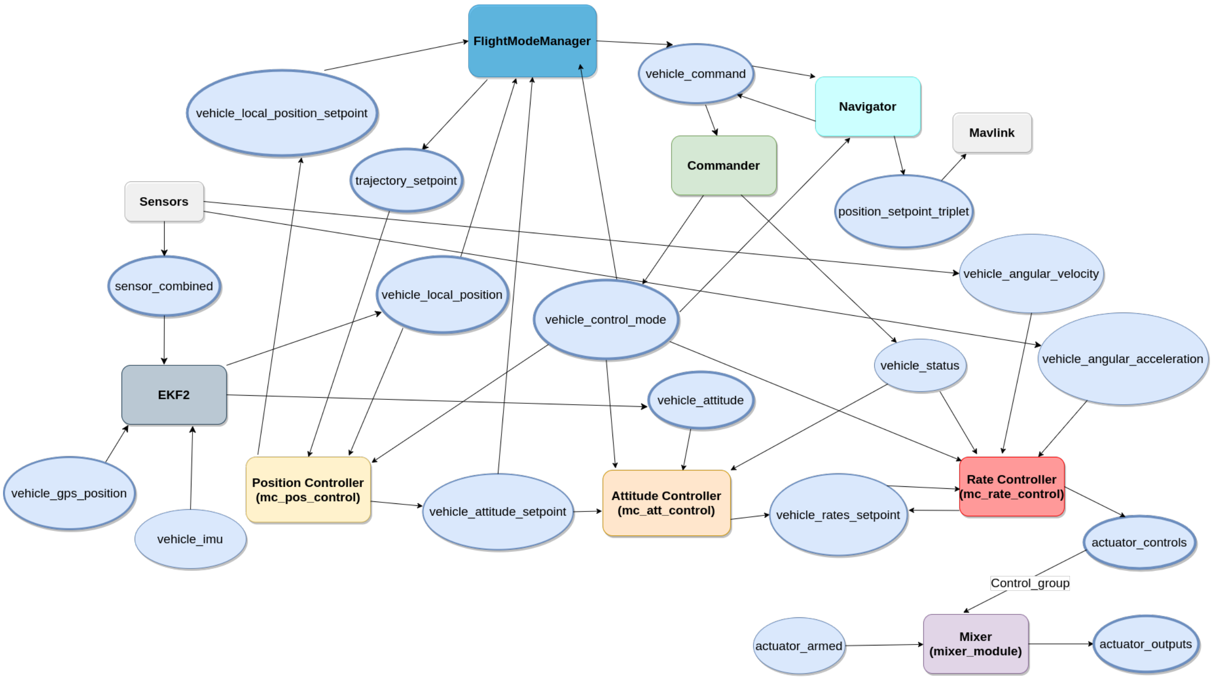


Figure 3.6. Control topics graph
(The circled topics are the ones described in this section)

actuator outputs

The actuator outputs topic contains the PWM values that are passed to the different ports on the autopilot board (e.g. Pixracer). It is generally a value between 1000 and 2000, whereas 900 is the default value for disarmed condition (motors off). The number of PWM values depend on the specific autopilot board. Unfortunately, it is not possible to directly write the PWM value on a specific pin without interfering with the others; the only solution is to pass through actuator controls topic. The modules that mainly publish on this topic are: mixer, uavcan (internal bus communication driver), px4io (IO communication driver). The most important one for our purposes is *mixer*, because it is used to convert values coming from actuator controls into suitable PWM outputs.
The modules that subscribe to it are mavlink and simulator.

actuator controls

The actuator controls topic contains the control values of all the possible control groups, where a control group characterize the type of vehicle we are going to use, in particular the input/output relation (for a description about the control groups see Section 4.8.1, or look at [31]). The different control values (for different control groups) are:

- **actuator controls 0**: roll, pitch, yaw and thrust (multicopter).
- **actuator controls 1**: roll, pitch, yaw and thrust (VTOL).

- **actuator controls 2**: roll, pitch, yaw and shutter (gimbal). This topic can be used only in the presence of the AUX ports.
- **actuator controls 3**: manual passthrough. Useful values are the 6th and the 7th (number 5 and 6 respectively), since they allow to exploit the original mixer files and tune PWM output on pin 5 and 6.
- **actuator controls virtual fw (fixed-wing)**: only used for fixed wing VTOL code, see [32].
- **actuator controls virtual mc (multicopter)**: only used for multicopter VTOL code, see [32].

The module that publishes on this topic is the rate controller (in autonomous flight mode).
The modules that subscribe to it are mixer, commander, mavlink, sensors.

sensor combined

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

- **gyro rad**: average angular rate measured in the XYZ body frame in rad/s over the last gyro sampling period .
- **accelerometer** m/s^2: average value acceleration measured in the XYZ body frame in m/s^2 over the last accelerometer sampling period.

The sensor combined topic receives data from the module sensors and sends it to ekf2, navigator, commander, and mavlink.

vehicle gps position

The vehicle gps position topic contains all the informations coming from gps sensor, such as velocity with respect to ground, accuracy, etc.
Gps publishes on vehicle gps position because it implements driver functionality.
The modules that subscribe to vehicle gps position are: Mavlink, EKF2, commander, navigator.

vehicle attitude

The vehicle attitude topic contains two important parameters:

- **q**: quaternion rotation from XYZ body frame to NED (North East Down) earth frame.
- **delta q reset**: quaternion variation from the last reset.

EKF2 publishes on vehicle attitude, because it provides filtered data based on the measured ones.
The modules that subscribe to vehicle attitude topic are: mavlink and the attitude controller (see Section 4.6).

trajectory setpoint/vehicle local position setpoint

Both topics use the same msg file (see Section A.1) that includes:

- **Position**; x, y, z in NED
- **Orientation**; as yaw angle
- **Speed**; both linear and angular
- **Acceleration**; linear
- **Jerk**; linear (not computed by default)
- **Thrust**; normalized thrust vector in NED

This trajectory setpoint is used by the position controller that subscribe to it and computes the attitude reference and thrust needed value looking at the error between the setpoint and the actual vehicle position. The topic is published by the mavlink interface, if for example MAVROS APIs (see Section 3.5) are used, and from *FlightModeManager* that publishes the trajectory read from the active flight task (see Section 4.4.1).

position setpoint triplet

It contains contains three position setpoint messages (see Section 3.4.1):

- current
- previous
- next

This topic message is mostly used for the global position update from QGC station, and for autonomous missions. The topic is published by the navigator. The module that subscribe to this position setpoint is the FlightTaskAuto and indirectly the flight mode manager that publishes then the trajectory setpoint.

vehicle local position

This topic contains almost the same parameters of the position setpoint (see Section 3.4.1) (position, speed, acelleration, heading), it is updated periodically by the position estimators and keeps track of the vehicle position (in NED frames) as the name suggests. The topic is published by EKF2 and local position estimator. The position controller, commander, navigator and other modules subscribe to it.

vehicle command

The vehicle command topic contains a huge list of commands (each identified with a specific number) to be given to the drone in order to perform specific operations. A complete list of all the available commands can be found at [33]. As it contains such a number of commands, a lot of modules are subscribed to this topic. The most relevant ones are: mavlink, px4io, uavcan, camera trigger (external camera driver), commander, navigator.

vehicle control mode

The vehicle control mode topic contains a list of boolean values used to define the kind of control mode that is active (complete list available at *Flight Modes* [23]). Commander module publishes on this topic deciding the flight mode to use.
The *controllers* modules subscribe to this topic in order to check if they have to compute the control action or not.

3.5 ROS/MAVROS

PX4 allows to use a companion computer onboard or offboard where heavy computional task can be executed in parallel (for example obstacle avoidance).
The companion computer could be a Raspberry Pi where ROS system is running. ROS system is able to communicate with the drone using the *MAVROS* protocol [34]. Additional information can be found in a detailed master thesis [35], or on the online developer guide.

3.6 Simulation environment and functionalities

This section introduces to the simulation environments used by PX4 highlighting some of their features.

3.6.1 Simulators

Gazebo

One of the simulation environments used by PX4 is ***Gazebo*** (Figure 3.7) [36], it is a well known 3D simulation environment mostly used by ROS applications.
The main features are:

- **Dynamic Simulation**, physics engines including ODE, Bullet, Simbody and others
- **Advanced 3D Graphics**, utilizing OGRE (an open source graphic engine), provides realistic rendering
- **Sensors and Noise**, generate various sensors data, optionally with noise

- **Robot Models**, many robots are provided or you can build your own using SDF files
- Other features and characteristics can be found on Gazebo website [36]

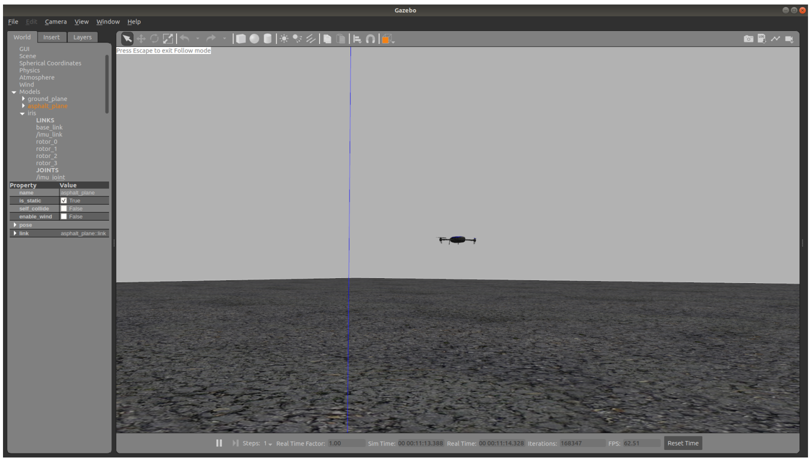


Figure 3.7. Gazebo

Jmavsim and others

JMAVSim (Figure 3.8) [37] is a simple multirotor/Quad simulator that allows you to fly copter type vehicles running PX4 around a simulated world.

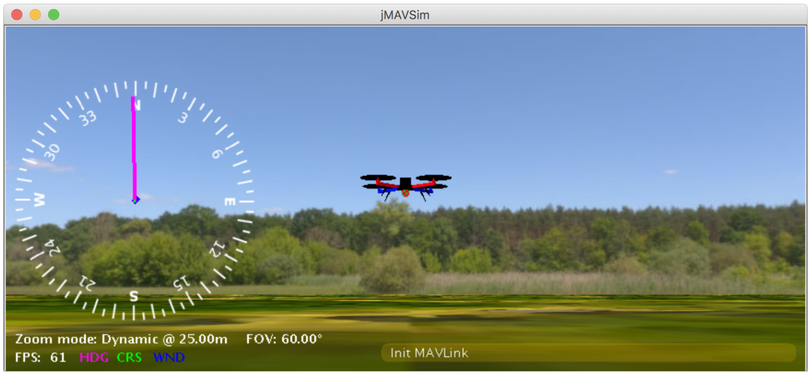


Figure 3.8. jmavsim

Other simulators are:

- **FlightGear Simulation**
- **JSBSim Simulation**

3.6.2 Simulation modes

SITL

Simulation-In-The-Loop allows to test the firmware, the flight stack, the control design and all its features without the need of having the physical hardware (see Section 2.2).

In SITL (Figure 3.9) the simulator recreate the complete drone (hardware, sensors, motors) plus the environment in which it is flying; the drone can also be connected to *QGC* where you can give commands, log files, etc. [38].

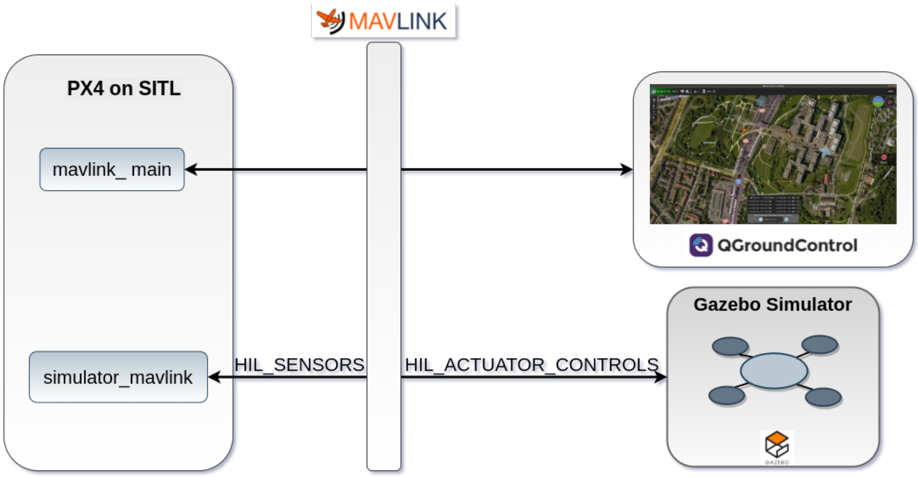


Figure 3.9. SITL scheme

This simulation mode is fundamental for safety reasons, it allows to test changes to PX4 code, avoiding the risk to break something on the real drone, once everything seems to work fine in SITL, then different tests in HITL (see 3.6.2) or on real world can be attempted.

SITL Lockstep

This is a fundamental feature implemented in PX4. Lockstep [39] is a simulation technique that handles the synchronization between two process running with different clocks, in this case the **flight stack** one side and the **simulator** on the other. Without it all the sensor and actuator values will be not synchronized and this will bring to a system completely unstable and not reliable.
The PX4 sequence of steps for lockstep are:

- The simulation sends a sensor message **HIL_SENSOR** including a timestamp to update the sensor state and time of PX4
- PX4 receives this and does one iteration of state estimation, controls, etc. and eventually sends an actuator message **HIL_ACTUATOR_CONTROLS**
- The simulation waits until it receives the actuator/motor message, then simulates the physics and calculates the next sensor message to send to PX4 again.

Note: it is allowed to disable lockstep, both from PX4 side or from Gazebo side, this will permit the system to do not wait each others and run freely at a customized frequency [40].

SITL Simulation Speed

Thanks to **lockstep**, SITL simulation also allows to change the simulation speed to go faster or slower then real time, and adjust the desired speed factor with respect to development machine capabilities.
This feature allows to simulate very long trajectories in a faster time then real one, so that more tests can be done in a short time slot.

HITL

Hardware-in-the-Loop (Figure 3.10) is a simulation mode in which normal PX4 firmware is run on real flight controller hardware, it permit to test the flight code on the real hardware. The simulator (Gazebo, jmavsim or others) recreate the environment and all sensors and motors dynamic, data between Gazebo and QGroundControl are shared through **mavlink** communication, and between PX4 and Gazebo through **usb**.

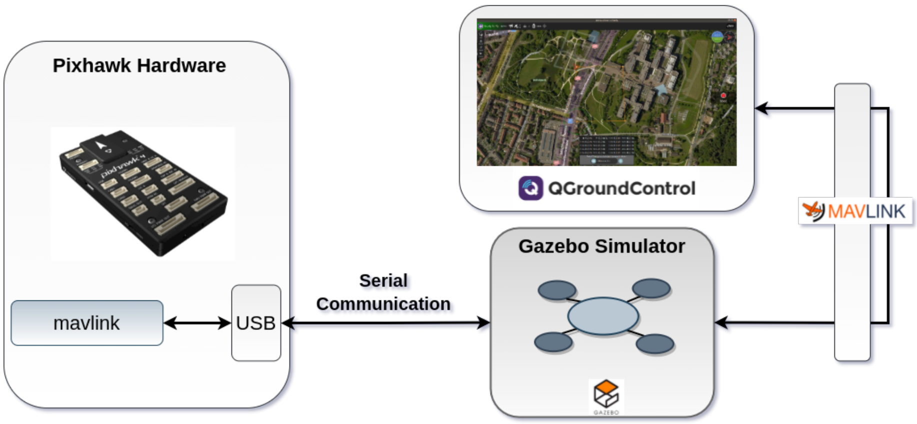


Figure 3.10. HITL scheme

It is useful to assure that the computational power in the hardware is enough and to have a more realistic test [41].

SIH

Simulation-In-Hardware (SIH) is an alternative to HITL for a quadrotor. In this setup, everything is running on embedded hardware - the controller, the state estimator, and the simulator. The Desktop computer is only used to display the virtual vehicle [42].

Chapter 4

PX4 Control Architecture

The PX4 control structure is based on multiple controller blocks linked in cascade and feedback, the control loops are mainly based on simple P and PID controllers. The modules activation time is determined by an internal scheduler. In order to well behave, the internal loops will have a loop frequency much higher than the external ones.

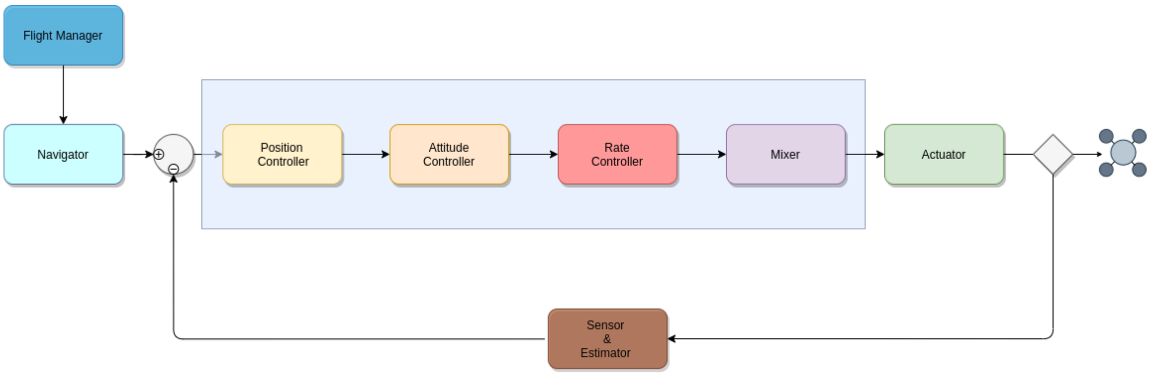


Figure 4.1. Complete generic control scheme

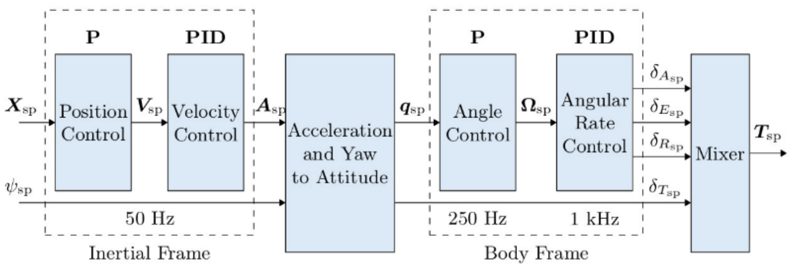


Figure 4.2. Multicopter Controller

Source: PX4 Development Guide

The blocks showed in Figure 4.1 and Figure 4.2 are:

- The **estimator** takes one or more sensor inputs, combines them according to specific algorithms, and computes vehicle states (for instance the attitude measurement is based on IMU sensors data).

- The **navigator (or the RC controller)** sends the setpoints to the controller. The navigator works in autonomous flight mode, the RC is enabled in the manual/stable mode (see Section 2.5).

- The **controller** takes the reference setpoints and the measurements as input, its aim is to adjust the value of the process variable so that it matches the target setpoints.

- The **mixer** takes force commands (e.g. for multicopter: thrust force, pitch, roll and yaw torques) and translates them into individual motor commands, ensuring that some limits are not exceeded (as the maximum motor rotation speed). This translation is specific for each vehicle type and geometry, it depends on various factors, such as the motor arrangements with respect to the center of gravity, the drag coefficient and some other terms.

- The **actuator** sends the computed actuator commands to the motors as PWM references via Mavlink messages (see Section 3.3).

A more detailed control scheme can be seen in Figure 4.3.

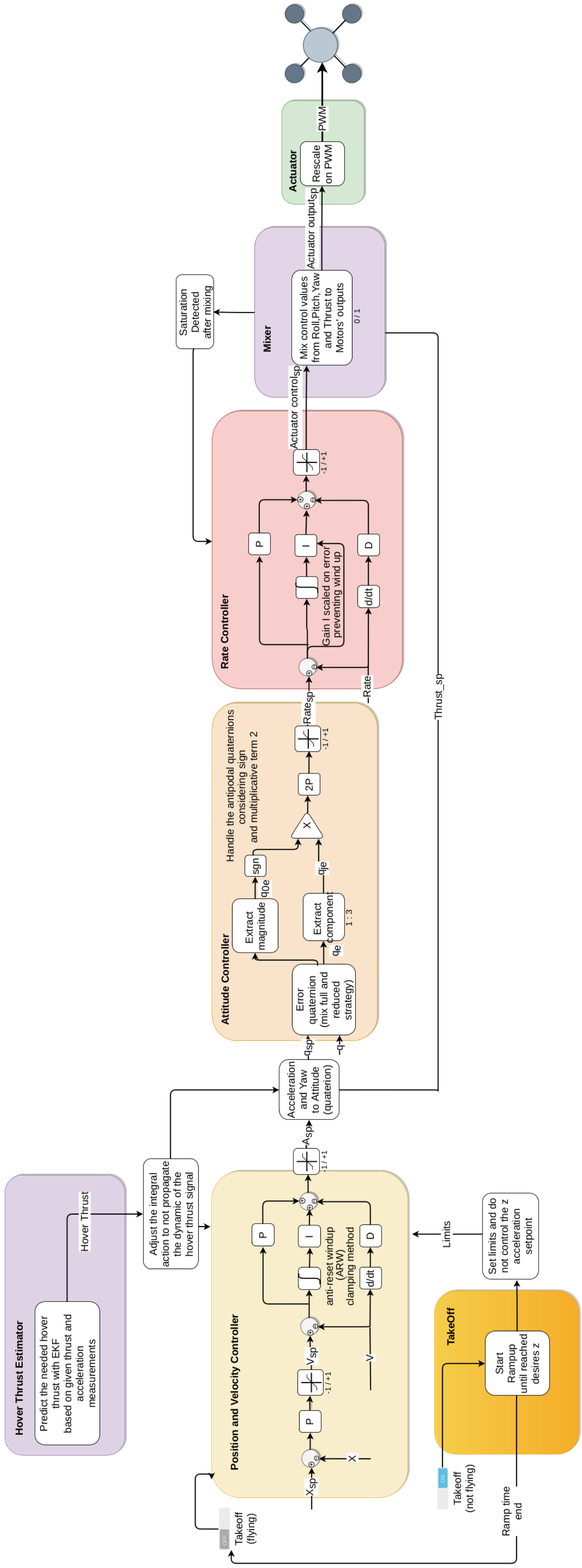


Figure 4.3. Detailed Multicopter Controller Scheme

The showed blocks in Figure 4.3 are:

- **Position Controller** (see Section 4.5) takes the position setpoint and compute the acceleration setpoint looking the hover thrust estimation (from estimator module) and the Takeoff state. It also contains the block "**Acceleration and yaw to attitude**" where the thrust is computed and the quaternion setpoint is calculated

- **Attitude Controller** (see Section 4.6) takes the quaternion setpoint and computes the angular rate setpoint

- **Rate Controller** (see Section 4.6) takes the rate setpoint and computes the actuator controls (roll, pitch and yaw torques)

- **Mixer** (see Section 4.8) take the thrust setpoint (sent by the position controller) and the torques computed by the rate controller and mix them to the actuator outputs

- **Actuator** takes the actuator outputs computed by the mixer and send the commands to the rotors (see Section 4.9)

Note: the main topics used by the described modules and some of the dependencies between them was shown in Figure 3.6 in Chapter 3.2.4:

4.1 Sensors and Estimator

Sensor and Estimator modules take the Mavlink messages from sensors (IMU, magnetometer, height, GPS, air speed etc.) and after filtering and estimating data, publish them on topics. PX4 has more then one estimator available, but the most used is EKF2 [43].
The EKF (Extended Kalman Filter) runs on a delayed "fusion time horizon" to allow for different time delays on each measurement relative to the IMU. Data for each sensor is FIFO buffered and retrieved from the buffer by the EKF at the correct time. Some of the available measurements (if the sensors are mounted on the vehicle) and estimates provided by EKF2 processing the sensor data [44] are:

- **Quaternion**, defining the rotation from NED local earth frame to X, Y, Z body frame

- **IMU speed**- NED (m/s)

- **IMU position**- NED (m)

- **IMU delta angle bias estimates** - X, Y, Z (rad)

- **Earth Magnetic field components** - NED (gauss)

- **Wind velocity**- North, East (m/s)

sensor module

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

- Read the output from the sensor drivers (sensor gyro, sensor accel, sensor baro, sensor mag, differential pressure). If there are multiple of the same type, do voting and failover handling (verifying data validity and choosing the best data available). Then apply the board rotation and temperature calibration (if enabled). And finally publish the data; one of the published topics is "sensor combined" (see topic in Section 3.4.1), used by many parts of the system.

- Make sure the sensor drivers get the updated calibration parameters (scale and offset) when the parameters change or on startup. The sensor drivers use the ioctl interface (which is the output driver communicating with the IO co-processor) for parameter updates. For this to work properly, the sensor drivers must already be running when 'sensors' is started [45].

- Do pre-flight sensor consistency checks and publish the sensor pre-flight topic.

4.2 Commander

The commander is the central block for PX4 architecture, basically the PX4 brain, it takes inputs from sensors, estimators, *QGC*, RC controller; then based on the vehicle condition and the command received by the user it sets the "**vehicle control mode**" (Figure 2.5), it also handles flight failure (failsafe strategies).
An example is the case when the user give the command "takeoff" to the drone, the commander first checks if everything is working fine, then publishes the control mode "takeoff", once the take off is completed the commander publishes the new control mode "hold", it keeps the drone in hovering until a new command arrives.
The commander module subscribes to a huge number of topics as it can be seen in (Figure 4.4).

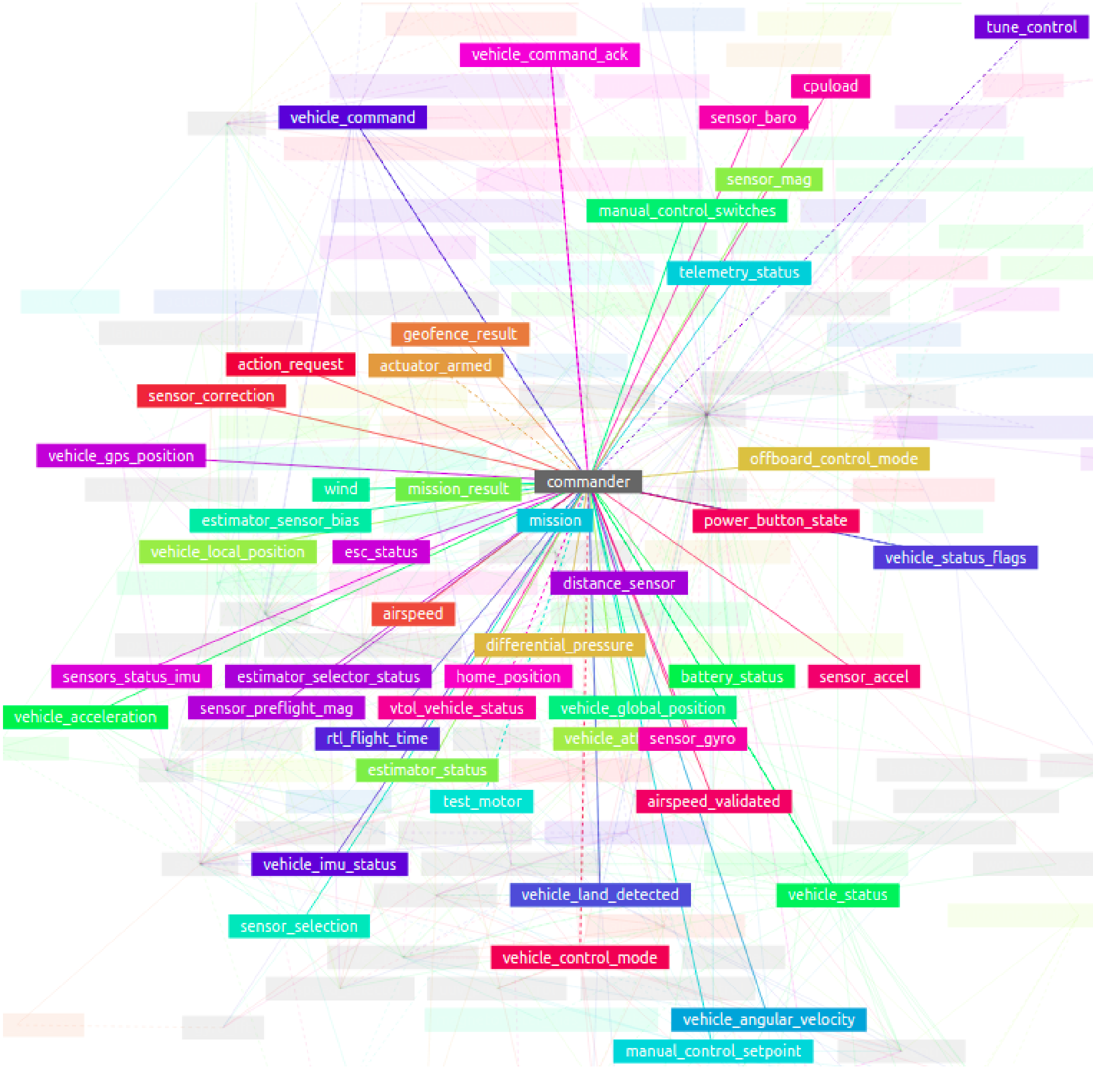


Figure 4.4. Commander Topics

Source: PX4 Development Guide

4.3 Navigator and RC

Navigator is the module responsible for autonomous flight modes. Examples of such a flight modes are mission, takeoff and *RTL*. Moreover, navigator is in charge for geo-fence violation monitoring (geofencing is a safety measure put in place to prohibit drones access to a restricted area; these no-fly zones are protected airspace areas, for example can be used to prevent a vehicle flying out of range of the RC controller).
The different internal modes are implemented as separate classes that inherit from a common base class NavigatorMode. The member "_navigation_mode" contains the current active mode [23].
The navigator publish the position_setpoint_triplets topic, that is first included by FlightTaskAuto that sets the current position setpoint as a *waypoint*, then the waypoint is taken from FligthModeManager and published on trajectory_setpoint topic [46].

The **navigator** uses the topics in Figure 4.5.

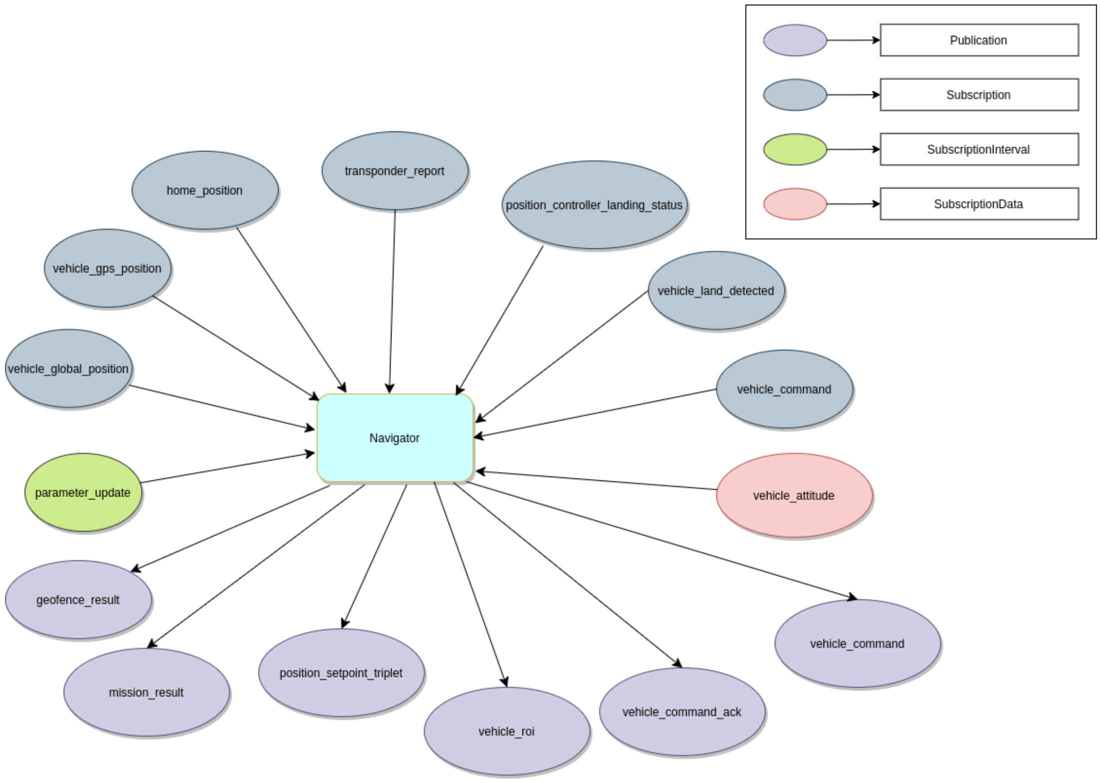


Figure 4.5. Navigator Topics

4.4 Flight Mode Manager

The Flight Mode Manager is an important module for PX4 firmware, it handle the different task requests as "Takeoff, Land, Orbit, Manual" (see Figure 2.5) generating the related trajectory setpoints.
The advantage of keeping this block separated from the controller and the navigator, is to add new customized tasks in an simple way.
The developer has to create a new task with the same form of others and add the command to call it in the commander module.

4.4.1 Flight Task

It is a class of the Flight Mode Manager module (see Section 4.4), it includes all the specific movement behaviours like follow me, orbit, flight smoothing.
Its function is to generate setpoints for the controller from arbitrary input data. Each task implements a different vehicle behavior following a specific trajectory.
Programmers typically override the activate() and update() virtual methods by calling the base task's minimal implementation and extending it with the implementation of the desired behavior. The activate() method is called when switching to the task and allows to initialize its state and take over gently from the passed over setpoints the previous task was just applying.

The update() method is called on every loop iteration during the execution and contains the core behavior implementation producing setpoints.
The online guide contains a wider overview of it and a guide on how to add a customized task [47].

4.4.2 Flight Mode Manager Code

The **multicopter** position controller uses the topics in Figure 4.6.

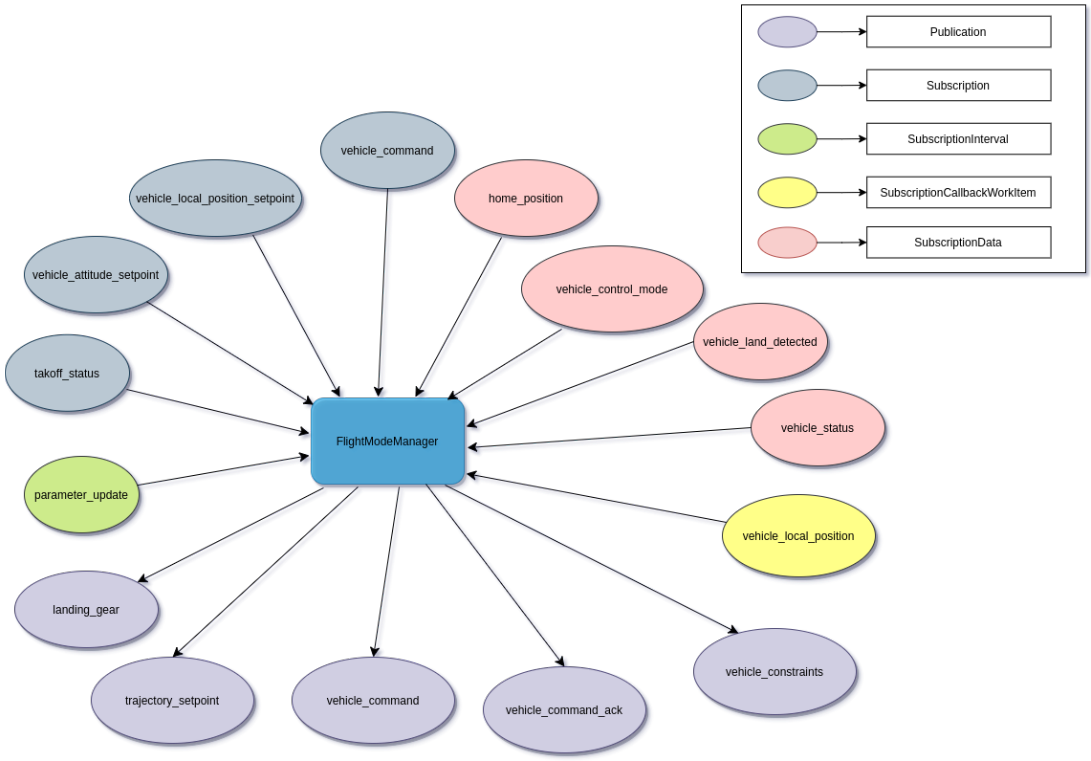


Figure 4.6. Flight Mode Manager Topics

Below the main FlightModeManager methods, and a description of them, are shown into the code.

Run() method (Listing 1).
Firstly updates the parameters on which the FlighModeManager subscribes.
Then updates the local position, control mode and land topics.
Activates the necessary controllers for the active mode and task.
Starts the Flight Task and then looking at the vehicle status ("nav_state") switches to the corresponding task if not yet activated.
Finally if any task is active generates the corresponding setpoint.

```
void FlightModeManager::Run()
{
        if (should_exit()) {
                _vehicle_local_position_sub.unregisterCallback();
                exit_and_cleanup();
                return;
        }

        perf_begin(_loop_perf);

        // Check if parameters have changed
        if (_parameter_update_sub.updated()) {
                // clear update
                parameter_update_s param_update;
                _parameter_update_sub.copy(&param_update);
                updateParams();
        }

        // generate setpoints on local position changes
        vehicle_local_position_s vehicle_local_position;

        if (_vehicle_local_position_sub.update(&vehicle_local_position)) {
                const hrt_abstime time_stamp_now = hrt_absolute_time();
                // Guard against too small (< 0.2ms) and too large (> 100ms) dt's.
                const float dt = math::constrain(((time_stamp_now - _time_stamp_last_loop) / 1e6f),
                ↪  0.0002f, 0.1f);
                _time_stamp_last_loop = time_stamp_now;

                _home_position_sub.update();
                _vehicle_control_mode_sub.update();
                _vehicle_land_detected_sub.update();

                if (_vehicle_status_sub.update()) {
                        if (_vehicle_status_sub.get().is_vtol && (_wv_controller == nullptr)) {
                                // if vehicle is a VTOL we want to enable weathervane capabilities
                                _wv_controller = new WeatherVane();
                        }
                }

                // activate the weathervane controller if required. If activated a flighttask can
                ↪  use it to implement a yaw-rate control strategy
                // that turns the nose of the vehicle into the wind
                if (_wv_controller != nullptr) {

                        // in manual mode we just want to use weathervane if position is controlled
                        ↪  as well
                        // in mission, enabling wv is done in flight task
                        if (_vehicle_control_mode_sub.get().flag_control_manual_enabled) {
                                if (_vehicle_control_mode_sub.get().flag_control_position_enabled &&
                                ↪  _wv_controller->weathervane_enabled()) {
                                        _wv_controller->activate();

                                } else {
                                        _wv_controller->deactivate();
                                }
                        }

                        vehicle_attitude_setpoint_s vehicle_attitude_setpoint;
                        _vehicle_attitude_setpoint_sub.copy(&vehicle_attitude_setpoint);
                        _wv_controller->update(matrix::Quatf(vehicle_attitude_setpoint.q_d).dcm_z(),
                        ↪  vehicle_local_position.heading);
                }

                start_flight_task();

                if (_vehicle_command_sub.updated()) {
                        handleCommand();
                }

                if (isAnyTaskActive()) {
                        generateTrajectorySetpoint(dt, vehicle_local_position);
                }

        }

        perf_end(_loop_perf);
}
```

Listing 1. Flight Mode Manager Run()

FlightModeManager::generateTrajectorySetpoint(...) (Listing 2). *Firstly the required controllers are enabled (setYawHandler), the setpoint of the current task is updated calling the method "_current_task.task->update()", then the method* *"getPositionSetpoint()" takes the new setpoint values and store them in "setpoint", then "setpoint" is published on "trajectory_ setpoint topic".*

```
void FlightModeManager::generateTrajectorySetpoint(const float dt,
                const vehicle_local_position_s &vehicle_local_position)
{
        _current_task.task->setYawHandler(_wv_controller);

        // If the task fails sned out empty NAN setpoints and the controller will emergency failsafe
        vehicle_local_position_setpoint_s setpoint = FlightTask::empty_setpoint;
        vehicle_constraints_s constraints = FlightTask::empty_constraints;

        if (_current_task.task->updateInitialize() && _current_task.task->update() &&
        ↪  _current_task.task->updateFinalize()) {
                // setpoints and constraints for the position controller from flighttask
                setpoint = _current_task.task->getPositionSetpoint();
                constraints = _current_task.task->getConstraints();
        }

        // limit altitude according to land detector
        limitAltitude(setpoint, vehicle_local_position);

        if (_takeoff_status_sub.updated()) {
                takeoff_status_s takeoff_status;

                if (_takeoff_status_sub.copy(&takeoff_status)) {
                        _takeoff_state = takeoff_status.takeoff_state;
                }
        }

        if (_takeoff_state < takeoff_status_s::TAKEOFF_STATE_RAMPUP) {
                // reactivate the task which will reset the setpoint to current state
                _current_task.task->reActivate();
        }


        setpoint.timestamp = hrt_absolute_time();
        _trajectory_setpoint_pub.publish(setpoint);

        constraints.timestamp = hrt_absolute_time();
        _vehicle_constraints_pub.publish(constraints);

        // if there's any change in landing gear setpoint publish it
        landing_gear_s landing_gear = _current_task.task->getGear();

        if (landing_gear.landing_gear != _old_landing_gear_position
            && landing_gear.landing_gear != landing_gear_s::GEAR_KEEP) {

                landing_gear.timestamp = hrt_absolute_time();
                _landing_gear_pub.publish(landing_gear);
        }

        _old_landing_gear_position = landing_gear.landing_gear;
}
```

Listing 2. FlightModeManager::generateTrajectorySetpoint()

FlightTask::getPositionSetpoint() (Listing 3).
Takes the setpoint values given by the active task and copy it to "vehicle_ local_ position_ setpoint" topic.

```
const vehicle_local_position_setpoint_s FlightTask::getPositionSetpoint()
{
	vehicle_local_position_setpoint_s vehicle_local_position_setpoint{};

	//myelic_status_s elic_status;

	/* fill position setpoint message */

	vehicle_local_position_setpoint.timestamp = hrt_absolute_time();

	vehicle_local_position_setpoint.x = _position_setpoint(0);
	vehicle_local_position_setpoint.y = _position_setpoint(1);
	vehicle_local_position_setpoint.z = _position_setpoint(2);
	//PX4_INFO("ZRef=%f" ,_position_setpoint(2));
	vehicle_local_position_setpoint.vx = _velocity_setpoint(0);
	vehicle_local_position_setpoint.vy = _velocity_setpoint(1);
	vehicle_local_position_setpoint.vz = _velocity_setpoint(2);

	vehicle_local_position_setpoint.yaw = _yaw_setpoint;
	vehicle_local_position_setpoint.yawspeed = _yawspeed_setpoint;

	_acceleration_setpoint.copyTo(vehicle_local_position_setpoint.acceleration);
	_jerk_setpoint.copyTo(vehicle_local_position_setpoint.jerk);

	// deprecated, only kept for output logging
	matrix::Vector3f(NAN, NAN, NAN).copyTo(vehicle_local_position_setpoint.thrust);
	return vehicle_local_position_setpoint;

}
```

Listing 3. FlightTask::getPositionSetpoint()

4.5 Position controller

The multicopter position controller diagram is shown in Figure 4.7.

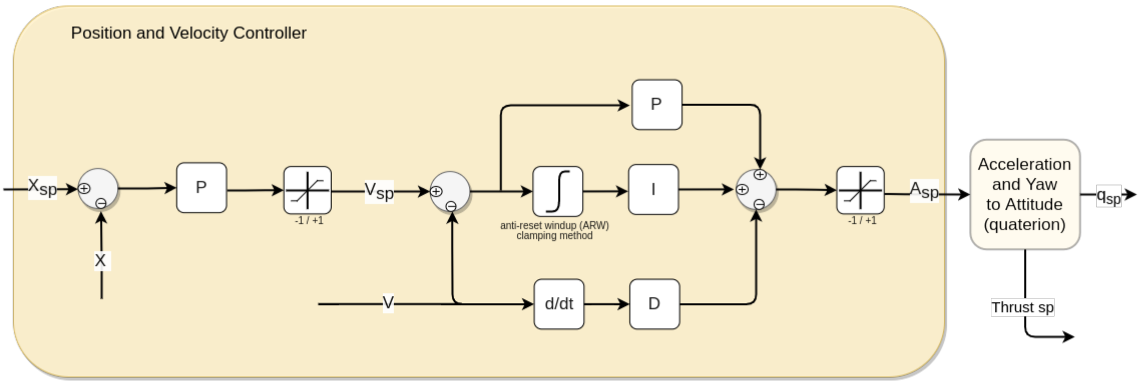


Figure 4.7. Position Controller

The position control output is a thrust vector split in two components: thrust direction (namely rotation matrix for multicopter orientation) and thrust scalar (i.e. multicopter thrust itself). The controller does not need Euler angles for its routine, they are generated to be used and understood by developers (logging).
Some remark about such a control schematic based on the developer guide [48]:

- The position loop is used for holding position, keep the drone on the desired altitude (along z direction) and when the desired velocity along an axis is null. However the outer position loop may be bypassed depending on the used mode (whenever the position should not be held).

- The integrator within the inner loop controller (velocity) includes an anti-windup for saturation, using clamping method [49].

The controller is made of three main blocks: a P (Proportional) loop for position error, a PID (Proportional Integral Derivative) loop for velocity error and then a block that transforms the accelerations in thrust module and direction necessary to compute the quaternion, used by the attitude controller next (see Section 4.6).

4.5.1 Position Controller Code

The **multicopter** *position* controller uses the topics in Figure 4.8.

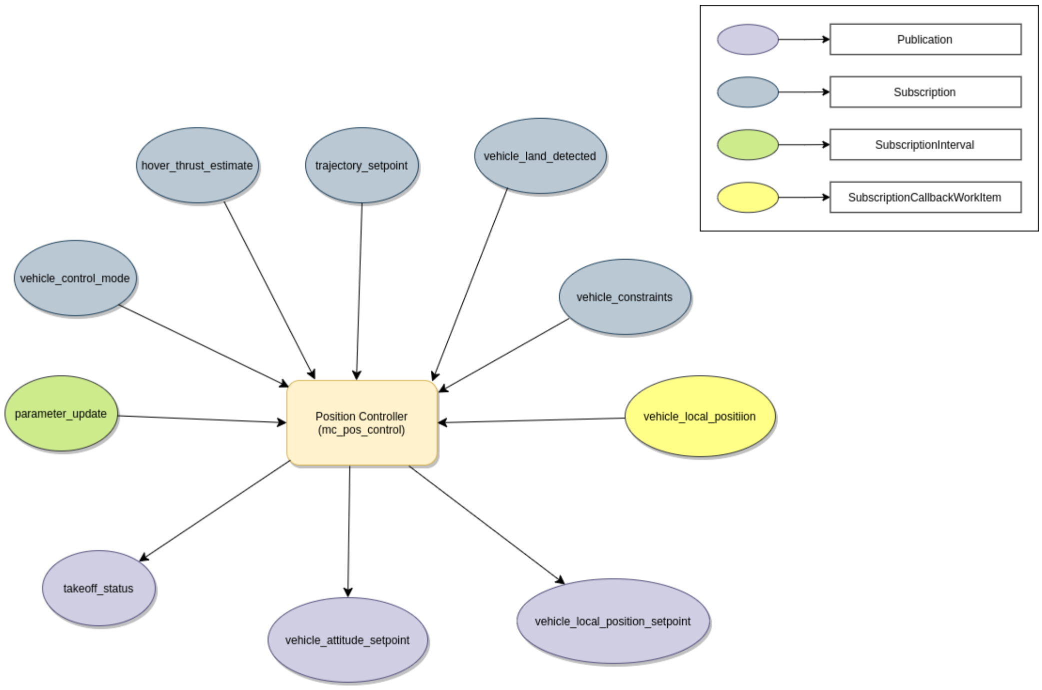


Figure 4.8. Position Controller Topics

Below the main methods of the position controller and the blocks described above are shown into the code.
Multicopter Position Control is a scheduled work item (nav_and_controllers priority stack, the highest after rate controller and sensors).
The method Run() is called iteratevely at fixed sampling time, it runs the update method on the controller object, that will run the Position, Velocity, and Acceleration methods.
The take off transition is handled as well.

Run() method part 1 (Listing 4).
The scheduling time is set.

```
void MulticopterPositionControl::Run()
{
	if (should_exit()) {
		_local_pos_sub.unregisterCallback();
		exit_and_cleanup();
		return;
	}

	// reschedule backup
	ScheduleDelayed(100_ms);
	parameters_update(false);

	perf_begin(_cycle_perf);

}
```

Listing 4. MulticopterPositionControl 1

Run() method part 2 (Listing 5).
Updates the local position topic and computes the delta time passed between calls.

```
	 vehicle_local_position_s local_pos;
	if (_local_pos_sub.update(&local_pos)) {
		const hrt_abstime time_stamp_now = local_pos.timestamp;
		const float dt = math::constrain(((time_stamp_now - _time_stamp_last_loop) * 1e-6f),
		↪  0.002f, 0.04f);
		_time_stamp_last_loop = time_stamp_now;

		// set _dt in controllib Block for BlockDerivative
		setDt(dt);

		const bool was_in_failsafe = _in_failsafe;
		_in_failsafe = false;
```

Listing 5. MulticopterPositionControl 2

Run() method part 3 (Listing 6).
Updates the control mode published by commander and the land status.
Then updates the estimated thrust needed to hoover.
*The method "updateHoverThrust" adjusts the integral value of the velocity for z direction, as showed in the equation below (**G** is the gravitational acceleration):*

$$vel(z) = vel(z) + \frac{(HoverThrust_{new} - HoverThrust_{old}) * G}{HoverThrust_{new}}; \quad (4.1)$$

Note: *the estimation of the hover value is done separately by the module mc_hover_thrust_estimation. It estimates the hovering thrust using both the acceleration and the thrust given by the position controller during the take off phase.*

```
                _vehicle_control_mode_sub.update(&_vehicle_control_mode);
                _vehicle_land_detected_sub.update(&_vehicle_land_detected);

                if (_param_mpc_use_hte.get()) {
                        hover_thrust_estimate_s hte;

                        if (_hover_thrust_estimate_sub.update(&hte)) {
                                if (hte.valid) {
                                        _control.updateHoverThrust(hte.hover_thrust);
                                }
                        }
                }
}
}
```

Listing 6. MulticopterPositionControl 3

Run() method part 4 (Listing 7).
Sets internally the states read from local_pos.
Verifies that the position control is enabled in the control mode topic.
Adjusts existing (or older) setpoint with any EKF reset deltas if the reset counter is different. This happens when EKF estimator for some reasons changes the first measure source, resetting the data, when this occurs, EKF increases the reset counter and saves a delta to be used for compensate the setpoints, in order to avoid glitches.

```
        PositionControlStates states{set_vehicle_states(local_pos)};

        if (_vehicle_control_mode.flag_multicopter_position_control_enabled) {

                const bool is_trajectory_setpoint_updated =
                ↪  _trajectory_setpoint_sub.update(&_setpoint);

                // adjust existing (or older) setpoint with any EKF reset deltas
                if (_setpoint.timestamp < local_pos.timestamp) {
                        if (local_pos.vxy_reset_counter != _vxy_reset_counter) {
                                _setpoint.vx += local_pos.delta_vxy[0];
                                _setpoint.vy += local_pos.delta_vxy[1];
                        }

                        if (local_pos.vz_reset_counter != _vz_reset_counter) {
                                _setpoint.vz += local_pos.delta_vz;
                        }

                        if (local_pos.xy_reset_counter != _xy_reset_counter) {
                                _setpoint.x += local_pos.delta_xy[0];
                                _setpoint.y += local_pos.delta_xy[1];
                        }

                        if (local_pos.z_reset_counter != _z_reset_counter) {
                                _setpoint.z += local_pos.delta_z;
                        }

                        if (local_pos.heading_reset_counter != _heading_reset_counter) {
                                _setpoint.yaw += local_pos.delta_heading;
                        }
            }
```

Listing 7. MulticopterPositionControl 4

Run() method part 5 (Listing 8).
If the offboard control mode is enabled (control the PX4 flight stack using software running outside of the autopilot), set the corresponding flags and constraints for take off

```
        if (_vehicle_control_mode.flag_control_offboard_enabled) {

                bool want_takeoff = _vehicle_control_mode.flag_armed &&
                ↪  _vehicle_land_detected.landed
                                    && hrt_elapsed_time(&_setpoint.timestamp) < 1_s;

                if (want_takeoff && PX4_ISFINITE(_setpoint.z)
                    && (_setpoint.z < states.position(2))) {

                        _vehicle_constraints.want_takeoff = true;

                } else if (want_takeoff && PX4_ISFINITE(_setpoint.vz)
                           && (_setpoint.vz < 0.f)) {

                        _vehicle_constraints.want_takeoff = true;

                } else if (want_takeoff && PX4_ISFINITE(_setpoint.acceleration[2])
                           && (_setpoint.acceleration[2] < 0.f)) {

                        _vehicle_constraints.want_takeoff = true;

                } else {
                        _vehicle_constraints.want_takeoff = false;
                }

            // override with defaults
            _vehicle_constraints.speed_xy = _param_mpc_xy_vel_max.get();
                _vehicle_constraints.speed_up = _param_mpc_z_vel_max_up.get();
                _vehicle_constraints.speed_down =_param_mpc_z_vel_max_dn.get();
        }
```

Listing 8. MulticopterPositionControl 5

Run() method part 6 (Listing 9).
Handles smooth take off, this transition is internally managed by another class "Take-off", this phase consists in a Ramp (rampup phase) for the thrust action allowing limited slope. The ramp grows until the desired z position is reached in a predefined fixed time, when the process end the "flight" mode is enabled.
Checks if "flight mode" is not active yet, and if true sets the z acceleration setpoint to NAN value, which means (for PX4 controller), "do not track this variable".
Checks if not taken off, or flying but for some reason the ground is touched, and if true sets an high downward acceleration to make sure that thrust goes to zero, then in order to prevent windup problems, reset the integral state.

```
        // handle smooth takeoff
        _takeoff.updateTakeoffState(_vehicle_control_mode.flag_armed, _vehicle_land_detected.landed,
                                    _vehicle_constraints.want_takeoff,
                                    _vehicle_constraints.speed_up, false, time_stamp_now);

        const bool flying = (_takeoff.getTakeoffState() >= TakeoffState::flight);

        if (is_trajectory_setpoint_updated) {

                // make sure takeoff ramp is not amended by acceleration feed-forward
                if (!flying) {
                        _setpoint.acceleration[2] = NAN;
                }

                const bool not_taken_off = (_takeoff.getTakeoffState() < TakeoffState::rampup);
                const bool flying_but_ground_contact = (flying &&
                ↪  _vehicle_land_detected.ground_contact);

                if (not_taken_off || flying_but_ground_contact) {
                        // we are not flying yet and need to avoid any corrections
                        reset_setpoint_to_nan(_setpoint);
                        Vector3f(0.f, 0.f, 100.f).copyTo(_setpoint.acceleration); // High downwards
                        ↪  acceleration to make sure there's no thrust
                        // prevent any integrator windup
                        _control.resetIntegral();
                }
        }
```

Listing 9. MulticopterPositionControl 6

Run() method part 7 (Listing 10).
If take off state is active limits the control action with constraints on tilt angle, speed and thrust.
Then sets the thrust limits (used as scale factor for x and y direction), and velocity limits.

```
        // limit tilt during takeoff ramupup
        const float tilt_limit_deg = (_takeoff.getTakeoffState() < TakeoffState::flight)
                                     ? _param_mpc_tiltmax_lnd.get() : _param_mpc_tiltmax_air.get();
        _control.setTiltLimit(_tilt_limit_slew_rate.update(math::radians(tilt_limit_deg), dt));

        const float speed_up = _takeoff.updateRamp(dt,
                               PX4_ISFINITE(_vehicle_constraints.speed_up) ?
                               ↪  _vehicle_constraints.speed_up : _param_mpc_z_vel_max_up.get());
        const float speed_down = PX4_ISFINITE(_vehicle_constraints.speed_down) ?
        ↪  _vehicle_constraints.speed_down :
                                 _param_mpc_z_vel_max_dn.get();
        const float speed_horizontal = PX4_ISFINITE(_vehicle_constraints.speed_xy) ?
        ↪  _vehicle_constraints.speed_xy :
                                       _param_mpc_xy_vel_max.get();

        // Allow ramping from zero thrust on takeoff
        const float minimum_thrust = flying ? _param_mpc_thr_min.get() : 0.f;

        _control.setThrustLimits(minimum_thrust, _param_mpc_thr_max.get());

        _control.setVelocityLimits(
                math::constrain(speed_horizontal, 0.f, _param_mpc_xy_vel_max.get()),
                math::min(speed_up, _param_mpc_z_vel_max_up.get()), // takeoff ramp starts with
                ↪  negative velocity limit
                math::constrain(speed_down, 0.f, _param_mpc_z_vel_max_dn.get()));
```

Listing 10. MulticopterPositionControl 7

Run() method part 8 (Listing 11).
Sets the setpoint values into the object "_control".

```
        _control.setInputSetpoint(_setpoint);

        // update states
        if (!PX4_ISFINITE(_setpoint.z)
            && PX4_ISFINITE(_setpoint.vz) && (fabsf(_setpoint.vz) > FLT_EPSILON)
            && PX4_ISFINITE(local_pos.z_deriv) && local_pos.z_valid && local_pos.v_z_valid) {
                // A change in velocity is demanded and the altitude is not controlled.
                // Set velocity to the derivative of position
                // because it has less bias but blend it in across the landing speed range
                //  <  MPC_LAND_SPEED: ramp up using altitude derivative without a step
                //  >= MPC_LAND_SPEED: use altitude derivative
                float weighting = fminf(fabsf(_setpoint.vz) / _param_mpc_land_speed.get(), 1.f);
                states.velocity(2) = local_pos.z_deriv * weighting + local_pos.vz * (1.f -
                ↪  weighting);
        }
```

Listing 11. MulticopterPositionControl 8

Run() method part 9 (Listing 12).

Set states taken before from local_pos into Position control object (_control). Then the "update" method runs the Position, Velocity and Acceleration control method. If the update method returns 0, activate a "failsafe" behaviour, it checks if setpoint timing is wrong and sets the failure type and the corresponding setpoints to give to the position control in order to solve the problem.

```
        _control.setState(states);

    // Run position control
        if (_control.update(dt)) {
                _failsafe_land_hysteresis.set_state_and_update(false, time_stamp_now);

        } else {
                // Failsafe
                if ((time_stamp_now - _last_warn) > 2_s) {
                ↪  PX4_WARN("invalid setpoints");
                           _last_warn = time_stamp_now;
                }

                vehicle_local_position_setpoint_s failsafe_setpoint{};

                failsafe(time_stamp_now, failsafe_setpoint, states, !was_in_failsafe);

                // reset constraints
                _vehicle_constraints = {0, NAN, NAN, NAN, false, {}};

                _control.setInputSetpoint(failsafe_setpoint);
                _control.setVelocityLimits(_param_mpc_xy_vel_max.get(),
                ↪  _param_mpc_z_vel_max_up.get(), _param_mpc_z_vel_max_dn.get());
                _control.update(dt);
                }
```

Listing 12. MulticopterPositionControl 9

Run() method part 10 (Listing 13).

The method "getLocalPositionSetpoint" simply takes the setpoint computed by the update method and save it to local_pos_sp.
*The method "getAttitudeSetpoint" transforms the thrust and yaw setpoints to a quaternion, needed for the attitude control (**"Yaw to attitude block"**).*
Last part of run method adjusts the reset counter used in the first part of the method (Listing 4).

```
// Publish internal position control setpoints
// on top of the input/feed-forward setpoints these contain the PID corrections
// This message is used by other modules (such as Landdetector) to determine vehicle intention.
                        vehicle_local_position_setpoint_s local_pos_sp{};

                        _control.getLocalPositionSetpoint(local_pos_sp);
                        local_pos_sp.timestamp = hrt_absolute_time();
                        _local_pos_sp_pub.publish(local_pos_sp);

                        // Publish attitude setpoint output
                        vehicle_attitude_setpoint_s attitude_setpoint{};
                        _control.getAttitudeSetpoint(attitude_setpoint);
                        attitude_setpoint.timestamp = hrt_absolute_time();
                        _vehicle_attitude_setpoint_pub.publish(attitude_setpoint);

                } else {
                        // an update is necessary here because otherwise the takeoff state doesn't
                        ↪  get skiped with non-altitude-controlled modes
                        _takeoff.updateTakeoffState(_vehicle_control_mode.flag_armed,
                        ↪  _vehicle_land_detected.landed, false, 10.f, true,
                                                    time_stamp_now);
                }

                // Publish takeoff status
                const uint8_t takeoff_state = static_cast<uint8_t>(_takeoff.getTakeoffState());

                if (takeoff_state != _takeoff_status_pub.get().takeoff_state
                    || !isEqualF(_tilt_limit_slew_rate.getState(),
                    ↪  _takeoff_status_pub.get().tilt_limit)) {
                        _takeoff_status_pub.get().takeoff_state = takeoff_state;
                        _takeoff_status_pub.get().tilt_limit = _tilt_limit_slew_rate.getState();
                        _takeoff_status_pub.get().timestamp = hrt_absolute_time();
                        _takeoff_status_pub.update();
                }

                // save latest reset counters
                _vxy_reset_counter = local_pos.vxy_reset_counter;
                _vz_reset_counter = local_pos.vz_reset_counter;
                _xy_reset_counter = local_pos.xy_reset_counter;
                _z_reset_counter = local_pos.z_reset_counter;
                _heading_reset_counter = local_pos.heading_reset_counter;
        }

        perf_end(_cycle_perf);
}
```

Listing 13. MulticopterPositionControl 10

_positionControl() method (Listing 14).

Computes the velocity setpoint as an element by element multiplication between the position error (xyz) and the proportional gain vector.
Then if the previous velocity setpoint is not NAN (looking element by element) adds the computed one to it, otherwise write on velocity setpoint only the one computed at the first line.
As last thing limits the ***xy*** *term looking if the setpoint plus the delta are more then the maximum value, leaving space for the prioritized* ***z*** *direction.*

```
void PositionControl::_positionControl()
{
        // P-position controller
        Vector3f vel_sp_position = (_pos_sp - _pos).emult(_gain_pos_p);
        // Position and feed-forward velocity setpoints or position states being NAN results in them
        ↪  not having an influence
        ControlMath::addIfNotNanVector3f(_vel_sp, vel_sp_position);
        // make sure there are no NAN elements for further reference while constraining
        ControlMath::setZeroIfNanVector3f(vel_sp_position);

        // Constrain horizontal velocity by prioritizing the velocity component along the
        // the desired position setpoint over the feed-forward term.
        _vel_sp.xy() = ControlMath::constrainXY(vel_sp_position.xy(), (_vel_sp -
        ↪  vel_sp_position).xy(), _lim_vel_horizontal);
        // Constrain velocity in z-direction.
        _vel_sp(2) = math::constrain(_vel_sp(2), -_lim_vel_up, _lim_vel_down);
}
```

Listing 14. PositionControl()

_velocityControl() method (Listing 15).
Compute the PID control action, the acceleration setpoint is the sum of: proportional part computed on the velocity error, integral part computed next in the code (it consider and compensate the hover thrust), and the derivative term computed on the time derivative of the velocity (acceleration), to avoid the "derivative kickback" (an high jump for control action caused by the derivative term [50]).
Then sets the thrust setpoint based on hover_ thrust and the desired direction (computed in "_ ***accelerationControl()****" (Listing 16)).*
Then before computing the integral term, the velocity loop priotitizes the z direction and computes the remaining thrust to allocate for xy as:

$$(XY)^2 = (MAX)^2 - Z^2$$
$$(XY) = \sqrt[2]{(XY)^2}$$

Once the maximum value for the xy thrust is calculated, if the xy thrust computed before is greater, scales it according to the maximum.
In the last part of the code the setpoint is rescaled considering the gravity, and the error between setpoint and the integral action is computed.

```
void PositionControl::_velocityControl(const float dt)
{
        // PID velocity control
        Vector3f vel_error = _vel_sp - _vel;
        Vector3f acc_sp_velocity = vel_error.emult(_gain_vel_p) + _vel_int -
        ↪  _vel_dot.emult(_gain_vel_d);

        // No control input from setpoints or corresponding states which are NAN
        ControlMath::addIfNotNanVector3f(_acc_sp, acc_sp_velocity);

        _accelerationControl();

        // Integrator anti-windup in vertical direction
        if ((_thr_sp(2) >= -_lim_thr_min && vel_error(2) >= 0.0f) ||
            (_thr_sp(2) <= -_lim_thr_max && vel_error(2) <= 0.0f)) {
                vel_error(2) = 0.f;
        }

        // Saturate maximal vertical thrust
        _thr_sp(2) = math::max(_thr_sp(2), -_lim_thr_max);
        // Get allowed horizontal thrust after prioritizing vertical control
        const float thrust_max_squared = _lim_thr_max * _lim_thr_max;
        const float thrust_z_squared = _thr_sp(2) * _thr_sp(2);
        const float thrust_max_xy_squared = thrust_max_squared - thrust_z_squared;
        float thrust_max_xy = 0;

        if (thrust_max_xy_squared > 0) {
                thrust_max_xy = sqrtf(thrust_max_xy_squared);
        }

        // Saturate thrust in horizontal direction
        const Vector2f thrust_sp_xy(_thr_sp);                         // take first two element of
        ↪  _thr_sp
        const float thrust_sp_xy_norm = thrust_sp_xy.norm();     // compute the norm

        if (thrust_sp_xy_norm > thrust_max_xy) {
                _thr_sp.xy() = thrust_sp_xy / thrust_sp_xy_norm * thrust_max_xy;
        }

        // Use tracking Anti-Windup for horizontal direction: during saturation, the integrator is
        ↪  used to unsaturate the output
        // see Anti-Reset Windup for PID controllers, L.Rundqwist, 1990
        const Vector2f acc_sp_xy_limited = Vector2f(_thr_sp) * (CONSTANTS_ONE_G / _hover_thrust);
        const float arw_gain = 2.f / _gain_vel_p(0);
        vel_error.xy() = Vector2f(vel_error) - (arw_gain * (Vector2f(_acc_sp) - acc_sp_xy_limited));

        // Make sure integral doesn't get NAN
        ControlMath::setZeroIfNanVector3f(vel_error);
        // Update integral part of velocity control
        _vel_int += vel_error.emult(_gain_vel_i) * dt;

        // limit thrust integral
        _vel_int(2) = math::min(fabsf(_vel_int(2)), CONSTANTS_ONE_G) * sign(_vel_int(2));
}
```

Listing 15. VelocityControl 1.2

_accelerationControl() method (Listing 16).

It defines z local normalized axis, limits tilt angle with respect to the angle between z fixed direction and local z.

Then the thrust is scaled assuming that the hover_thrust compensates for gravity (the sign is negative since the z direction points downwards).

As last thing the thrust is projected to body z axis, its value is limited, and the thrust setpoint is set.

```
void PositionControl::_accelerationControl()
{
        // Assume standard acceleration due to gravity in vertical direction for attitude generation
        Vector3f body_z = Vector3f(-_acc_sp(0), -_acc_sp(1), CONSTANTS_ONE_G).normalized();
        ControlMath::limitTilt(body_z, Vector3f(0, 0, 1), _lim_tilt);
        // Scale thrust assuming hover thrust produces standard gravity
        float collective_thrust = _acc_sp(2) * (_hover_thrust / CONSTANTS_ONE_G) - _hover_thrust;
        // Project thrust to planned body attitude
        collective_thrust /= (Vector3f(0, 0, 1).dot(body_z));
        collective_thrust = math::min(collective_thrust, -_lim_thr_min);
        _thr_sp = body_z * collective_thrust;
}
```

Listing 16. AccelerationControl()

4.6 Attitude controller

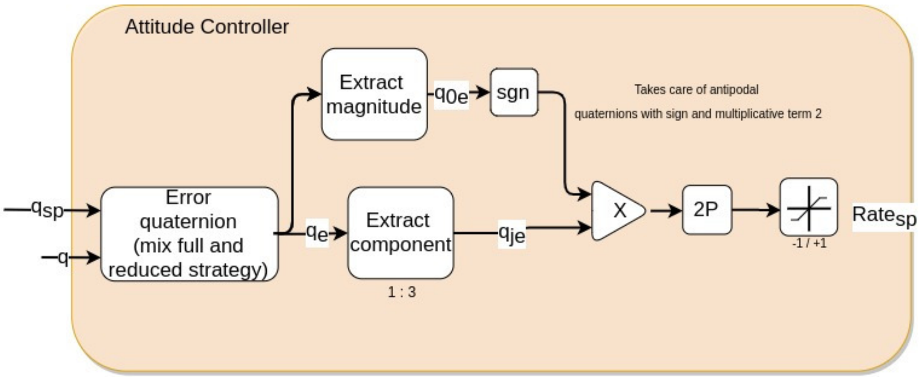


Figure 4.9. Attitude Controller

The multicopter attitude controller (Figure 4.9) takes attitude setpoints as inputs and gives as output the rate setpoint, used by the rate controller.
The control is based on a P loop for angular error which is expressed as a quaternion. The advantage of using a quaternion is to avoid singularities that could happen with simple Euler angles, in fact with quaternions we have four terms to represent three dof (Degree of Freedom) and not only three.
A detailed publication on the quaternion attitude control can be found here [51].

4.6.1 Attitude Controller Code

The **multicopter** *attitude* controller uses the topics in Figure 4.10.

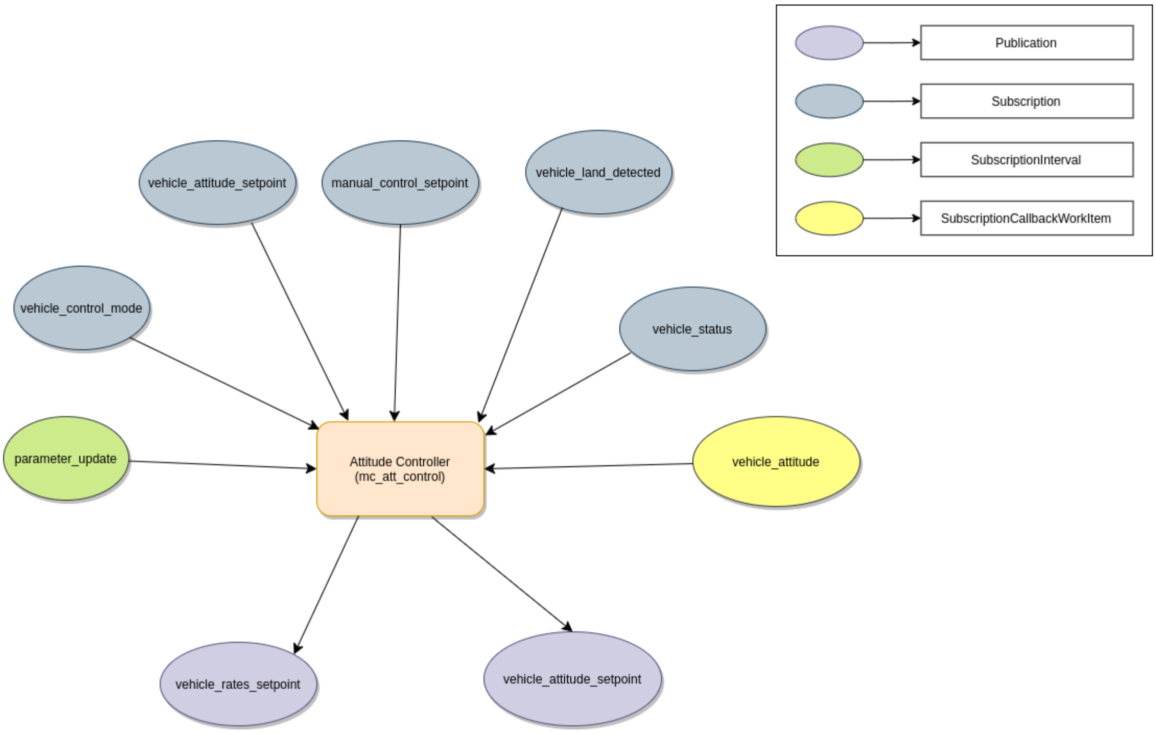


Figure 4.10. Attitude Controller Topics

Below the main methods of the attitude controller and the blocks described above are shown into the code.

Run() method part 1 (Listing 17).
Attitude Control is a simple work item (nav_ and_ controllers priority stack).
This Run method it's called at higher frequency (relative to a Subscription Callback on received data) then position module.
The first part contains only parameters update.

```
void
MulticopterAttitudeControl::Run()
{
        if (should_exit()) {
                _vehicle_attitude_sub.unregisterCallback();
                exit_and_cleanup();
                return;
        }

        perf_begin(_loop_perf);

        // Check if parameters have changed
        if (_parameter_update_sub.updated()) {
                // clear update
                parameter_update_s param_update;
                _parameter_update_sub.copy(&param_update);

                updateParams();
                parameters_updated();
        }
```

Listing 17. Run() Attitude Controller 1

Run() method part 2 (Listing 18).
If new setpoints are found set them in the attitude control object (v_ att).

Then this part checks for heading reset, the quat_reset_counter is compared with the one of the attitude control object, if this counter it's different (not synchronized), the yaw angle is extractet and the setpoint adapted with any available angular deltas (synchronizing the setpoint).

```
// run controller on attitude updates
vehicle_attitude_s v_att;

if (_vehicle_attitude_sub.update(&v_att)) {

        // Check for new attitude setpoint
        if (_vehicle_attitude_setpoint_sub.updated()) {
                vehicle_attitude_setpoint_s vehicle_attitude_setpoint;
                _vehicle_attitude_setpoint_sub.update(&vehicle_attitude_setpoint);
                _attitude_control.setAttitudeSetpoint(Quatf(vehicle_attitude_setpoint.q_d),
                ↪  vehicle_attitude_setpoint.yaw_sp_move_rate);
                _thrust_setpoint_body = Vector3f(vehicle_attitude_setpoint.thrust_body);
        }

        // Check for a heading reset
        if (_quat_reset_counter != v_att.quat_reset_counter) {
                const Quatf delta_q_reset(v_att.delta_q_reset);
        // for stabilized attitude generation only extract the heading change from the delta
        ↪  quaternion
                _man_yaw_sp += Eulerf(delta_q_reset).psi();
                _attitude_control.adaptAttitudeSetpoint(delta_q_reset);

                _quat_reset_counter = v_att.quat_reset_counter;
        }
    // Guard against too small (< 0.2ms) and too large (> 20ms) dt's.
        const float dt = math::constrain(((v_att.timestamp - _last_run) * 1e-6f), 0.0002f, 0.02f);
        _last_run = v_att.timestamp;

        /* check for updates in other topics  as manual setpoint and vehicle control mode*/
        _manual_control_setpoint_sub.update(&_manual_control_setpoint);
        _v_control_mode_sub.update(&_v_control_mode);
```

Listing 18. Run() AttitudeControl 2

Run() method part 3 (Listing 19).

In this part first is checked if the vehicle should land looking at the corresponding state, this state is then saved in _landed boolean variable.
Then in this part is checked if any updates are present in the vehicle status, and taking track of them, it is also checked if we are in hoovering (for rotary wings vehicle) or in tailsitter transition (for vtol), if one of this two states is active, the attitude controller should be runned (the boolean value run_att_ctrl is set to 1).

```
	if (_vehicle_land_detected_sub.updated()) {
		vehicle_land_detected_s vehicle_land_detected;

		if (_vehicle_land_detected_sub.copy(&vehicle_land_detected)) {
			_landed = vehicle_land_detected.landed;
		}
	}

	if (_vehicle_status_sub.updated()) {
		vehicle_status_s vehicle_status;

		if (_vehicle_status_sub.copy(&vehicle_status)) {
			_vehicle_type_rotary_wing = (vehicle_status.vehicle_type ==
			↪  vehicle_status_s::VEHICLE_TYPE_ROTARY_WING);
			_vtol = vehicle_status.is_vtol;
			_vtol_in_transition_mode = vehicle_status.in_transition_mode;
		}
	}

	bool attitude_setpoint_generated = false;

	const bool is_hovering = (_vehicle_type_rotary_wing && !_vtol_in_transition_mode);

	// vehicle is a tailsitter in transition mode
	const bool is_tailsitter_transition = (_vtol_tailsitter && _vtol_in_transition_mode);

	bool run_att_ctrl = _v_control_mode.flag_control_attitude_enabled && (is_hovering ||
	↪  is_tailsitter_transition);
```

Listing 19. Run() AttitudeControl 3

Run() method part 4 (Listing 20).
For Manual control only, it generate the attitude setpoint from stick inputs controlling the tilt angle and limiting its values.

```
	if (run_att_ctrl) {

		const Quatf q{v_att.q};

		// Generate the attitude setpoint from stick inputs if we are in Manual/Stabilized
		↪  mode
		if (_v_control_mode.flag_control_manual_enabled &&
		    !_v_control_mode.flag_control_altitude_enabled &&
		    !_v_control_mode.flag_control_velocity_enabled &&
		    !_v_control_mode.flag_control_position_enabled) {

			generate_attitude_setpoint(q, dt, _reset_yaw_sp);
			attitude_setpoint_generated = true;

		} else {
			_man_x_input_filter.reset(0.f);
			_man_y_input_filter.reset(0.f);
		}
```

Listing 20. Run() AttitudeControl 4

Run() method part 5 (Listing 21).
Runs the attitude control and set a the rate setpoint.
The rate setpoint is computed looking the yaw speed setpoint (computed from quaternions) and then projeted on the body z axis.

```
                Vector3f rates_sp = _attitude_control.update(q);

                // publish rate setpoint
                vehicle_rates_setpoint_s v_rates_sp{};
                v_rates_sp.roll = rates_sp(0);
                v_rates_sp.pitch = rates_sp(1);
                v_rates_sp.yaw = rates_sp(2);
                _thrust_setpoint_body.copyTo(v_rates_sp.thrust_body);
                v_rates_sp.timestamp = hrt_absolute_time();

                _v_rates_sp_pub.publish(v_rates_sp);
        }

        // reset yaw setpoint during transitions, tailsitter.cpp generates
        // attitude setpoint for the transition
        _reset_yaw_sp = !attitude_setpoint_generated || _landed || (_vtol &&
        ↪  _vtol_in_transition_mode);
}

perf_end(_loop_perf);
}
```

Listing 21. Run() AttitudeControl 5

AttitudeControl::update(const Quatf q) (Listing 22).
As already mentioned the attitude controller is based on the idea of using the quaternion substituting the classical yaw, pitch, roll angles.
*One of the possible problem to handle is the fact that the quaternion representation is not unique, each pair of antipodal quaternions ("+-q") corresponds to the same physical attitude, so the sign has to be checked. As last thing the controller computes two different desired attitudes (**reduced** and **full**) and merge them to achieve better performances (**mixed**).*
*The **full** attitude considers the yaw angle using rotation matrix (the orientation given by the position control).*
*For the **reduced** attitude only the crucial pointing direction of the thrust is controlled; the yaw angle is not controlled directly, but the quaternion is always chosen such that no rotation about the yaw axis is induced.*
*It is not necessary that the yaw angle is well followed in order to follow a trajectory (the dynamic around z is much slower), but it is sometimes desirable so the **mixed** reference is preferred. **Note**: given a desired orientation, the error setpoint is taken as the shortest rotation to reach it [51].*

```
matrix::Vector3f AttitudeControl::update(const Quatf &q) const
{
        Quatf qd = _attitude_setpoint_q;

// calculate reduced desired attitude neglecting vehicle's yaw to prioritize roll and pitch
        const Vector3f e_z = q.dcm_z();
        const Vector3f e_z_d = qd.dcm_z();
        Quatf qd_red(e_z, e_z_d);

        if (fabsf(qd_red(1)) > (1.f - 1e-5f) || fabsf(qd_red(2)) > (1.f - 1e-5f)) {
// In the infinitesimal corner case where the vehicle and thrust have completely opposite direction,
// full attitude control anyways generates no yaw input and directly takes the combination of
// roll and pitch leading to the correct desired yaw. Ignoring this case would still be totally safe
↪  and stable.
                qd_red = qd;

        } else {
// transform rotation from current to desired thrust vector into a world frame reduced desired
↪  attitude
                qd_red *= q;
        }

// mix full and reduced desired attitude
        Quatf q_mix = qd_red.inversed() * qd;
        q_mix.canonicalize();
        // catch numerical problems with the domain of acosf and asinf
        q_mix(0) = math::constrain(q_mix(0), -1.f, 1.f);
        q_mix(3) = math::constrain(q_mix(3), -1.f, 1.f);
        qd = qd_red * Quatf(cosf(_yaw_w * acosf(q_mix(0))), 0, 0, sinf(_yaw_w * asinf(q_mix(3))));

// quaternion attitude control law, qe is rotation from q to qd
        const Quatf qe = q.inversed() * qd;

// using sin(alpha/2) scaled rotation axis as attitude error (see quaternion definition by axis
↪  angle)
        // also taking care of the antipodal unit quaternion ambiguity (taking care of the sign)
        const Vector3f eq = 2.f * qe.canonical().imag();

        // calculate angular rates setpoint
        matrix::Vector3f rate_setpoint = eq.emult(_proportional_gain);

// Feed forward the yaw setpoint rate.
// yawspeed_setpoint is the feed forward commanded rotation around the world z-axis,
// but we need to apply it in the body frame (because _rates_sp is expressed in the body frame).
// Therefore we infer the world z-axis (expressed in the body frame) by taking the last column of
↪  R.transposed (== q.inversed)
// and multiply it by the yaw setpoint rate (yawspeed_setpoint).
// This yields a vector representing the commanded rotation around the world z-axis expressed in the
↪  body frame
// such that it can be added to the rates setpoint.
        if (is_finite(_yawspeed_setpoint)) {
                rate_setpoint += q.inversed().dcm_z() * _yawspeed_setpoint;
        }

        // limit rates
        for (int i = 0; i < 3; i++) {
                rate_setpoint(i) = math::constrain(rate_setpoint(i), -_rate_limit(i),
                ↪  _rate_limit(i));
        }

        return rate_setpoint;
}
```

Listing 22. update method attitude Controller

4.7 Rate controller

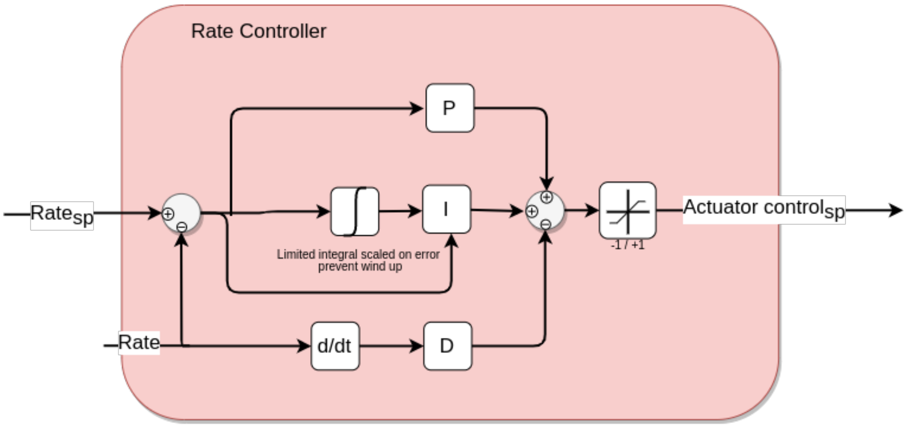


Figure 4.11. Rate Controller

The multicopter rate controller (Figure 4.11) takes the rate setpoint and gives as output the actuator control value. The controller is based on a PID loop, the structure is similar to the VelocityController (Listing 15).

4.7.1 Rate Controller Code

The **multicopter** *rate* controller uses topics in Figure 4.12.

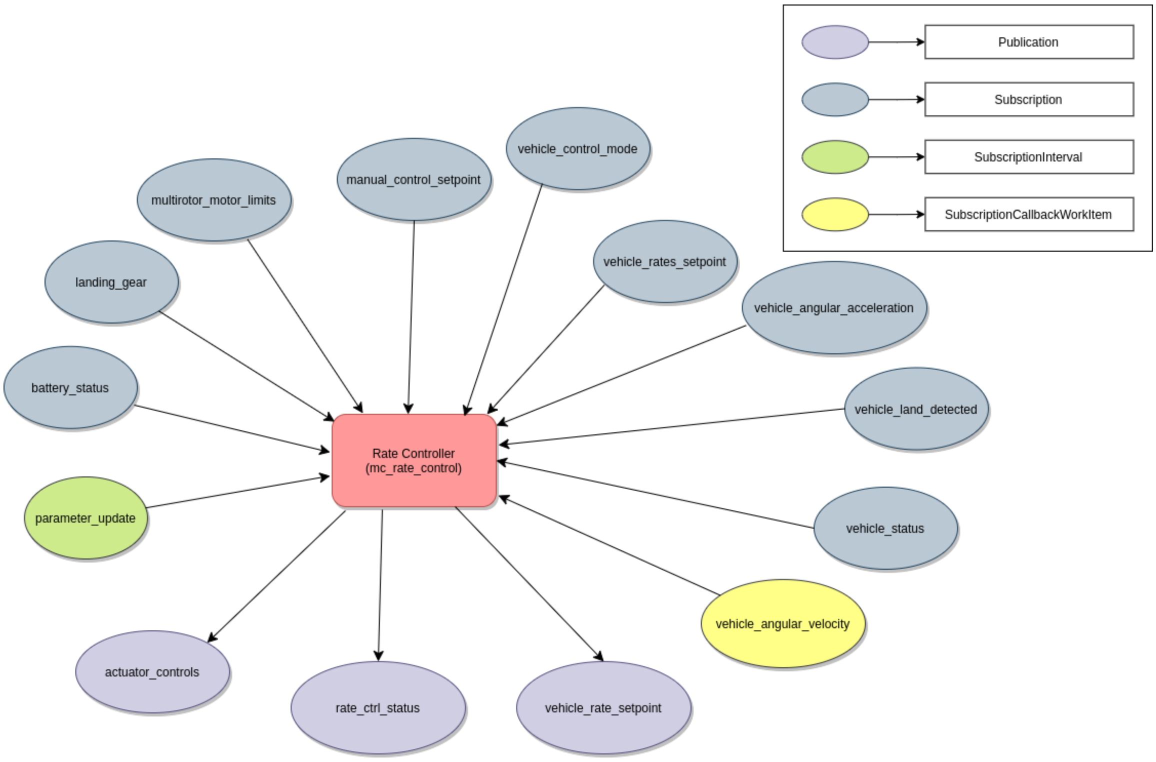


Figure 4.12. Rate Controller Topics

Below the main methods of the rate controller and the blocks described above are shown into the code.

Run() method part 1 (Listing 23).
Rate Control is a simple Work Item (part of rate_ ctrl priority stack which have maximum priority)
The Run method is called on a data Subscription Callback as in the Attitude Controller. It is the most internal control loop and it is responsible for actuator_ control publication, then read by the mixer that computes the actuator outputs adapting the control values to the vehicle geometry.

```
void
MulticopterRateControl::Run()
{
        if (should_exit()) {
                _vehicle_angular_velocity_sub.unregisterCallback();
                exit_and_cleanup();
                return;
        }

        perf_begin(_loop_perf);

        // Check if parameters have changed
        if (_parameter_update_sub.updated()) {
                // clear update
                parameter_update_s param_update;
                _parameter_update_sub.copy(&param_update);

                updateParams();
                parameters_updated();
        }
```

Listing 23. Run() RateController 1

Run() method part 2 (Listing 24).
The controller is called on every gyro changes, the frequency of the used sensor determines also the frequency of the controller.
First the time deltas between calls is computed, the active control mode and landing status is checked, then vehicle status is updated.

```
        /* run controller on gyro changes */
        vehicle_angular_velocity_s angular_velocity;

        if (_vehicle_angular_velocity_sub.update(&angular_velocity)) {

                // grab corresponding vehicle_angular_acceleration immediately after
                ↪  vehicle_angular_velocity copy
                vehicle_angular_acceleration_s v_angular_acceleration{};
                _vehicle_angular_acceleration_sub.copy(&v_angular_acceleration);

                const hrt_abstime now = angular_velocity.timestamp_sample;

                // Guard against too small (< 0.125ms) and too large (> 20ms) dt's.
                const float dt = math::constrain(((now - _last_run) * 1e-6f), 0.000125f, 0.02f);
                _last_run = now;

                const Vector3f angular_accel{v_angular_acceleration.xyz};
                const Vector3f rates{angular_velocity.xyz};

                /* check for updates in other topics */
                _v_control_mode_sub.update(&_v_control_mode);

                if (_vehicle_land_detected_sub.updated()) {
                        vehicle_land_detected_s vehicle_land_detected;

                        if (_vehicle_land_detected_sub.copy(&vehicle_land_detected)) {
                                _landed = vehicle_land_detected.landed;
                                _maybe_landed = vehicle_land_detected.maybe_landed;
                        }
                }
                _vehicle_status_sub.update(&_vehicle_status);

                if (_landing_gear_sub.updated()) {
                        landing_gear_s landing_gear;

                        if (_landing_gear_sub.copy(&landing_gear)) {
                                if (landing_gear.landing_gear != landing_gear_s::GEAR_KEEP) {
                                        _landing_gear = landing_gear.landing_gear;
                                }
                        }
                }
```

Listing 24. Run() RateController 2

Run() method part 3 (Listing 25).
Checks if manual and attitude control are not enabled, in condition is true generate the manual_ control_ setpoints. Then updates the manual stick references and stabilizes the control setpoints for "Acro" mode. If there was no manual setpoints update, the rate setpoints published from the attitude controller are taken.

```
        if (_v_control_mode.flag_control_manual_enabled &&
        ↪  !_v_control_mode.flag_control_attitude_enabled) {
                // generate the rate setpoint from sticks
                manual_control_setpoint_s manual_control_setpoint;

                if (_manual_control_setpoint_sub.update(&manual_control_setpoint)) {
                        // manual rates control - ACRO mode
                        const Vector3f man_rate_sp{
                                math::superexpo(manual_control_setpoint.y,
                                ↪  _param_mc_acro_expo.get(),
                                ↪  _param_mc_acro_supexpo.get()),
                                math::superexpo(-manual_control_setpoint.x,
                                ↪  _param_mc_acro_expo.get(),
                                ↪  _param_mc_acro_supexpo.get()),
                                math::superexpo(manual_control_setpoint.r,
                                ↪  _param_mc_acro_expo_y.get(),
                                ↪  _param_mc_acro_supexpoy.get())};

                        _rates_sp = man_rate_sp.emult(_acro_rate_max);
                        _thrust_sp = math::constrain(manual_control_setpoint.z, 0.0f, 1.0f);

                        // publish rate setpoint
                        vehicle_rates_setpoint_s v_rates_sp{};
                        v_rates_sp.roll = _rates_sp(0);
                        v_rates_sp.pitch = _rates_sp(1);
                        v_rates_sp.yaw = _rates_sp(2);
                        v_rates_sp.thrust_body[0] = 0.0f;
                        v_rates_sp.thrust_body[1] = 0.0f;
                        v_rates_sp.thrust_body[2] = -_thrust_sp;
                        v_rates_sp.timestamp = hrt_absolute_time();

                        _v_rates_sp_pub.publish(v_rates_sp);
                }

        } else {
                // use rates setpoint topic
                vehicle_rates_setpoint_s v_rates_sp;

                if (_v_rates_sp_sub.update(&v_rates_sp)) {
                        _rates_sp(0) = PX4_ISFINITE(v_rates_sp.roll)  ? v_rates_sp.roll  :
                        ↪  rates(0);
                        _rates_sp(1) = PX4_ISFINITE(v_rates_sp.pitch) ? v_rates_sp.pitch :
                        ↪  rates(1);
                        _rates_sp(2) = PX4_ISFINITE(v_rates_sp.yaw)   ? v_rates_sp.yaw   :
                        ↪  rates(2);
                        _thrust_sp = -v_rates_sp.thrust_body[2];
                }
        }
```

Listing 25. Run() RateController 3

Run() method part 4 (Listing 26).
In this part is checked if the rate controller is enabled and not circuit breaker is detected (actuators works fine), in both are valid the controller is run in autonomous flight mode.
Then is checked if any saturation was detected in the previous cycle, and this saturation status is set (for roll, pitch, yaw) into the Rate controller object. Last part runs the actuator control and sets yaw, pitch and roll control value.

```
// run the rate controller
if (_v_control_mode.flag_control_rates_enabled &&
↪  !_actuators_0_circuit_breaker_enabled) {

        // reset integral if disarmed
        if (!_v_control_mode.flag_armed || _vehicle_status.vehicle_type !=
        ↪  vehicle_status_s::VEHICLE_TYPE_ROTARY_WING) {
                _rate_control.resetIntegral();
        }

        // update saturation status from mixer feedback
        if (_motor_limits_sub.updated()) {
                multirotor_motor_limits_s motor_limits;

                if (_motor_limits_sub.copy(&motor_limits)) {
                        MultirotorMixer::saturation_status saturation_status;
                        saturation_status.value = motor_limits.saturation_status;

                        _rate_control.setSaturationStatus(saturation_status);
                }
        }

        // run rate controller
        const Vector3f att_control = _rate_control.update(rates, _rates_sp,
        ↪  angular_accel, dt, _maybe_landed || _landed);

        // publish rate controller status
        rate_ctrl_status_s rate_ctrl_status{};
        _rate_control.getRateControlStatus(rate_ctrl_status);
        rate_ctrl_status.timestamp = hrt_absolute_time();
        _controller_status_pub.publish(rate_ctrl_status);

        // publish actuator controls
        actuator_controls_s actuators{};
        actuators.control[actuator_controls_s::INDEX_ROLL] =
        ↪  PX4_ISFINITE(att_control(0)) ? att_control(0) : 0.0f;
        actuators.control[actuator_controls_s::INDEX_PITCH] =
        ↪  PX4_ISFINITE(att_control(1)) ? att_control(1) : 0.0f;
        actuators.control[actuator_controls_s::INDEX_YAW] =
        ↪  PX4_ISFINITE(att_control(2)) ? att_control(2) : 0.0f;
        actuators.control[actuator_controls_s::INDEX_THROTTLE] =
        ↪  PX4_ISFINITE(_thrust_sp) ? _thrust_sp : 0.0f;
        actuators.control[actuator_controls_s::INDEX_LANDING_GEAR] = _landing_gear;
        actuators.timestamp_sample = angular_velocity.timestamp_sample;
```

Listing 26. Run() RateController 4

Run() method part 5 (Listing 27).

The main thing done in this part is to scale control action with respect to the battery status (that varies between 0 and 1 respectively low and full battery), multiplying the the control values for the battery scale.

```
                        // scale effort by battery status if enabled
                        if (_param_mc_bat_scale_en.get()) {
                                if (_battery_status_sub.updated()) {
                                        battery_status_s battery_status;

                                        if (_battery_status_sub.copy(&battery_status)) {
                                                _battery_status_scale = battery_status.scale;
                                        }
                                }

                                if (_battery_status_scale > 0.0f) {
                                        for (int i = 0; i < 4; i++) {
                                                actuators.control[i] *= _battery_status_scale;
                                        }
                                }
                        }

                        actuators.timestamp = hrt_absolute_time();
                        _actuators_0_pub.publish(actuators);

                } else if (_v_control_mode.flag_control_termination_enabled) {
                        if (!_vehicle_status.is_vtol) {
                                // publish actuator controls
                                actuator_controls_s actuators{};
                                actuators.timestamp = hrt_absolute_time();
                                _actuators_0_pub.publish(actuators);
                        }
                }
        }

        perf_end(_loop_perf);
}
```

Listing 27. Run() RateController 5

RateControl::update(...) (Listing 28).
Compute the error between rate measurement and setpoint, calculate the proportional part on the error, the derivative on the angular acceleration, add the integral part computed separately and add the feed-forward term on the rate_ setpoint (used only for helicopters).

```
Vector3f RateControl::update(const Vector3f &rate, const Vector3f &rate_sp, const Vector3f
↪  &angular_accel,
                             const float dt, const bool landed)
{
        // angular rates error
        Vector3f rate_error = rate_sp - rate;

        // PID control with feed forward
        const Vector3f torque = _gain_p.emult(rate_error) + _rate_int - _gain_d.emult(angular_accel)
        ↪  + _gain_ff.emult(rate_sp);

        // update integral only if we are not landed
        if (!landed) {
                updateIntegral(rate_error, dt);
        }

        return torque;
}
```

Listing 28. Rate controller update()

RateControl::updateIntegral(...) (Listing 29).
In this method the integral part of the PID control action, given by the rate controller, is updated.
First is checked is any saturation was detected, if is the case, then further saturation is prevented modifying the error to zero. In order to prevent saturation as well, the

integral gain is mapped with respect to the error, as the error increases the integral gain is reduced .

```
void RateControl::updateIntegral(Vector3f &rate_error, const float dt)
{
	for (int i = 0; i < 3; i++) {
		// prevent further positive control saturation
		if (_mixer_saturation_positive[i]) {
			rate_error(i) = math::min(rate_error(i), 0.f);
		}

		// prevent further negative control saturation
		if (_mixer_saturation_negative[i]) {
			rate_error(i) = math::max(rate_error(i), 0.f);
		}

// I term factor: reduce the I gain with increasing rate error.
// This counteracts a non-linear effect where the integral builds up quickly upon a large setpoint
// change (noticeable in a bounce-back effect after a flip).
// The formula leads to a gradual decrease w/o steps, while only affecting the cases where it
↪  should:
// with the parameter set to 400 degrees, up to 100 deg rate error, i_factor is almost 1 (having no
↪  effect),
// and up to 200 deg error leads to <25% reduction of I.
		float i_factor = rate_error(i) / math::radians(400.f);
		i_factor = math::max(0.0f, 1.f - i_factor * i_factor);

		// Perform the integration using a first order method
		float rate_i = _rate_int(i) + i_factor * _gain_i(i) * rate_error(i) * dt;

		// do not propagate the result if out of range or invalid
		if (PX4_ISFINITE(rate_i)) {
			_rate_int(i) = math::constrain(rate_i, -_lim_int(i), _lim_int(i));
		}
	}
}
```

Listing 29. Rate controller updateIntegral()

4.8 Mixer

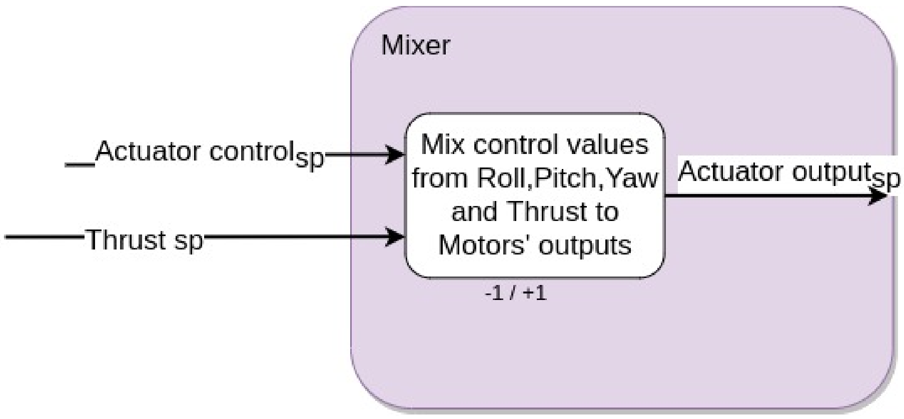


Figure 4.13. Mixer Block

Mixing means translating high-level commands into actuator inputs (Figure 4.13). Separating the mixer logic from the Attitude and Rate controllers strongly improves reusability, since the same Attitude/Rate controller can be used to handle different airframes, where each airframe has a specific mixer file, namely a custom way to convert high-level inputs to actuator commands.
The control pipeline can be summarized in this way: the rate controller sends a specific

normalized force or torque query (scaled within the range -1 to +1) to the mixer, which then converts the input value into individual actuator commands.
Right after, the output driver (generally UART, UAVCAN or PWM) will scale it to the actuators specific units, for instance, considering a PWM driver, it may be a value of 1300, which is sent to the actual actuator after being converted in a compatible signal [31].

4.8.1 Control Groups

PX4 uses control groups and output groups to manage inputs and outputs respectively. A control group is used for the flight controls, or payloads (gimbal for instance). On the other hand, an output group is a physical bus, e.g. the first 8 PWM outputs for servos and motors (Figure 4.15). Each of these groups has 8 normalized (from -1 to +1) output command ports, that can be mapped and scaled through the mixer.
The control group used by default for multicopter, and that was used for this work, is the control group 0 (see Figure 4.14), where the first 4 input are the control values given by the controllers and the last four inputs are usually used for manual commands on auxiliary (AUX) servos, for example flaps.

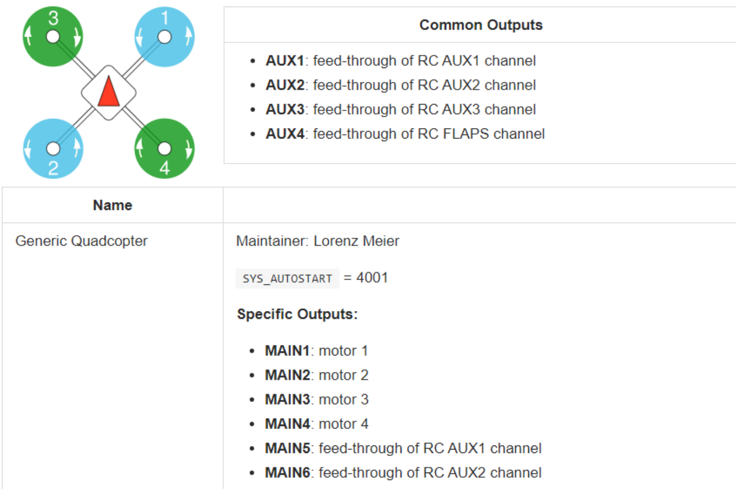


Figure 4.14. Generic Quadcopter Outputs

Source: PX4 Developer Guide

Control Group #0 (Flight Control)

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

Figure 4.15. Control Group #0 (Flight Control)

Source: PX4 Developer Guide

Control Group #3 (Manual Passthrough)

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

Figure 4.16. Control Group #3 (Manual Passthrough)

Source: PX4 Developer Guide

Note: the group in Figure 4.16 is only used to define mapping of RC inputs to specific outputs during normal operation. In the event of manual IO failsafe override this mapping.

4.8.2 Multirotor Mixer

As far as the multirotor mixers is concerned, things are a bit different from the other mixers, because roll, pitch and yaw cannot be controlled directly acting on a single motor.
The multirotor mixer combines four control inputs (roll, pitch, yaw, thrust) within the range -1 to +1, except thrust that is between 0 and 1, into a bundle of actuator outputs intended to drive motor speeds [31].
Whenever an actuator saturates, all actuator values are rescaled so that the saturating component is limited to 1.0 [52].

Geometry file
The standard matrix used by mixer for quadrotor, “quad_x”, is showed in Figure (30),

it define the allocation matrix mappig from control inputs to actuator outputs. This matrix is generated by a python script written in "MultirotorMixer" folder, that takes as input the geometry file showed in Listing 31.

```
 const MultirotorMixer::Rotor _config_quad_x[] = {
        { -0.707107,  0.707107,  1.000000,  1.000000 },
        {  0.707107, -0.707107,  1.000000,  1.000000 },
        {  0.707107,  0.707107, -1.000000,  1.000000 },
        { -0.707107, -0.707107, -1.000000,  1.000000 },
};
```

Listing 30. Multirotor mixer matrix quad_x

```
# Generic Quadcopter in X configuration
[info]
key = "4x"
description = "Generic Quadcopter in X configuration"

[rotor_default]
direction = "CW"
axis      = [0.0, 0.0, -1.0]
Ct        = 1.0
Cm        = 0.05

[[rotors]]
name      = "front_right"
position  = [0.707107, 0.707107, 0.0]
direction = "CCW"

[[rotors]]
name      = "rear_left"
position  = [-0.707107, -0.707107, 0.0]
direction = "CCW"

[[rotors]]
name     = "front_left"
position = [0.707107, -0.707107, 0.0]

[[rotors]]
name     = "rear_right"
position = [-0.707107, 0.707107, 0.0]
```

Listing 31. Multirotor mixer matrix quad_x

Mix file Sintax

A mixing file can be seen in Listing 32, in order to understand it, in the following the used tags are explained:

- **R**: multirotor mixer, it combines four control inputs (roll, pitch, yaw, thrust) into a set of actuator outputs intended to drive motor speed controllers.

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

- **M**: summing mixer, used for actuator and servo control, a summing (simple) mixer combines zero or more control inputs into a single actuator output.

```
M: <control count>
```

- **S**: means the same tag used above describing the control inputs and their scaling, in the form:

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

- **Z**: null mixer, it consumes no controls and generates a single actuator output with a value that is always zero.It may also be used to control the value of an output used for a failsafe device (the output is 0 in normal use; during failsafe the mixer is ignored and a failsafe value is used instead).

```
Z:
```

Once defined the geometry, a mixer file can be created and modified, named XXX.main.mix, responsible for the mixing of MAIN outputs, or XXX.aux.mix if it mixes AUX outputs. Each of the roll, pitch and yaw scale values determines scaling of the roll, pitch and yaw controls relative to the thrust control. Whilst the calculations are performed as floating-point operations, the values stored in the mix file are scaled by a factor of 10000; i.e. a factor of 0.5 is encoded as 5000 [31].
Idle speed can range from 0.0 to 1.0. Idle speed is relative to the maximum speed of motors and it is the speed at which the motors are commanded to rotate when all control inputs are zero.

Mix file example

An example of a mixing file is the one associated with the standard quadrocopter quad x.main.mix, as is shown in Listing 32

```
R: 4x  10000 10000 10000 0

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

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

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

Listing 32. Quadrotor geometry quad_x.main.mix file

The first line defines the geometry (4x which can be seen in Listing 31), and the scaling factor for the three allowed motions (roll, pitch and yaw), plus the idle speed 0, so no motor speed (= 0) whenever there is no other input [31].
The AUX (Auxiliary outputs) are mostly used to manually command servomotors added to the quadcopter, they can be used to command flaps, gimbal camera or other.

AUX1 Passthrough is defined with a single input (M: 1), therefore only one "S:" is defined (S describes which is the control input and its scaling). In this case S: 3 5 points at control group 3 (manual passthrough, Figure 4.16), element 5 (RC aux1, mapped by parameter RC MAP AUX1). If you want to associate a specific RC channel to aux1 you should change parameter RC MAP AUX1.
Thus pin 5 will follow the RC command associated with the selected channel, for instance, choosing channel 12, pin 5 will follow RC movements in channel 12. Then the following numbers define the mapping between the input and the output. Both the first and the second ones are 10000, meaning that no sign inversion is needed (since it is a positive value), and that the slope is standard. The third value is the offset; being it 0, nothing shall be added. Last two values are -10000 and 10000, therefore the output range is maximum (1000 to 2000 that is the standard PWM range used).
AUX2 Passthrough is defined with a single input (M: 1), therefore only one "S:" is defined. In this case S: 3 6 points at control group 3 (manual passthrough) element 6 (RC aux2, mapped by parameter RC MAP AUX2). If you want to associate a specific RC channel to aux2 you should do the same thing said for AUX1.

Note: New custom mixing files, geometries and airframes can be added to PX4 architecture [53], [17].

4.8.3 Mixer Code

The **mixer** uses the topics shown in Figure 4.17.

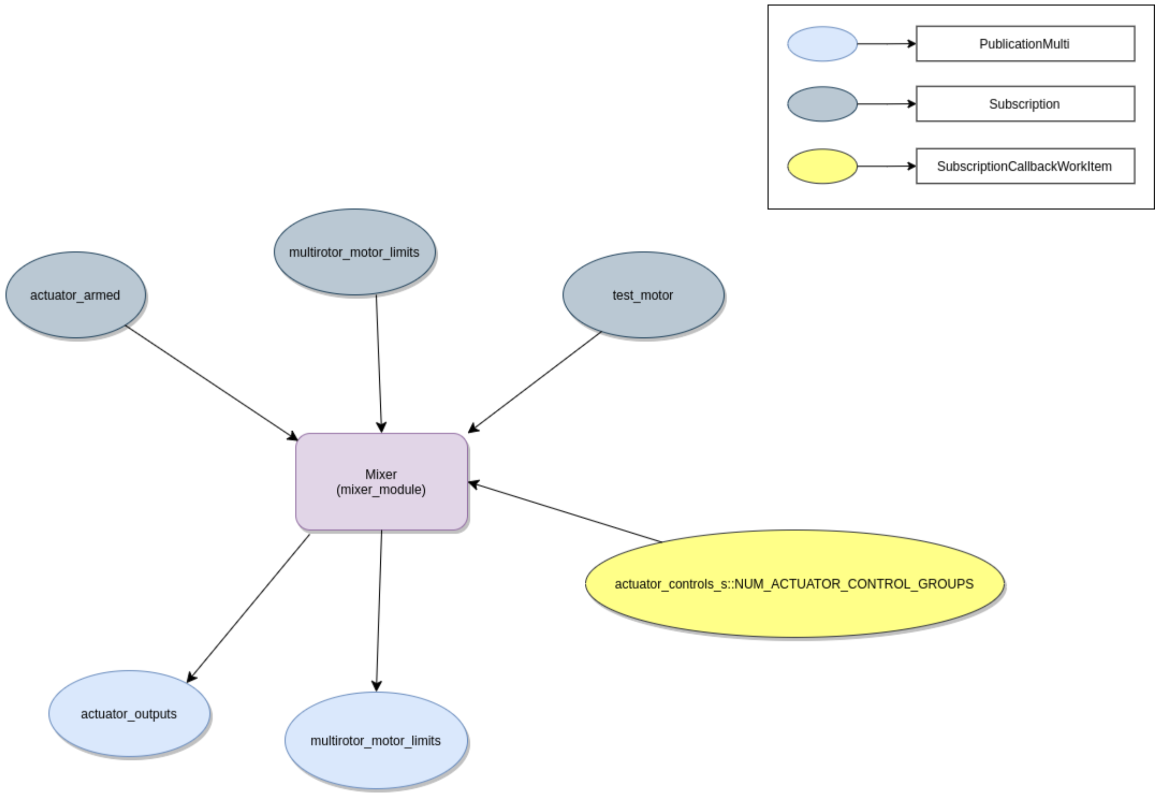


Figure 4.17. Mixer Topics

In the code below the main methods for mixed module and the mixing phase for the multicopter are shown.

update() method part 1 (Listing 33).
It is called regularly in the Run() method by main.cpp of PCA9685 module (see Listing 45) or by PWMSIM (see Listing 41) (if in SITL mode), its function is to mix the actuator control value into actuator outputs to be sent to motors.
The mixing phase is relative to the geometry used for the vehicle and maps the thrust and 3 torques into commands to each motor.
These command values will be first scaled in PWM value for motor, then if SITL mode is active will be rescaled between 0 and 1 (the range value used by simulators).
In this first part of the code it's simply checked the arming state.

```
bool MixingOutput::update()
{
        if (!_mixers) {
                handleCommands();
                // do nothing until we have a valid mixer
                return false;
        }

        // check arming state
        if (_armed_sub.update(&_armed)) {
                _armed.in_esc_calibration_mode &= _support_esc_calibration;

                if (_ignore_lockdown) {
                        _armed.lockdown = false;
                }

                /* Update the armed status and check that we're not locked down.
                 * We also need to arm throttle for the ESC calibration. */
                _throttle_armed = (_armed.armed && !_armed.lockdown) ||
                ↪  _armed.in_esc_calibration_mode;

                if (_armed.armed) {
                        _motor_test.in_test_mode = false;
                }
        }
```

Listing 33. MixingOutput update() 1

update() method part 2 (Listing 34).
Looking at the motor max slew rate (response time for motor) decide to use the multirotor mixer or the slew rate only. the multiroror mixer set the maximum delta allowed in this cycle, it is done to limit the rate, while the slew rate mixer does not.
Then takes the control value from the actuator control topic and puts them into "_controls".

```
if (_param_mot_slew_max.get() > FLT_EPSILON) {
        updateOutputSlewrateMultirotorMixer();
}
// update dt for output slew rate in simple mixer
updateOutputSlewrateSimplemixer();
unsigned n_updates = 0;

/* get controls for required topics */
for (unsigned i = 0; i < actuator_controls_s::NUM_ACTUATOR_CONTROL_GROUPS; i++) {
        if (_groups_subscribed & (1 << i)) {
                if (_control_subs[i].copy(&_controls[i])) {
                        n_updates++;
                }

                /* During ESC calibration, we overwrite the throttle value. */
                if (i == 0 && _armed.in_esc_calibration_mode) {

                        /* Set all controls to 0 */
                        memset(&_controls[i], 0, sizeof(_controls[i]));

                        /* except thrust to maximum ,the index value is 3 . */
                        _controls[i].control[actuator_controls_s::INDEX_THROTTLE] = 1.0f;
                        /* Switch off the output limit ramp for the calibration. */
                        _output_limit.state = OUTPUT_LIMIT_STATE_ON;
                }
        }
}
```

Listing 34. MixingOutput update() 2

update() method part 3 (Listing 35).

This part makes tests the motors (if not motor armed); "updateOutputs" is called on PCA9685 driver (or PWMSIM in SITL), it sets the PWM values on motors in order to test if they work fine, then returns true if everything worked.

```
        if (!_armed.armed && !_armed.manual_lockdown) {
                unsigned num_motor_test = motorTest();
                if (num_motor_test > 0) {
                        if (_interface.updateOutputs(false, _current_output_value, num_motor_test,
                        ↪  1)) {
                                actuator_outputs_s actuator_outputs{};
                                setAndPublishActuatorOutputs(num_motor_test, actuator_outputs);
                        }

                        handleCommands();
                        return true;
                }
        }
```

Listing 35. MixingOutput update() 3

update() method part 4 (Listing 36).

The method "mix", called on mixer object, is a recursive function that give as output the number of channels of the mixer (the value is 8 for multirotors, see Figure 4.14) The function is called on the loaded mixer, for example if a multirotor mixer is loaded this function calls the multirotor mixing method recursevely.
The mix method for multirotor takes the actuator control values and mixs them according to the allocation matrix keeping a scaling value saturated between 0 and 1. A Callback is also called in each mixing phase, it is needed to read the control values set before into "_control".
Then the method "output_limit_calc" takes the output values and scales them in PWM and stores the value in "_current_output_values", then published in actuator outputs topic.

```
        /* do mixing */
        float outputs[MAX_ACTUATORS] {};
        const unsigned mixed_num_outputs = _mixers->mix(outputs, _max_num_outputs);

        /* the output limit call takes care of out of band errors, NaN and constrains */
        output_limit_calc(_throttle_armed, armNoThrottle(), mixed_num_outputs, _reverse_output_mask,
                          _disarmed_value, _min_value, _max_value, outputs, _current_output_value,
                          ↪  &_output_limit);

        /* overwrite outputs in case of force_failsafe with _failsafe_value values */
        if (_armed.force_failsafe) {
                for (size_t i = 0; i < mixed_num_outputs; i++) {
                        _current_output_value[i] = _failsafe_value[i];
                }
        }
```

Listing 36. MixingOutput update() 4

update() method part 5 (Listing 37).
Set the flag "stop_motors" to 1 if is not armed or some error happens in the mix method above.
Then sort (if needed) the outputs with respect to motors definition (actual position of motor 1,2,3,4).
Then the method "setAndPublishActuatorOutputs()" publish the actuator outputs scaled in PWM (this value will be rescaled between 0 and 1 if we are in SITL mode since the gazebo plugin takes scaled inputs).
Then publish the actuator status (saturation detected/not).

```
        bool stop_motors = mixed_num_outputs == 0 || !_throttle_armed;

        /* overwrite outputs in case of lockdown or parachute triggering with disarmed values */
        if (_armed.lockdown || _armed.manual_lockdown) {
                for (size_t i = 0; i < mixed_num_outputs; i++) {
                        _current_output_value[i] = _disarmed_value[i];
                }

                stop_motors = true;
        }

        /* apply _param_mot_ordering */
        reorderOutputs(_current_output_value);

        /* now return the outputs to the driver */
        if (_interface.updateOutputs(stop_motors, _current_output_value, mixed_num_outputs,
        ↪  n_updates)) {
                actuator_outputs_s actuator_outputs{};
                /* simply publish the Actuator outputs values */
                setAndPublishActuatorOutputs(mixed_num_outputs, actuator_outputs);

                publishMixerStatus(actuator_outputs);
                updateLatencyPerfCounter(actuator_outputs);
        }

        handleCommands();

        return true;
}
```

Listing 37. MixingOutput update() 5

MultirotorMixer::mix(...) (Listing 38).
First takes the control values given by the rate controller calling a CallBack defined in the mixer module.
Then it mixes the values using the scale given by the allocation matrix respect the different airmodes.
Then saturate again the control values between -1 and 1.

```
unsigned
MultirotorMixer::mix(float *outputs, unsigned space)
{
        if (space < _rotor_count) {
                return 0;
        }

        float roll    = math::constrain(get_control(0, 0), -1.0f, 1.0f);
        float pitch   = math::constrain(get_control(0, 1), -1.0f, 1.0f);
        float yaw     = math::constrain(get_control(0, 2), -1.0f, 1.0f);
        float thrust  = math::constrain(get_control(0, 3), 0.0f, 1.0f);

        // clean out class variable used to capture saturation
        _saturation_status.value = 0;

        // Do the mixing using the strategy given by the current Airmode configuration
        switch (_airmode) {
        case Airmode::roll_pitch:
                mix_airmode_rp(roll, pitch, yaw, thrust, outputs);
                break;

        case Airmode::roll_pitch_yaw:
                mix_airmode_rpy(roll, pitch, yaw, thrust, outputs);
                break;

        case Airmode::disabled:
        default: // just in case: default to disabled
                mix_airmode_disabled(roll, pitch, yaw, thrust, outputs);
                break;
        }

        // Apply thrust model and scale outputs to range [idle_speed, 1].
        // At this point the outputs are expected to be in [0, 1], but they can be outside, for
        ↪  example
        // if a roll command exceeds the motor band limit.
        for (unsigned i = 0; i < _rotor_count; i++) {
                // Implement simple model for static relationship between applied motor pwm and
                ↪  motor thrust
                // model: thrust = (1 - _thrust_factor) * PWM + _thrust_factor * PWM^2
                if (_thrust_factor > 0.0f) {
                        outputs[i] = -(1.0f - _thrust_factor) / (2.0f * _thrust_factor) +
                        ↪  sqrtf((1.0f - _thrust_factor) *
                                        (1.0f - _thrust_factor) / (4.0f * _thrust_factor *
                                        ↪  _thrust_factor) + (outputs[i] < 0.0f ? 0.0f : outputs[i]
                                        ↪  /
                                                    _thrust_factor));
                }

                outputs[i] = math::constrain((2.f * outputs[i] - 1.f), -1.f, 1.f);
        }
//...CODE
}
```

Listing 38. Mix() method for multirotor

MultirotorMixer::mix_airmode_rpy(...) (Listing 39). *Motor outputs are computed multiplying roll, pitch, yaw and thrust values (between -1 and 1) by the corresponding scale (Listing 32).*

```
void
MultirotorMixer::mix_airmode_rpy(float roll, float pitch, float yaw, float thrust, float *outputs)
{
        // Airmode for roll, pitch and yaw

        // Do full mixing
        for (unsigned i = 0; i < _rotor_count; i++) {
                outputs[i] = roll * _rotors[i].roll_scale +
                             pitch * _rotors[i].pitch_scale +
                             yaw * _rotors[i].yaw_scale +
                             thrust * _rotors[i].thrust_scale;

                // Thrust will be used to unsaturate if needed
                _tmp_array[i] = _rotors[i].thrust_scale;
        }

        minimize_saturation(_tmp_array, outputs, _saturation_status);

        // Unsaturate yaw (in case upper and lower bounds are exceeded)
        // to prioritize roll/pitch over yaw.
        for (unsigned i = 0; i < _rotor_count; i++) {
                _tmp_array[i] = _rotors[i].yaw_scale;
        }

        minimize_saturation(_tmp_array, outputs, _saturation_status);
}
```

Listing 39. mix airmode roll, pitch ,yaw

output_limit_calc(...) (Listing 40).
Here is showed the transformation from the actuator outputs after the mixing phase (Listing 36) (values are between 0 and 1) into PWM values.
The resulting value is saved in "effective_output" where the default min and max are set respectively to 1000 and 2000 (PWM per us).

```
void output_limit_calc(const bool armed, const bool pre_armed, const unsigned num_channels, const
↪  uint16_t reverse_mask,
                       const uint16_t *disarmed_output, const uint16_t *min_output, const uint16_t
                       ↪  *max_output,
                       const float *output, uint16_t *effective_output, output_limit_t *limit)
{
//...other code
case OUTPUT_LIMIT_STATE_ON:
                for (unsigned i = 0; i < num_channels; i++) {
                        float control_value = output[i];

                        /* check for invalid / disabled channels */
                        if (!PX4_ISFINITE(control_value)) {
                                effective_output[i] = disarmed_output[i];
                                continue;
                        }

                        if (reverse_mask & (1 << i)) {
                                control_value = -1.0f * control_value;
                        }

                        effective_output[i] = control_value * (max_output[i] - min_output[i]) / 2 +
                        ↪  (max_output[i] + min_output[i]) / 2;

                        /* last line of defense against invalid inputs */
                        if (effective_output[i] < min_output[i]) {
                                effective_output[i] = min_output[i];

                        } else if (effective_output[i] > max_output[i]) {
                                effective_output[i] = max_output[i];
                        }
                }
                break;
}
```

Listing 40. output_limit_calc() used in mixer module

4.9 Actuator

The actuator (px4io , pwm_out, pwm_9685_pwm_outputs and pwm_sim depending on the board and if in SITL, HITL or SIH) subscribe to the actuator output topic through the mixer and publish the PWM outputs to the motors via Mavlink messages.

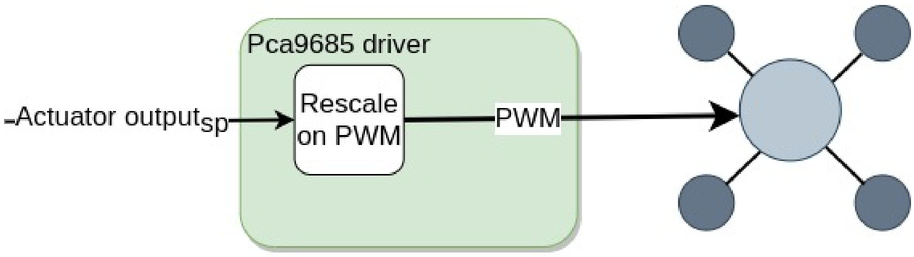


Figure 4.18. Actuator scheme

4.9.1 Actuator Code

Below the main used method are showed, remember that not all the module showed in the next pages are used simultaneously, e.g for SITL only PWMSIM driver and simulator module are used.

Run() method (Listing 41).
in this run method for simulation, simply the "update()" function is called on the mixer object. The same is done when using the pixhawk autopilot, but with different drivers and sending the actuator output values, transformed in PWM, directly to the real motors.

```
void
PWMSim::Run()
{
        if (should_exit()) {
                ScheduleClear();
                _mixing_output.unregister();

                exit_and_cleanup();
                return;
        }

        _mixing_output.update();

        // check for parameter updates
        if (_parameter_update_sub.updated()) {
                parameter_update_s pupdate;
                _parameter_update_sub.copy(&pupdate);
                updateParams();
        }

        // check at end of cycle (updateSubscriptions() can potentially change to a different
        ↪  WorkQueue thread)
        _mixing_output.updateSubscriptions(true);
}
```

Listing 41. PWMSim Run()

PWMSim::updateOutputs(...) (Listing 42).
This method is called from the update() method of mixing output (Listing 36), in simulation the updateOutputs does nothing (it is kept only for simplicity, in order to not change the code sequence).

```
bool
PWMSim::updateOutputs(bool stop_motors, uint16_t outputs[MAX_ACTUATORS], unsigned num_outputs,
                      unsigned num_control_groups_updated)
{
        // Nothing to do, as we are only interested in the actuator_outputs topic publication.
        // That should only be published once we receive actuator_controls (important for lock-step
        ↪  to work correctly)
        return num_control_groups_updated > 0;
}
```

Listing 42. PWMSim::updateOutputs

Simulator::run() (Listing 43).
It handles the interface between the firmware and the simulator (Gazebo/Jmavsim), for example the method send_controls() takes the actuator outputs topic, and send its values to the simulator as a MAVLINK message.

```
void Simulator::run()
{
//...CODE
                if (fds_actuator_outputs[0].revents & POLLIN) {
                        // Got new data to read, update all topics.
                        parameters_update(false);
                        check_failure_injections();
                        _vehicle_status_sub.update(&_vehicle_status);

                        // Wait for other modules, such as logger or ekf2
                        px4_lockstep_wait_for_components();

                        send_controls();
```

Listing 43. Simulator Run()

Simulator::send_controls() (Listing 44).
*The simulator before to send MAVLINK message to Gazebo, calls the method "**actuator_controls_from_outputs**", that re-transform the outputs from PWM values to a value scaled between 0 and 1, as can be seen in the equation below:*

$$controls = \frac{outputs - PWMMIN}{PWMMAX - PWMMIN} \tag{4.2}$$

```
void Simulator::send_controls()
{
        orb_copy(ORB_ID(actuator_outputs), _actuator_outputs_sub, &_actuator_outputs);

        if (_actuator_outputs.timestamp > 0) {
                mavlink_hil_actuator_controls_t hil_act_control;
                actuator_controls_from_outputs(&hil_act_control);

                mavlink_message_t message{};
                mavlink_msg_hil_actuator_controls_encode(_param_mav_sys_id.get(),
                ↪  _param_mav_comp_id.get(), &message, &hil_act_control);

                PX4_DEBUG("sending controls t=%ld (%ld)", _actuator_outputs.timestamp,
                ↪  hil_act_control.time_usec);

                send_mavlink_message(message);
        }
}
```

Listing 44. Simulator::send_controls()

Note: the next snippets of code are here only for completeness, they are not used in SITL simulation. They will partially show the driver for "PCA9685" PWM module, a chip used for IO communication with motors.

PCA9685Wrapper::Run() (Listing 45).
The following part of codes shows the actuation part for PCA9685 [54] (used for pixhawk autopilot to handle data communication).

```
void PCA9685Wrapper::Run()
{

//...CODE (exit condition) //

	perf_begin(_cycle_perf);
	switch (_state) {

	case STATE::INIT:
		pca9685->initReg();mixing_output.mix
		updatePWMParams();  // target frequency fetched, immediately apply it
		if (_targetFreq > 0.0f) {
			if (pca9685->setFreq(_targetFreq) != PX4_OK) {
				PX4_ERR("failed to set pwm frequency to %.2f, fall back to 50Hz",
				↪  (double)_targetFreq);
				pca9685->setFreq(50.0f);	// this should not fail
			}
			_targetFreq = -1.0f;
		} else {
			// should not happen
			PX4_ERR("INIT failed: invalid initial frequency settings");
		}
		pca9685->startOscillator();
		_state = STATE::WAIT_FOR_OSC;
		ScheduleDelayed(500);
		break;

	case STATE::WAIT_FOR_OSC: {
			pca9685->triggerRestart();  // start actual outputting
			_state = STATE::RUNNING;
			float schedule_rate = pca9685->getFrequency();

			if (_schd_rate_limit < pca9685->getFrequency()) {
				schedule_rate = _schd_rate_limit;
			}
			ScheduleOnInterval(1000000 / schedule_rate, 1000000 / schedule_rate);
		}
		break;

	case STATE::RUNNING:
		_mixing_output.update();
		// check for parameter updates
		if (_parameter_update_sub.updated()) {
			// clear update
			parameter_update_s pupdate;
			_parameter_update_sub.copy(&pupdate);

			// update parameters from storage
			updateParams();
		}
		_mixing_output.updateSubscriptions(false);
		if (_targetFreq > 0.0f) { // check if frequency should be changed
			ScheduleClear();
			pca9685->disableAllOutput();
			pca9685->stopOscillator();
			if (pca9685->setFreq(_targetFreq) != PX4_OK) {
				PX4_ERR("failed to set pwm frequency, fall back to 50Hz");
				pca9685->setFreq(50.0f);	// this should not fail
			}
			_targetFreq = -1.0f;
			pca9685->startOscillator();
			_state = STATE::WAIT_FOR_OSC;
			ScheduleDelayed(500);
		}
		break;
	}
	perf_end(_cycle_perf);
}
```

Listing 45. PCA 9685 driver

PCA9685Wrapper::updateOutputs(...) (Listing 46). *In this method, differetly from SITL situation, the updateOutputs() calls the method updatePWM() which converts the PWM values in a 12 bit resolution for the driver.*

```
bool PCA9685Wrapper::updateOutputs(bool stop_motors, uint16_t *outputs, unsigned num_outputs,
                                   unsigned num_control_groups_updated)
{
        return pca9685->updatePWM(outputs, num_outputs) == 0 ? true : false;
}
```

Listing 46. PCA updateOutputs

PCA9685::updatePWM(...) (Listing 47).

```
int PCA9685::updatePWM(const uint16_t *outputs, unsigned num_outputs)
{
        if (num_outputs > PCA9685_PWM_CHANNEL_COUNT) {
                num_outputs = PCA9685_PWM_CHANNEL_COUNT;
                PX4_DEBUG("PCA9685 can only drive up to 16 channels");
        }

        uint16_t out[PCA9685_PWM_CHANNEL_COUNT];
        memcpy(out, outputs, sizeof(uint16_t) * num_outputs);

        for (unsigned i = 0; i < num_outputs; ++i) {
                out[i] = (uint16_t)roundl((out[i] * _Freq * PCA9685_PWM_RES / (float)1e6)); //
                ↪  convert us to 12 bit resolution
        }

        setPWM(num_outputs, out);

        return 0;
}
```

Listing 47. PCA updatePWM method

PCA9685::setPWM(...) (Listing 48).

```
void PCA9685::setPWM(uint8_t channel_count, const uint16_t *value)
{
        uint8_t buf[PCA9685_PWM_CHANNEL_COUNT * PCA9685_REG_LED_INCREMENT + 1] = {};
        buf[0] = PCA9685_REG_LED0;

        for (int i = 0; i < channel_count; ++i) {
                if (value[i] >= 4096) {
                        PX4_DEBUG("invalid pwm value");
                        return;
                }

                buf[1 + i * PCA9685_REG_LED_INCREMENT] = 0x00;
                buf[2 + i * PCA9685_REG_LED_INCREMENT] = 0x00;
                buf[3 + i * PCA9685_REG_LED_INCREMENT] = (uint8_t)(value[i] & (uint8_t)0xFF);
                buf[4 + i * PCA9685_REG_LED_INCREMENT] = value[i] != 0 ? ((uint8_t)(value[i] >>
                ↪  (uint8_t)8)) :
                                PCA9685_LED_ON_FULL_ON_OFF_MASK;
        }

        int ret = transfer(buf, channel_count * PCA9685_REG_LED_INCREMENT + 1, nullptr, 0);

        if (OK != ret) {
                PX4_DEBUG("setPWM: i2c::transfer returned %d", ret);
        }
}
```

Listing 48. PCA setPWM method

Chapter 5

Model Linearization and Sliding Mode Control Design

The control strategy we want to use consists of a linearization of the quadcopter model and a fourth order $HOSM$ (High Order Sliding Mode) control applied to the linearized model.
Most of the following informations are a brief summary of a previous work based on a detailed study of the mentioned control strategy [55].

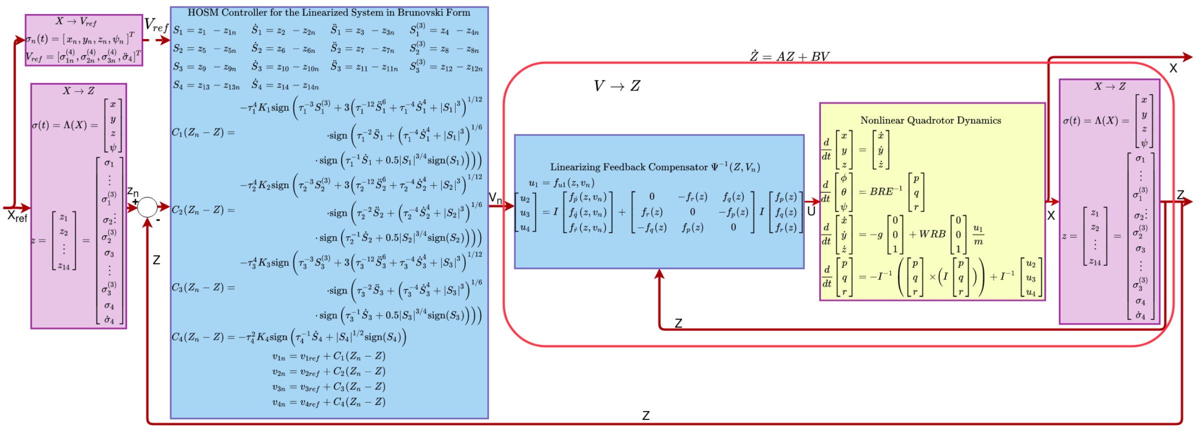


Figure 5.1. HOSM with Feedback Linearization

Source: Pekkaptan thesis [55]

5.1 Quadcopter dynamic model

The first thing done in order to design the control strategy was to model the quadrotor. Defining a fixed world frame and a mobile body one (see Figure 5.17), the position of the quadrotor was considered as the origin of the mobile frame, and its orientation, in Euler angles, as the rotation from fixed to mobile frame (5.1):

$$r = [x\ y\ z] \quad \Phi = [\phi\ \theta\ \psi] \tag{5.1}$$

Then under some simple assumptions, according to [55], the equation of motion of quadcopter can be written as:

$$\begin{aligned} & m\ddot{\mathbf{r}} = -mg\mathbf{z}_W +^W \mathbf{R}_B \mathbf{u}_1 \mathbf{z}_B \\ & \mathbf{I}\dot{\omega} + \omega \times I = \mathbf{u}_{234} \end{aligned} \tag{5.2}$$

where m and I are the mass and the inertia tensor, and g the gravitational constant. Then, z_B and z_W represent the Z axes in body and world frame, ${}^W\mathbf{R}_B$ represents the rotation matrix from fixed to mobile frame, and $\omega = [p\ q\ r]^T$ the rotational velocity vector in body frame, related to the derivative of the Euler angles $\mathbf{\Phi}$.
The inputs of the quadrotor are defined as $\mathbf{u_1}$ and $\mathbf{u_{234}}$, such that:

$$u_1 = \sum_{i=1}^{4} F_i \quad \mathbf{u}_{234} \triangleq \begin{bmatrix} L\,(F_2 - F_4) \\ L\,(F_3 - F_1) \\ M_1 + M_3 - M_2 - M_4 \end{bmatrix} \tag{5.3}$$

where L is the length of each body arm, F_i an M_i, $i = 1, ..., 4$, are the thrust and reaction moments generated by propellers.

5.2 Linearization

The linearization procedure exploits the differential flatness property of non linear systems, considering the quadrotor dynamic with differential equations and using a feedback or feedforward linearization method that transform the nonlinear model to a Brunovski canonical (normal) form (5.8).
Since the input vector is four-dimensional (thrust and roll, pitch, yaw moments) the flat position elements are:

$$\sigma = [\sigma_1\ \sigma_2\ \sigma_3\ \sigma_4] = [x\ y\ z\ \psi] \tag{5.4}$$

Then considering the 0-flat system in Brunovski form, the flat state (considering all the derivarives) vector becomes:

$$\begin{aligned} & z = [z_1\ z_2\ ...\ z_{14}] = [\sigma_1\ ...\ \sigma_1^{(3)}\ \sigma_2\ ...\ \sigma_2^{(3)}\ \sigma_3\ ...\ \sigma_3^{(3)}\ \sigma_4\ \dot{\sigma}_4] \\ & v_n = [v_1\ v_2\ v_3\ v_4] = [\sigma_1^{(4)}\ \sigma_2^{(4)}\ \sigma_3^{(4)}\ \ddot{\sigma}_4] \end{aligned} \tag{5.5}$$

The control values (5.6), (5.7) are computed on the flat states and then applied to the system that will behave as a series of integrators (5.8),

$$\begin{aligned} u_1 &= m\sqrt{\ddot{\sigma}_1^2 + \ddot{\sigma}_2^2 + (\ddot{\sigma}_3 + g)^2}\,\Big) \\ u_{234} &= I \begin{bmatrix} \dot{p} \\ \dot{q} \\ \dot{r} \end{bmatrix} + \begin{bmatrix} 0 & -r & q \\ r & 0 & -p \\ -q & p & 0 \end{bmatrix} \begin{bmatrix} p \\ q \\ r \end{bmatrix} \end{aligned} \tag{5.6}$$

$$\begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{bmatrix} = \begin{bmatrix} f_{u1}(\sigma, \dot{\sigma}, \dot{\sigma}, \sigma^{(3)}, \sigma^{(4)}) \\ f_{u2}(\sigma, \dot{\sigma}, \dot{\sigma}, \sigma^{(3)}, \sigma^{(4)}) \\ f_{u3}(\sigma, \dot{\sigma}, \dot{\sigma}, \sigma^{(3)}, \sigma^{(4)}) \\ f_{u4}(\sigma, \dot{\sigma}, \dot{\sigma}, \sigma^{(3)}, \sigma^{(4)}) \end{bmatrix} \tag{5.7}$$

The overall system will be captured by:

$$\dot{z} = Az + Bv \tag{5.8}$$

$$A = \begin{bmatrix} 0&1&0&0&0&0&0&0&0&0&0&0&0&0 \\ 0&0&1&0&0&0&0&0&0&0&0&0&0&0 \\ 0&0&0&1&0&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&1&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&1&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&1&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&1&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&0&1&0&0&0 \\ 0&0&0&0&0&0&0&0&0&0&0&1&0&0 \\ 0&0&0&0&0&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&0&0&0&0&1 \\ 0&0&0&0&0&0&0&0&0&0&0&0&0&0 \end{bmatrix} \quad B = \begin{bmatrix} 0&0&0&0 \\ 0&0&0&0 \\ 0&0&0&0 \\ 1&0&0&0 \\ 0&0&0&0 \\ 0&0&0&0 \\ 0&0&0&0 \\ 0&1&0&0 \\ 0&0&0&0 \\ 0&0&0&0 \\ 0&0&0&0 \\ 0&0&1&0 \\ 0&0&0&0 \\ 0&0&0&1 \end{bmatrix} \tag{5.9}$$

In the following, two control laws to transform the quadrotor model in Brunovsky canonical form are introduced, one based on a feedback and another one on a feedforward control action.

5.2.1 Feedback Linearization

The Feedback (FB) linearization method required all the states to be known. This approach implies that its source of problems are the measurements and estimates noise.
A general quadrotor has on-board GPS sensor, accelerometer and gyroscope, thus its frame position, speeds, and accelerations, both linear and rotational, can be directly measured. The only estimate left out is the jerk, which is the first derivative of the known acceleration terms.
A linearizing law can be derived from the input mapping substituting the highest order derivatives of flat outputs with the corresponding elements of virtual input vector $\mathbf{v_n}$, since looking at equation (5.6), $\mathbf{u_1}$ does not depend on virtual inputs, we had to derive it twice:

$$\ddot{u}_1 = \frac{m}{\sqrt{\alpha^\top \alpha}} \left[\alpha^\top \gamma + \beta^\top \beta - \frac{\left(\alpha^\top \beta\right)^2}{\alpha^\top \alpha} \right] \tag{5.10}$$

with $\alpha = \begin{bmatrix} \ddot{\sigma}1 & \ddot{\sigma}2 & \ddot{\sigma}_3 + g \end{bmatrix}^\top$, $\boldsymbol{\beta} = \begin{bmatrix} \sigma_1^{(3)} & \sigma_2^{(3)} & \sigma_3^{(3)} \end{bmatrix}^\top$, $\gamma = \begin{bmatrix} \sigma_1^{(4)} & \sigma_2^{(4)} & \sigma_3^{(4)} \end{bmatrix}^\top$.

The control values, written as function of the state $\mathbf{z}$, and the feedforward terms $\mathbf{v_n}$ (see equation (5.5)), will become the double integral of equation (5.10) for $\mathbf{u_1}$, while $\mathbf{u_{234}}$ will be the same as shown in equation (5.6).
Summarizing the control action will be:

$$u = \psi(z, v_n) \tag{5.11}$$

In order to compute the feedback linearization control input of equation (5.11), the moment inertia matrix I and the drone mass have to be known as well; mass it is not a problem at all, but the inertia could be another source of errors. In order to avoid undesired behaviour due to inaccurate calculation, robustifying controller should be implemented or a feedforward linearization (see Section 5.2.2), which is not function of any measurements.

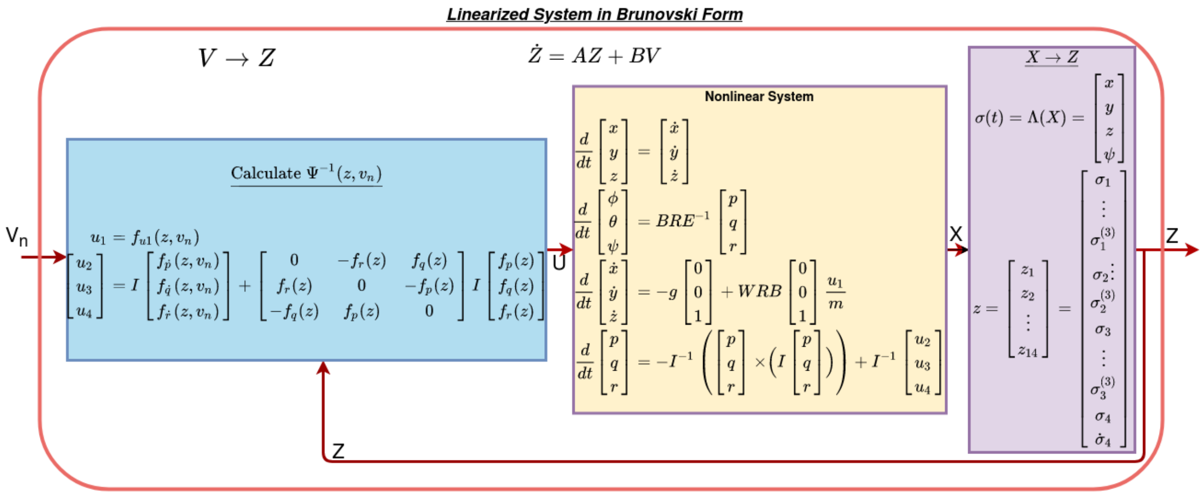


Figure 5.2. Feedback Linearization

Source: Pekkaptan thesis [55]

5.2.2 Feedforward Linearization

The Feedforward (FF) linearization method simply uses terms given by the setpoint trajectory. Differently from FB linearization, this method has no clue of the real state position with the advantage that there is no noise that could affect this technique.
The linearization functions remain the same of the one showed for FB method, the only difference is that all the states terms will be replaced by nominal flat output vector z_n. The control values become function of the trajectory feedforward terms z_n and v_n, i.e,

$$u = \psi(z_n, v_n) \tag{5.12}$$

The vector term z_n is the computed from the trajectory, considering it as a time function $f(t)$ of x, y, z, ψ, the z_n terms are all the partial derivative terms, as was done for measurements "z" in equation (5.5).

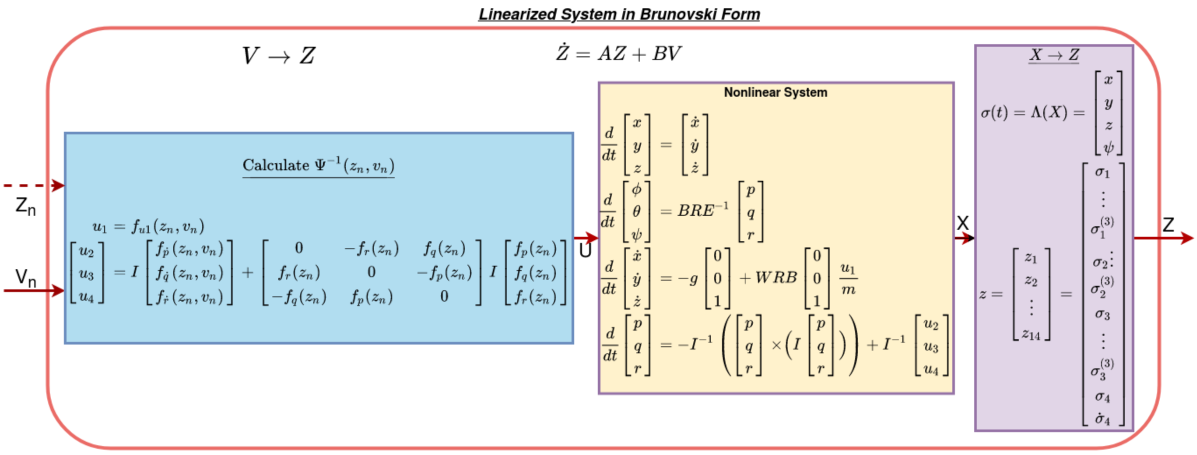


Figure 5.3. Feedforward Linearization

Source: Pekkaptan thesis [55]

5.3 HOSM Control

The HOSM (High Order Sliding Mode) is a nonlinear control strategy member of the SMC and VSC control techniques. In next sections the HOSM technique used in this work [55] is described.

Sliding Mode Control (SMC)

In order to better understand the HOSM control approach, some basic elements of SMC theory are described hereafter. The applied control values behave in a switching/discontinuous way, trying to bring to zero the sliding surface which is given by the difference between the setpoint and the state.
The dynamic of the system restricted to the sliding surface should be easy to control so as to converge to some suitable equilibrium.
In order to show the main characteristic of SMC theory, let's consider a simple example of an unstable linear system (5.13).

$$\text{S:} \begin{cases} \dot{x}1 = x2 \\ \dot{x}2 = -x1 + 2x_2 + u \\ y = x_1 \end{cases} \tag{5.13}$$

The control action $\psi(x)$ it's a non linear switching upon a sliding surface $s(x)$, equation (5.14), (5.15),

$$\text{C: } u = -\psi(x)y \tag{5.14}$$

$$\psi(x) = \begin{cases} -1, & s(x) < 0 \\ +1, & s(x) > 0 \end{cases}, \quad s(x) = x_1 \tag{5.15}$$

It can be easily seen that substituting the two control actions on system (5.13) the two resulting subsystems remain both unstables.

The peculiarity of the sliding surface and the discontinuous control action, is to keep the whole system on the edge between the two subsystems, resulting in a stable one as can be seen in Figure 5.4.

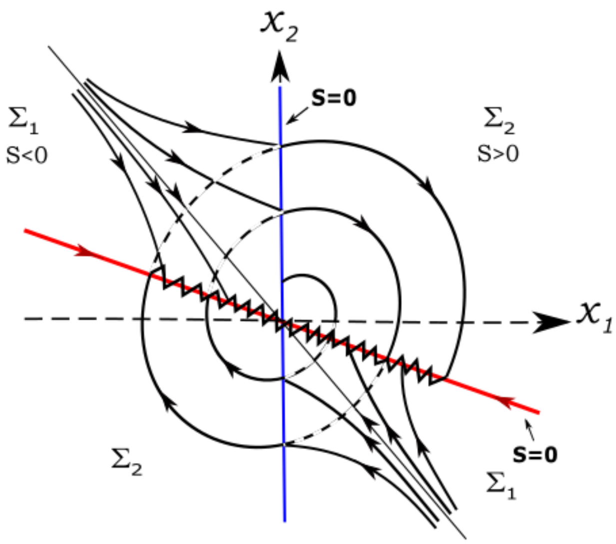
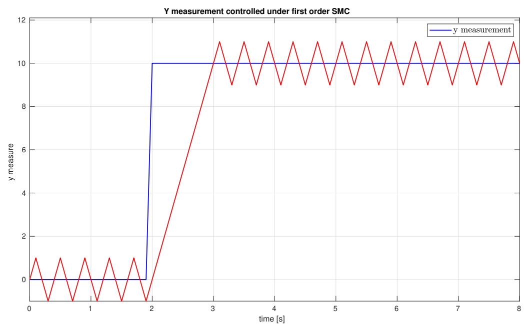


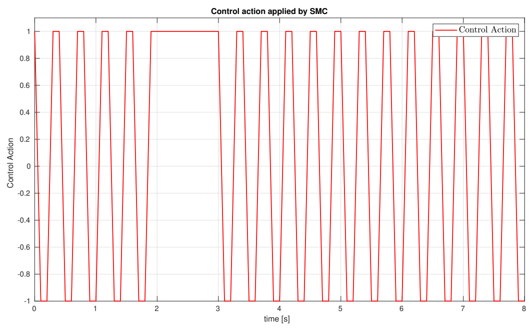
Figure 5.4. Sliding Mode general behaviour

Source: Teaching slides [56]

The control problem stated above has been implemented in Simulink/Matlab, and the stabilization results can be seen in Figure 5.5, showing both control action and y measurement.
It can be noticed that the convergence time is finite, and that the measure continues to oscillate along the reference point (that is a chattering phenonemon occurs).



(a) Y measure



(b) Control action

Figure 5.5. SMC example results

As can be seen in Figure 5.14, the main problem of classical SMC is the presence of oscillations (chattering) due to a strong discontinuity in the control values.
The HOSM control approach partially solves this issue considering higher derivatives

of the sliding surface transferring the discontinuity into an arbitrary higher order, making the control signal applied to the plant results to be a continuous one.
When a HOSM control law of order r is applied to a system, sliding variables and their derivatives up to order $r-1$ are put into sliding mode. Resulting in a so called r-sliding mode.

5.4 HOSM control for Quadrotor

Our HOSM controller [55] has to stabilize the linearized plant mentioned before that is equivalent to the Brunovski form (5.8).
The linearized system can be separated into four subsystem each one with a different relative degree (fourth order for x, y, z and second order for ψ). The sliding variables will consist in the error between the measurement and the setpoint (5.16), and the errors between the corresponding derivatives (5.17).

$$\begin{aligned}
S_1 &= \sigma_1 - \sigma_{1n} = z_1 - z_{1n} \\
S_2 &= \sigma_2 - \sigma_{2n} = z_5 - z_{5n} \\
S_3 &= \sigma_3 - \sigma_{3n} = z_9 - z_{9n} \\
S_4 &= \sigma_4 - \sigma_{4n} = z_{13} - z_{13n}
\end{aligned} \tag{5.16}$$

$$\begin{aligned}
\dot{S}_1 &= \dot{\sigma}_1 - \dot{\sigma}_{1n} = z_2 - z_{2n} \\
\dot{S}_2 &= \dot{\sigma}_2 - \dot{\sigma}_{2n} = z_6 - z_{6n} \\
\dot{S}_3 &= \dot{\sigma}_3 - \dot{\sigma}_{3n} = z_{10} - z_{10n} \\
\ddot{S}_1 &= \ddot{\sigma}_1 - \ddot{\sigma}_{1n} = z_3 - z_{3n} \\
\ddot{S}_2 &= \ddot{\sigma}_2 - \dot{\sigma}_{2n} = z_7 - z_{7n} \\
\ddot{S}_3 &= \ddot{\sigma}_3 - \ddot{\sigma}_{3n} = z_{11} - z_{11n} \\
S_1^{(3)} &= \sigma_1^{(3)} - \sigma_{1n}^{(3)} = z_4 - z_{4n} \\
S_2^{(3)} &= \sigma_2^{(3)} - \sigma_{2n}^{(3)} = z_8 - z_{8n} \\
S_3^{(3)} &= \sigma_3^{(3)} - \sigma_{3n}^{(3)} = z_{12} - z_{12n} \\
\dot{S}_4 &= \dot{\sigma}_4 - \dot{\sigma}_{4n} = z_{14} - z_{14n}
\end{aligned} \tag{5.17}$$

Finally the control law is computed as function of the higher order sliding surface as:

$$\begin{aligned}
C_{xyz}^{(4)}(z_n - z) &= f(S^{(3)}, \ddot{S}, \dot{S}, S) \\
C_{\psi}^{(2)}(z_n - z) &= f(\dot{S}, S)(\psi)
\end{aligned} \tag{5.18}$$

The final reference "v_n" given to the linearization block is given instead by:

$$\begin{aligned}
v_{n1} &= \sigma_{1n}^{(4)} + C_1^{(4)}(z_n - z) \\
v_{n2} &= \sigma_{2n}^{(4)} + C_2^{(4)}(z_n - z) \\
v_{n3} &= \sigma_{3n}^{(4)} + C_3^{(4)}(z_n - z) \\
v_{n4} &= \sigma\ddot{_{4n}} + C_4^{(2)}(z_n - z)
\end{aligned} \tag{5.19}$$

5.4.1 HOSM Control Laws

This section shows in more details the HOSM control law mentioned before (see equation (5.18), and (5.19)). For quadcopter four HOSM controllers can be applied, one for each state. Before deciding the HOSM control family to use we have to evaluate the relative degrees of the four separate systems (5.20), that are:

$$r = \begin{bmatrix} r_x \\ r_y \\ r_z \\ r_\psi \end{bmatrix} = \begin{bmatrix} 4 \\ 4 \\ 4 \\ 2 \end{bmatrix} \tag{5.20}$$

This means that a 4-th order, and a 2'nd order sliding mode have to be used (equation (5.21)),

$$\begin{aligned} C^{(4)}_{xyz}(z_n - z) =& - \tau_i^4 K_i \operatorname{sign}\left(\tau_i^{-3} S_i^{(3)} + 3\left(\tau_1^{-12} \ddot{S}_i^6 + \tau i^{-4} \ddot{S}_i^4 + |S_i|^3\right)^{1/12}\right. \\ & \cdot \operatorname{sign}\left(\tau_i^{-2} \dot{S}_i + \left(\tau_i^{-4} \dot{S}_i^4 + \mid S_i^3\right)^{1/6}\right. \\ & \left.\left.\operatorname{sign}\left(\tau_i^{-1} \dot{S}_i + 0.5\,|S_i|^{3/4} \cdot \operatorname{sign}(S_i)\right)\right)\right) \\ C^{(2)}_{\psi}(z_n - z) =& - \tau_4^2 K_4 \operatorname{sign}\left(\tau_4^{-1} \dot{S}1 + |S4|^{1/2} \cdot \operatorname{sign}(S_4)\right) \end{aligned} \tag{5.21}$$

As it will be seen in Section 5.6.7, due to noise measurements and numerical errors, the sign function brought some problems, the solution was to modify it with a "saturated sign" function (see Section 5.18), this improved the chattering problem as will be shown in Figure 5.19.

5.5 Code Implementation

The control technique was first tested in *Simulink* and *Matlab* environment as can be seen in [55].
The simulation results achieved were quite good, but the used environment is not very realistic, no noise on the measurements were used, indeed no external disturbance was considered and no estimation of variables was considered (all states are assumed to be known); the purpuse of this work was to test this control strategy in a simulation environment that simulates all the non-linearities and problems present on a real drone.

5.5.1 Comparison cpp Controller/Matlab

The first part was dedicated to implement the controller in cpp language, the one used by PX4 firmware (see Chapter 3).
All the tests have been done using an helical reference trajectory and comparing the results between the controller in the two forms.

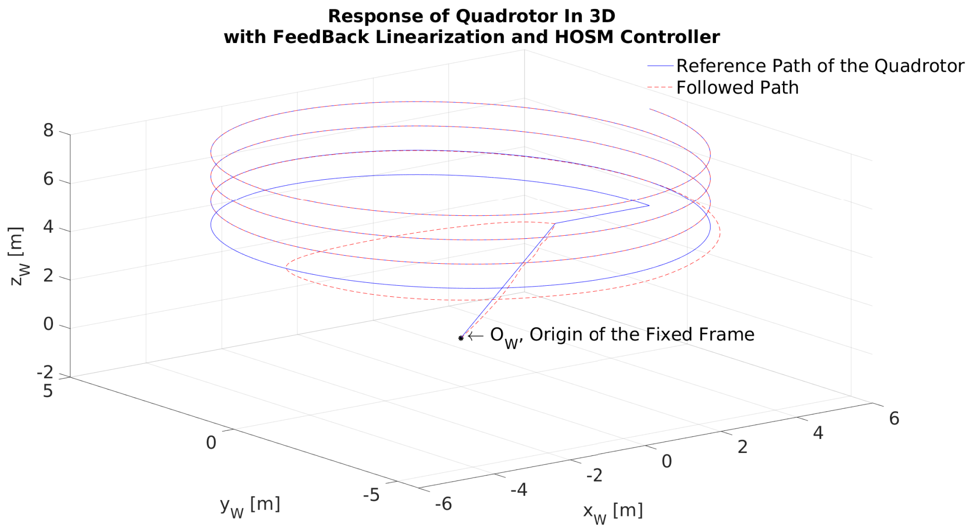


Figure 5.6. Helix Trajectory

All the tests showed are made with the Feedback linearization tecnique, but the same was done with the FeedForward strategy, with similar results.

Test Control Replica

In the first test, the simulation is firstly run in Simulink, then the reference and measurements time series are saved and read by the cpp controller that computes the corresponding control action which is then compared with the one given by the Simulink controller block (see Figure 5.7).

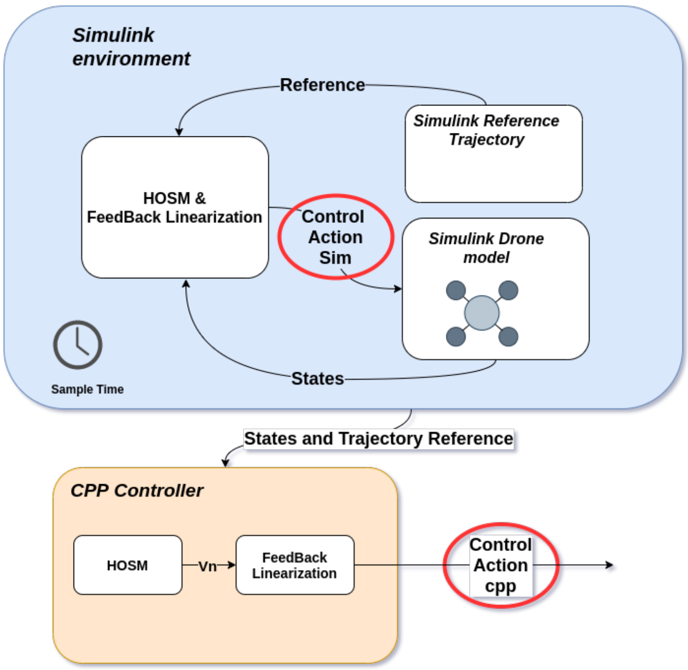
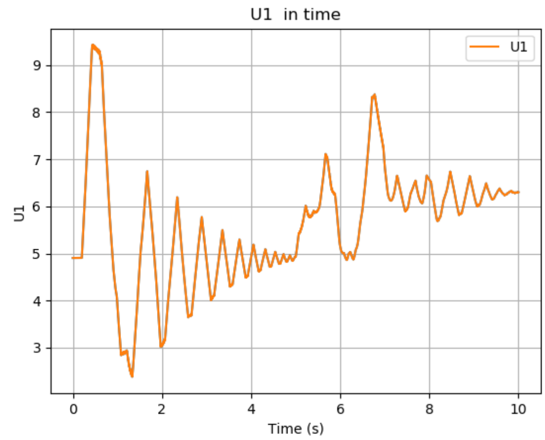
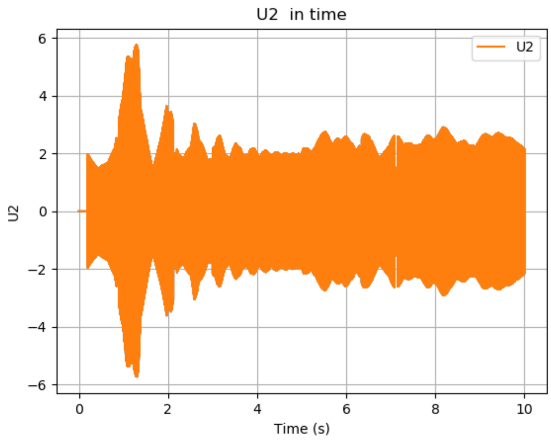


Figure 5.7. Test Open Loop Scheme

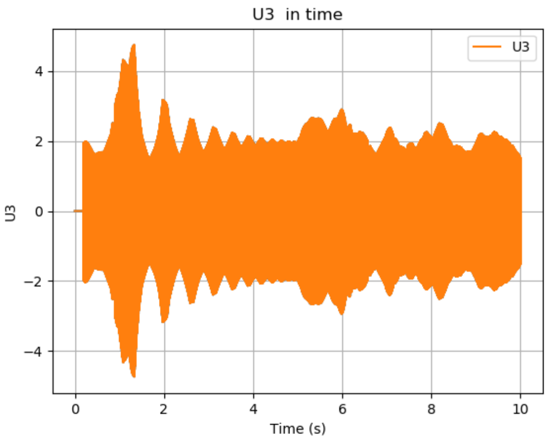
The achieved results were quite satisfactory, as can be seen in Figures 5.8 (control action) and 5.9 (control action error), the replica is almost perfect, and the error is close to zero.



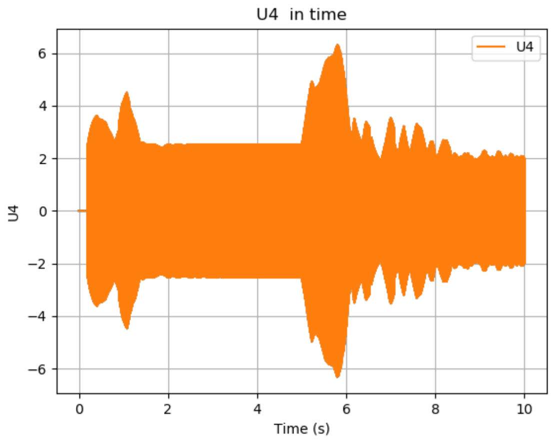
(a) U1



(b) U2

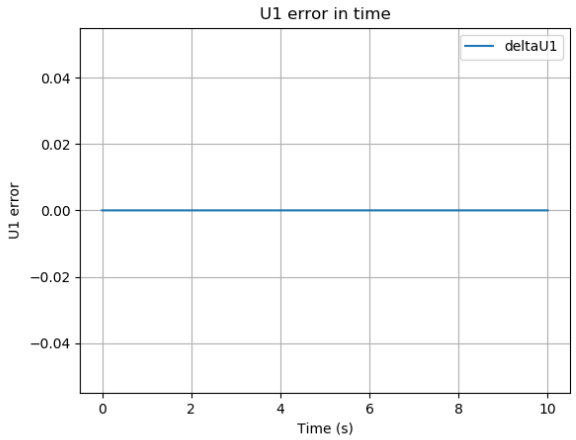


(c) U3

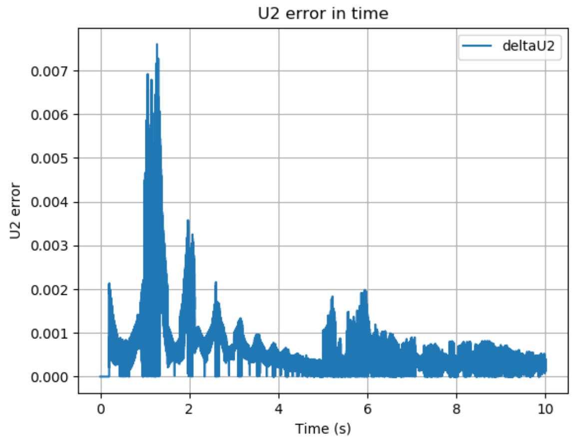


(d) U4

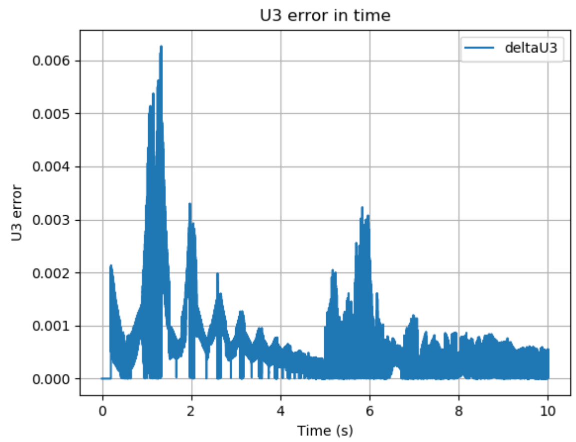
Figure 5.8. Control Action Replica



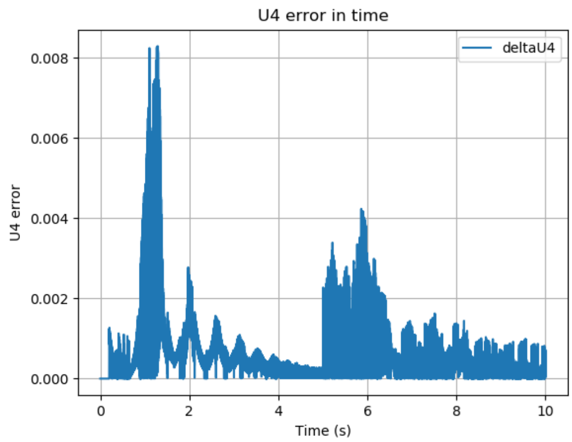
(a) U1 error



(b) U2 error



(c) U3 error



(d) U4 error

Figure 5.9. Control Action Error

Test Controller Closed Loop

In the second test, the control action computed by the controller written in cpp is given to the simulink quadcopter model, the simulation is run step by step for the whole trajectory, reading the references and the states from simulink and computing the control action with the cpp code at each step (see Figure 5.10).

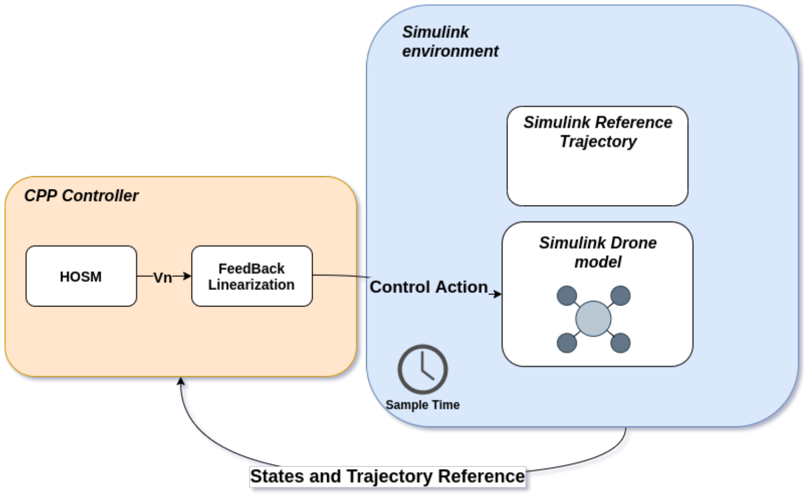
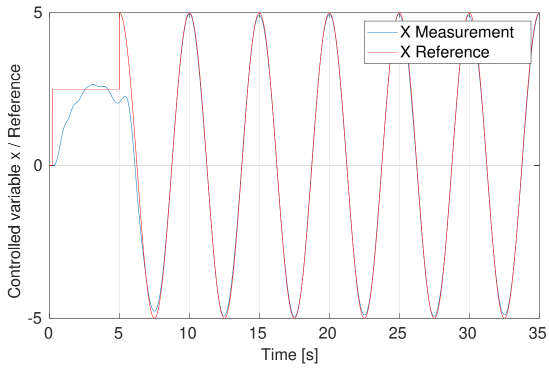
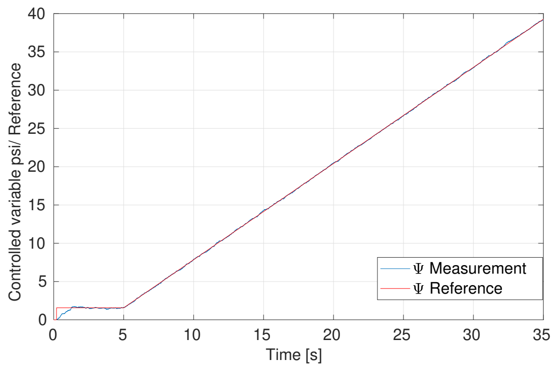


Figure 5.10. Test Closed Loop Scheme

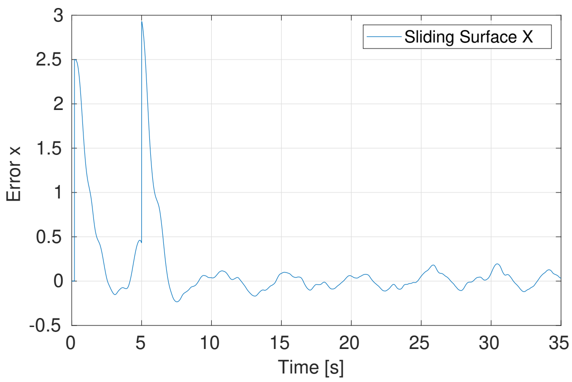
The achieved results were quite satisfactory, the controller is able to stabilize the quadcopter and to follow the given trajectory. This can be seen in Figure 5.11, showing X reference and measurement, and the error between the two on the left, then the same for Ψ variable (representing the heading of the drone) on the right.



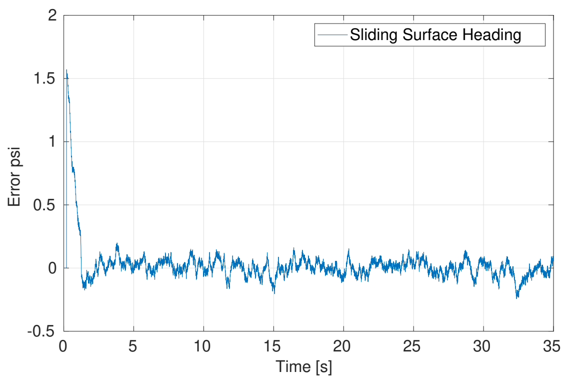
(a) X tracking



(b) Heading tracking



(c) X error



(d) Heading error

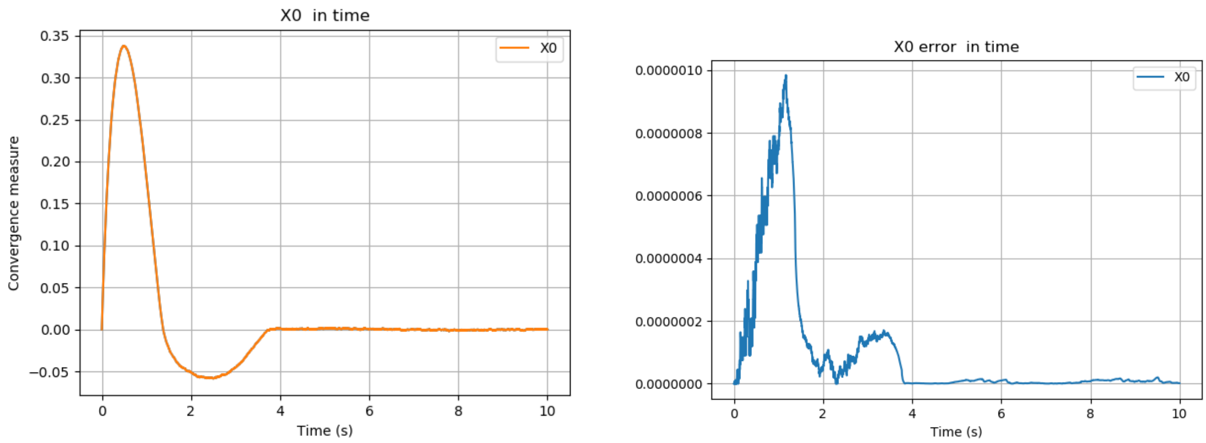
Figure 5.11. Closed Loop Testing Results

Differentiator

Before implementing the control strategy to the PX4 architecture it was needed to introduce and test an "estimator", in order to do it we used the Differentiator [57]. It was needed to estimate the jerks values along x, y and z directions since the highest measurement derivative available from PX4 was the acceleration.

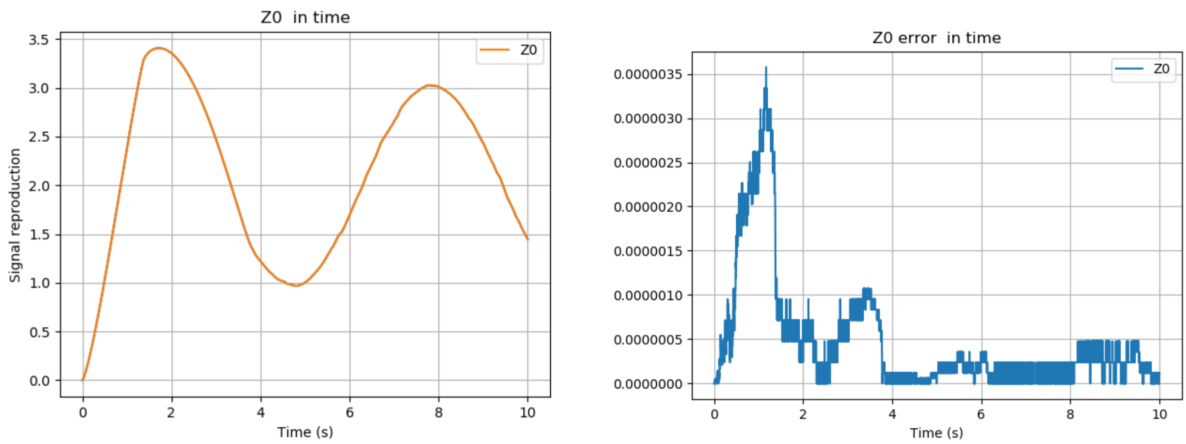
Differentiator Replica

It was used the same setup used for "Test Control Replica" (Figure 5.9), a simulation model that runs the simulink differentiator block and on the other hand its cpp implementation. The results are showed in Figure 5.12, 5.13 and 5.14, where X_0 represents a convergence indicator, Z_0 the signal to be derived, and Z_1 its derivative.



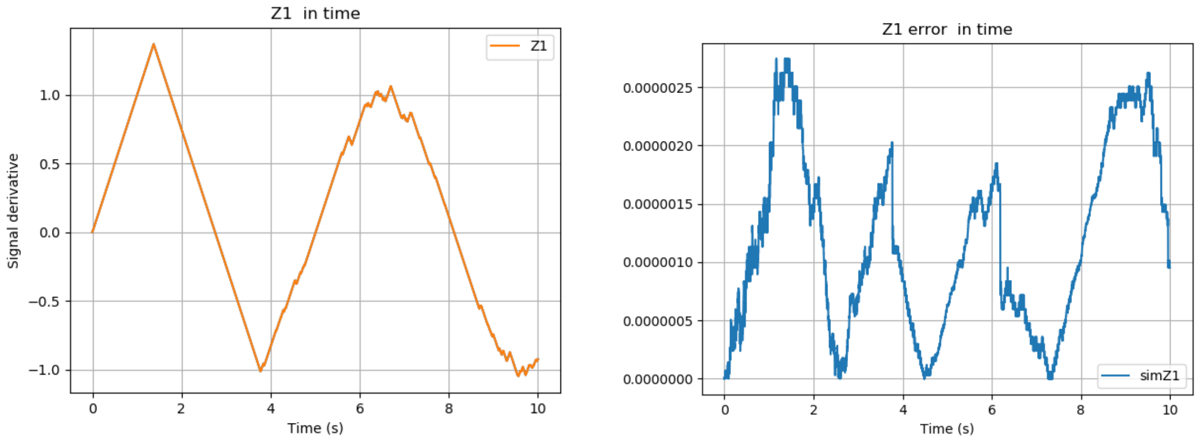
(a) X_0 (b) X_0 error

Figure 5.12. Differentiator Replica Testing Results X



(a) Z_0 (b) Z_0 error

Figure 5.13. Differentiator Replica Testing Results Z_0



(a) Z_1 (b) Z_1 error

Figure 5.14. Differentiator Replica Testing Results Z_1

Closed Loop with Differentiator

Finally, the same tests done in “Test Controller Closed Loop” were done using the Differentiator which estimated the jerk values from acceleration measurements (Figure 5.15), and quite the same results were achieved.

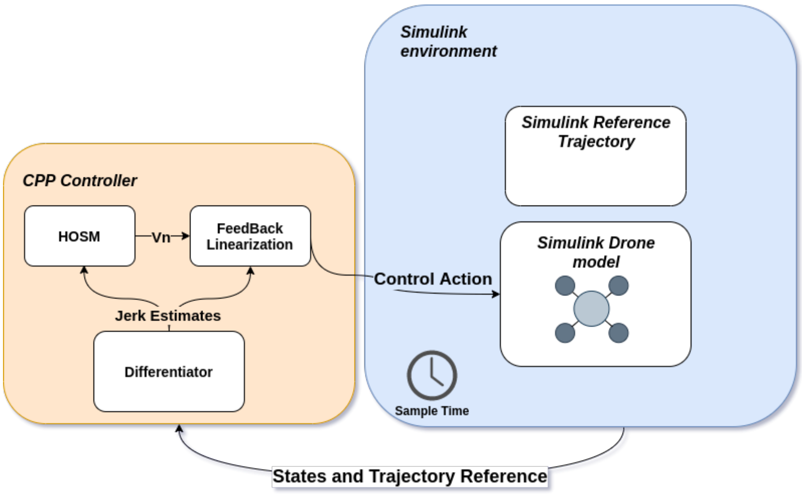


Figure 5.15. Test Closed Loop Scheme with Differentiator

5.5.2 Final Controller Architecture

The final control architecture consists of three main classes:

- **Differentiator :** used to estimate the acceleration derivatives (jerk).
- **HOSM :** the control algorithm.
- **Linearization :** it handles the linearization of the quadcopter model and sets the control values as Thrust, Roll, Pich and Yaw.

- **Math Libraries :** some libraries for matrix arithmetic are used, and a customized sign function.

5.6 Firmware Implementation

In this section the interface problems between the controller and the PX4 architecture are shown. The main idea (see Figure 5.16) was to keep the PX4 architecture mostly untouched and to create a new module, for our controller, that runs in parallel with the internal PX4's PID structure (see Chapter 4).

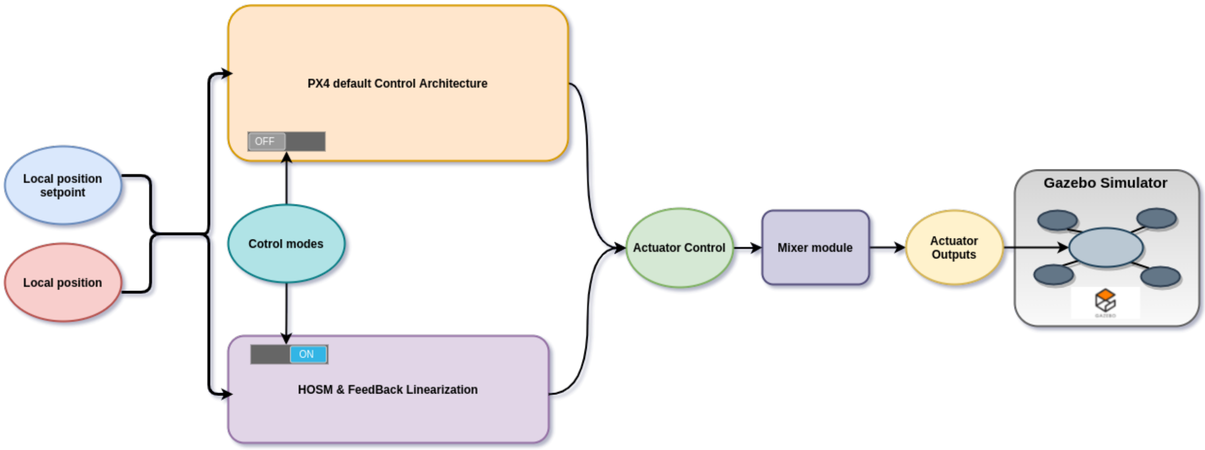


Figure 5.16. Firmware Implementation Scheme

A customized uORB topic (see Section 3.4) was also created in order to know when the HOSM has to be used or not, its parameters have been set from time to time inside the firmware, mostly set in the commander module (see Section 4.2). All the simulations and tests were made using the Gazebo environment (see Section 3.6.1).

5.6.1 Reference Frames

PX4 and the considered frames for linearization were different (Figure 5.17), thus we had to adapt our control strategy to PX4 frames.

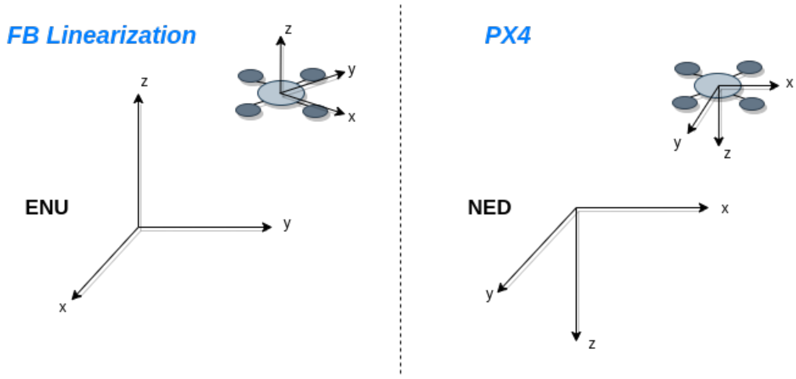


Figure 5.17. Different Reference Frames

The used Rotational matrix were:
For the fixed reference frame (used to adapt local measurements and trajectory reference point):

$$R_{NEDtoENU} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix} \tag{5.22}$$

For the local reference frame (used to adapt the control torques to be applied):

$$R_{FEEDtoPX4} = \begin{bmatrix} 0.707 & 0.707 & 0 \\ 0.707 & -0.707 & 0 \\ 0 & 0 & -1 \end{bmatrix} \tag{5.23}$$

In order to interface our controller we did a reorder of thrust and moments forces to the order used by PX4 mixer (see Section 4.8).

5.6.2 Scale Outputs

As was shown in Chapter 4 (see Section 4.8.2) the control action in PX4 control architecture is scaled between 0 and 1, and -1 and 1 indicating respectively the min the the max that the quadcopter can gives in terms of Thrust, Roll, Pitch and Yaw.
In order to adapt our control technique to this architecture, and to avoid to change the mixing phase, first the maximum rotor power was computed looking the Gazebo vehicle model used; then studying the quadcopter geometry the maximum allowed forces were computed.
This values were used to saturate and rescale the control action given from our controller.
Finally (similarly to the PX4 technique (Listing 15)), the z direction was prioritized limiting the value that can be given to the moments in order to prevent mixing saturation.

5.6.3 Measurements Update

The main problem was that the used simulation environment (Gazebo (Figure 3.7)) also simulates the sensors, and some measurements did not arrive at the same controller frequency, but at a lower one (as often happens in real applications). This constrained the use of a ZOH (Zero Order Hold) to keep constant the measurement until a new one arrive. This could bring some delay in the control action but the frequency was enough to keep the system stable.

5.6.4 Setpoint Trajectory

When using this type of non linear control strategy with fast variations and discontinuities, if the reference trajectory also have large jumps, the control action can give problem similar to the PID “derivative kick” [50]. A good practice is to set the reference trajectory as smooth as possible in order to avoid these type of discontinuities.
In our application the trajectory used for tests was added as a new “Flight Task” (see

Section 4.4), and we made sure to avoid large discontinuities with respect to previous position and during the whole path.

5.6.5 Switching Strategy

A problem to be solved was how to handle the switching phase, passing from the internal PX4 controller to our controller.
The solution was to keep the controllers in “tracking mode” [58]; in particular when the internal PID structure works our controller track the control action given by the other controller and reset any integral action, the same was done for the internal Rate Controller (see Section 4.7) when our controller was running.
The proposed solution allowed to have a switching phase smooth enough to keep the quadcopter stable.

5.6.6 Lockstep Simulation

Lockstep is a simulation tecnique needed to synchronize two programs that runs in parallels with different timestamps and clocks [39].
Hopefully this tecnique was already implemented in PX4 architecture, as can be seen in the online guide [40] and in Section (3.6.2), and it also include the possibility to customize it.
In our case we needed to raise the update frequency to at least 1Khz (2 Khz were used), this was done keeping the lockstep enabled only from the PX4 Firmware interface and disabling it from the simulator side (Gazebo). This allowed to run the controller with an appropriate frequency and make everything work properly.

5.6.7 Sign Saturation Technique

One peculiarity of the HOSM control implementation was to change the sign function of the sliding surfaces to a sign saturated function (Figure 5.18), this was needed to prevent problems caused from noise on measurements.

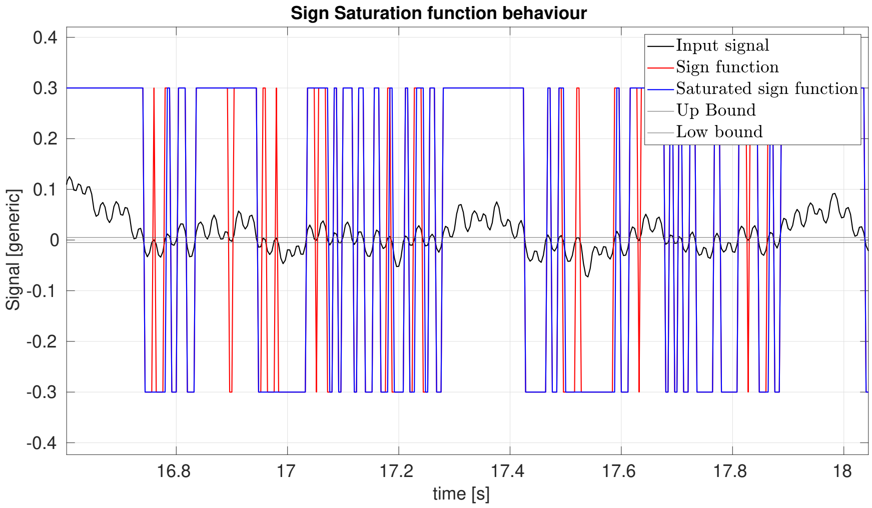
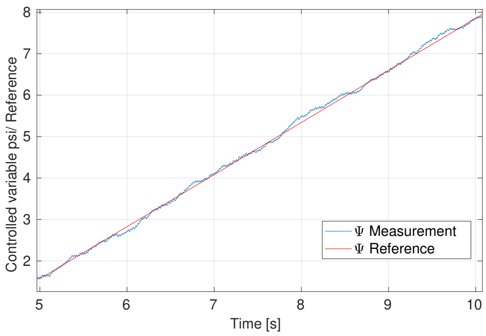
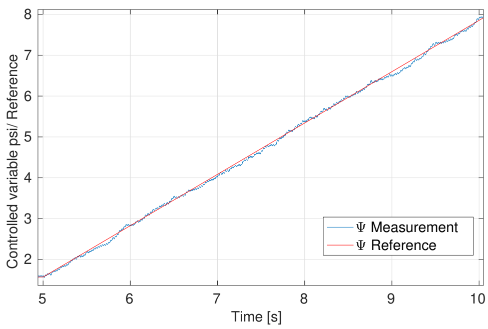


Figure 5.18. Saturation sign()

The idea is to avoid switching the control action when the error is very low, this prevent mislabed control action and reduce the chattering problem too.
An implementation result can be seen in Figure 5.19, where on the left, the signal with no saturation goes up and down with respect to the reference while in the graph on the right the measured signal has less discontinuities and less jumps due to saturation.



(a) No Saturation



(b) With Saturation

Figure 5.19. Sign Saturation

Chapter 6

Simulation Results

In this chapter, the testing phase made only on $SITL$ mode is illustrated and discussed.

6.1 Simulation Model

The simulation environment used was Gazebo.
This simulator allows to easily introduce new customized model in a modular way, in particular we were interested in:

- the sensors plugins able to introduce noise on the measurement and common problems that could be faced in real applications;
- motor plugin where not only noise can be introduced but also the actuator dynamic is simulated;
- realistic environment where for example external disturbances like wind can be introduced.

The second reason was the fact that Gazebo is the main simulator used by ROS, so it can be used for future works interfacing ROS and PX4 through MAVROS messaging protocol (see Section 3.5).

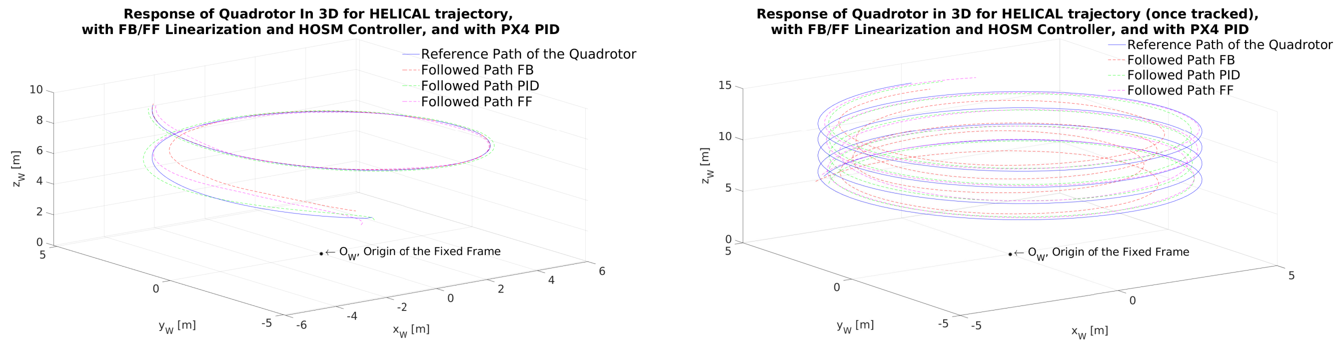
6.2 Reference Trajectory

The chosen trajectory is a smooth helical motion combined with a continous gyration around z_W, the helix was test with different gyration speed in order to see how the system performs with harder trajectories.
The helical path was added as a new flight task as described in Chapter 5 (see Section 5.6.4). The use of a customized trajectory was needed in order to include all the derivatives for reference x, y, z up to the fourth order and for ψ up to the second (needed for HOSM control and model linearization).

6.3 Trajectory Tracking Results

The tests used as reference a fast helix trajectory and a slow one.
The tested control strategies are:

- HOSM controller applied to Feedback linearized quadrotor.
- HOSM controller applied to Feedforward linearized quadrotor.
- PID inner architecture.



(a) Slow Helix

(b) Fast Helix

Figure 6.1. Fast and Slow helix trajectory

The data are taken from *QGC* where the uorb topics can be logged (see Section 3.4) and re-elaborated with Matlab.

6.3.1 Performance Index

In order to analyze the performances, ISE (Integral Square Error) was calculated. ISE is calculated by integrating square of the difference between the reference and measured trajectory for the whole time interval [59]. As second performance index $RMSE$ (Root Main Square Error) is computed too in order to have a direct perception of the error between the reference and the measure. Table (6.1) presents both the ISE and the RMSE values.

6.3.2 Comparison with the inner PID cascade architecture

The achieved results were satisfactory, the HOSM sliding mode performs pretty well both with Feedback and Feedforward linearization.
The Feedback seems to be more stable than the Feedforward one in the beginning (where the FF have some problems), while the FF performs better than the FB one once the setpoint is tracked.
Comparing both linearization methods with the PID controller can be said that apart from the initial transition phase, the non-linear approach have lower error with slow trajectory with both linearization tecniques, and the FF methods performs better even with faster trajectories.
At first look the PID seems to best follows the signal but looking at the Index parameters this is not true since the PID signal has a delay with respect to the reference as can be seen in Figure 6.4, and 6.8.
Note: all x axis in next plots start from time 5 seconds, it corresponds to the helix

trajectory start, and they ends when trajectory is finished, this is done since we are not interested in other behaviours, external to our control strategy.

Slow helix

Slow helix (Figure 6.2) is well followed by all the controllers, and apart from the first transient, the HOSM control, with FF and FB, perfectly follows the trajectory. The inner PID is not able to bring the error to zero. This can be seen both from Figure 6.5, where X and Y error continue to oscillate for PID controller (green line), and from Table 6.2 where can be seen that the AVG error for both thr HOSM strategies its almost one order less than PID one.

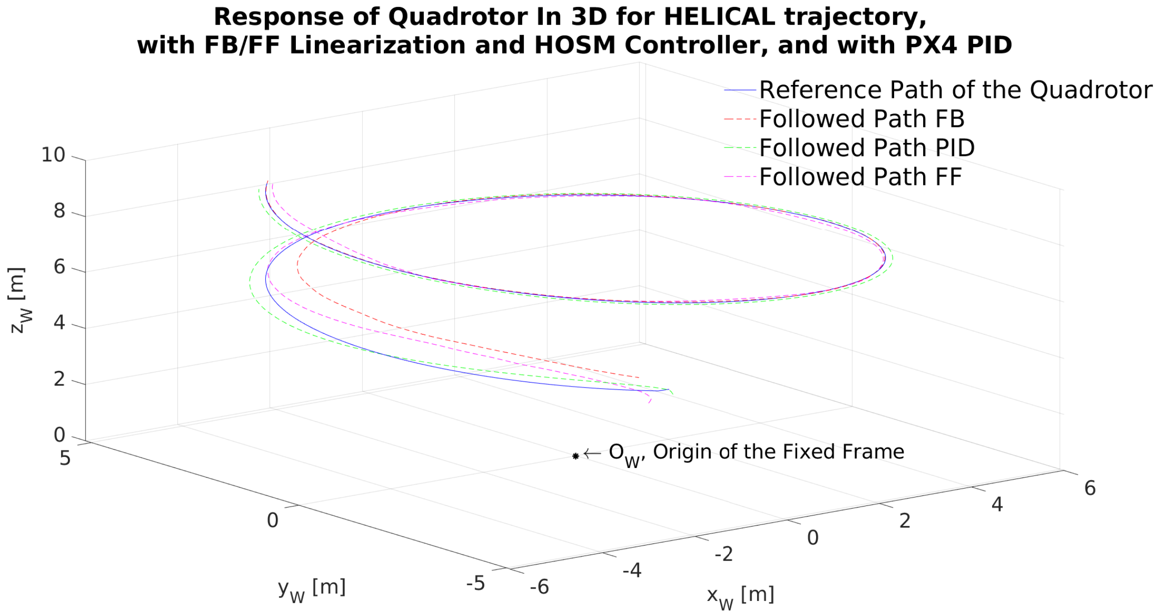
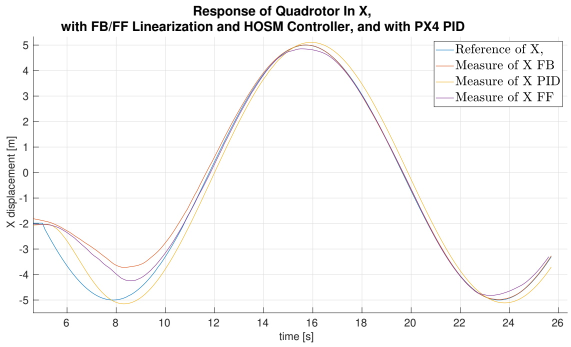
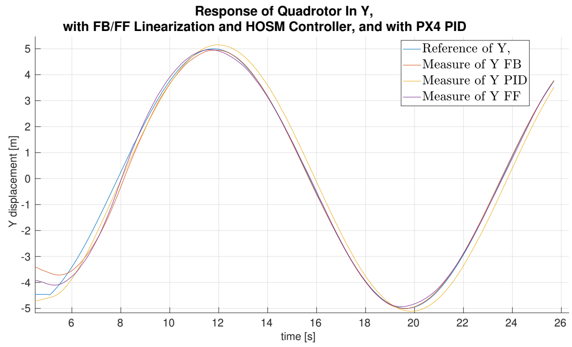


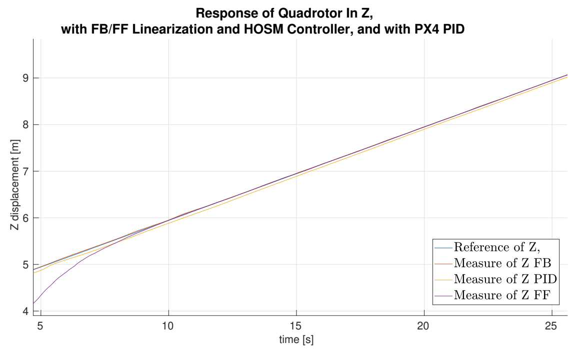
Figure 6.2. 3D trajectory once tracked (slow helix)



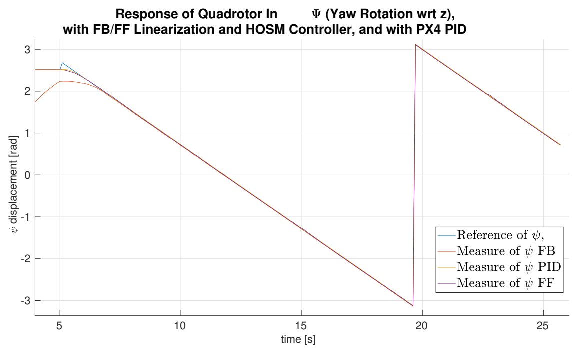
(a) X tracking



(b) Y tracking



(c) Z tracking



(d) Heading tracking

Figure 6.3. Tracking performance slow helix

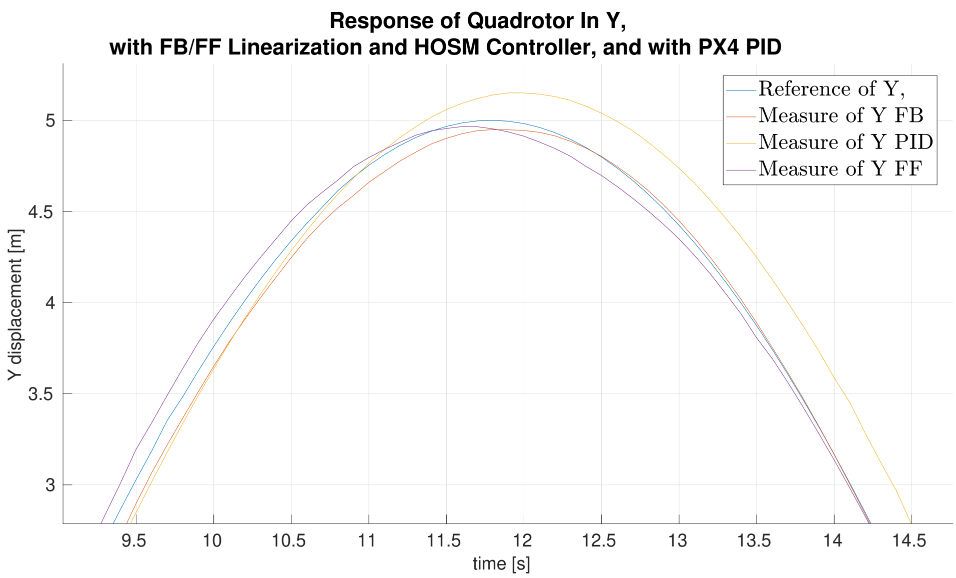
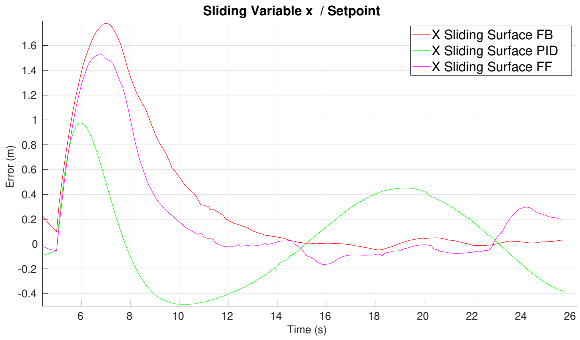
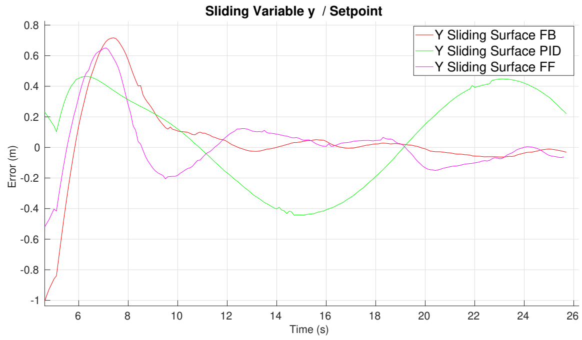


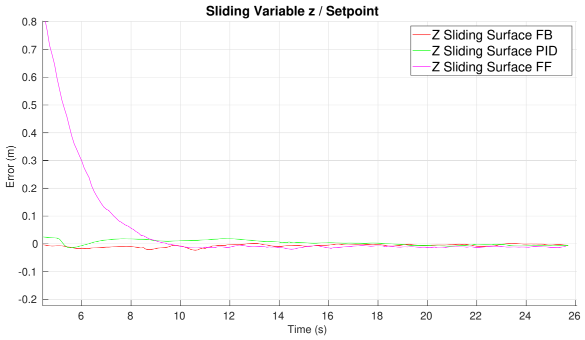
Figure 6.4. Zoomed Y tracking plot



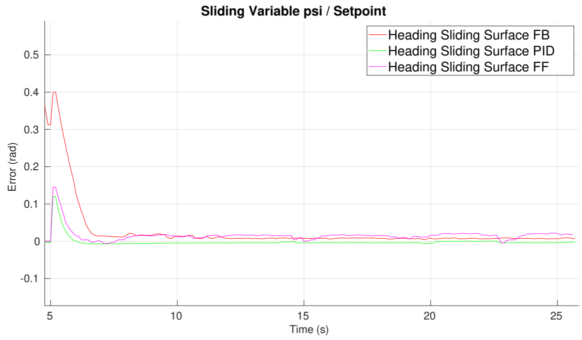
(a) X error



(b) Y error



(c) Z error



(d) Heading error

Figure 6.5. Sliding Surfaces Slow Helix

State	**Index**	**HOSM FB**	**HOSM FF**	**PID**
X	ISE	13.300	6.435	6.35
	RMSE	0.718	0.501	0.496
Y	ISE	30.761	11.99	18.32
	RMSE	1.09	0.683	0.843
Z	ISE	0.019	25.63	0.011
	RMSE	0.027	0.998	0.0205
$\boldsymbol{\psi}$	ISE	20.66	2.899	4.549
	RMSE	0.895	0.336	0.419
AVG	ISE	16.18	11.74	7.309
	RMSE	0.683	0.629	0.445

Table 6.1. Performance Index during the entire Slow Helical trajectory

State	Index	HOSM FB	HOSM FF	PID
X	ISE	0.0151	0.1974	1.01
	RMSE	0.0351	0.124	0.282
Y	ISE	0.0147	0.072	1.476
	RMSE	0.0338	0.0755	0.339
Z	ISE	3.175e-4	0.0017	3.767e-4
	RMSE	0.005	0.0114	0.0055
ψ	ISE	7.52e-4	0.0031	1.785e-4
	RMSE	0.0077	0.0155	0.037
AVG	ISE	0.0077	0.0686	0.623
	RMSE	0.0204	0.0569	0.1577

Table 6.2. Performance Index once the Slow Helical trajectory is tracked

Fast helix

Fast helix (Figure 6.6), despite from the slow one, has some more errors with HOSM when the trajectory starts. In this first transient there is a huge difference between PID and our controller as can be seen in Table 6.3 looking at the AVG. Once the transient is ended, the HOSM control with FF linearization performs really good and much better than PID, while the HOSM control with FB has similar performances in terms of AVG error. This can be seen both from Figure 6.9, and Table 6.4. The reason for FB worse performances it's probably caused by fast motion that generate higher measurements errors, bringing to a wrong linearization.

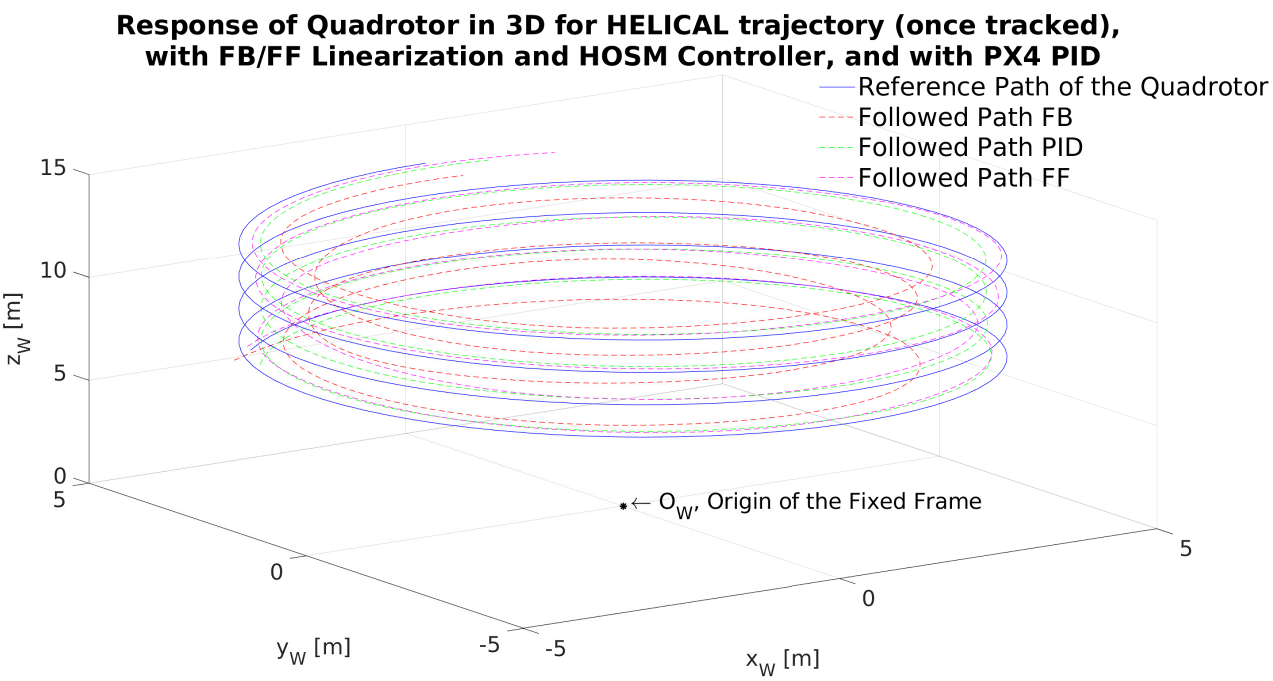
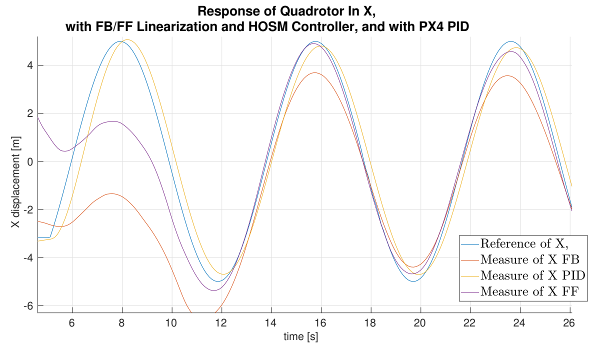
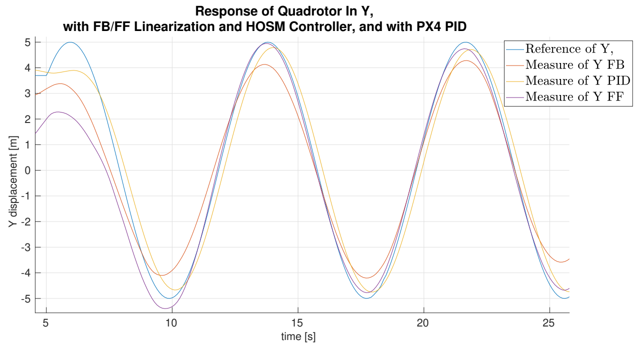


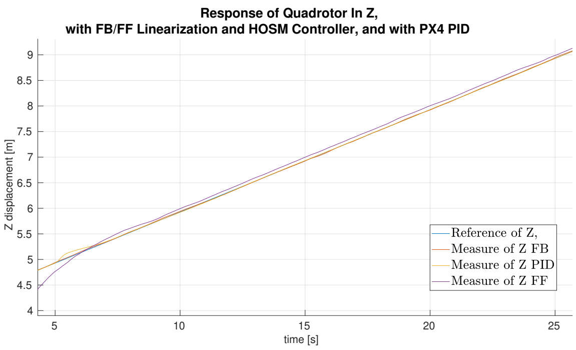
Figure 6.6. 3D trajectory once tracked (Fast Helix)



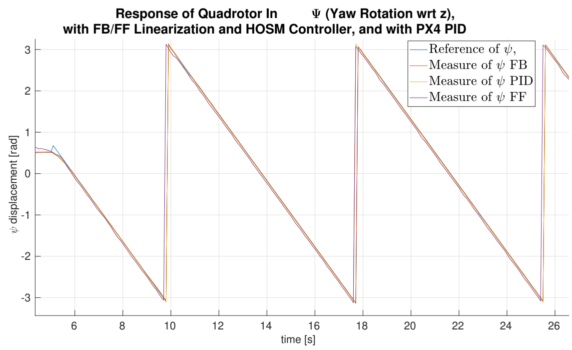
(a) X tracking



(b) Y tracking



(c) Z tracking



(d) Heading tracking

Figure 6.7. Tracking Performance Fast Helix

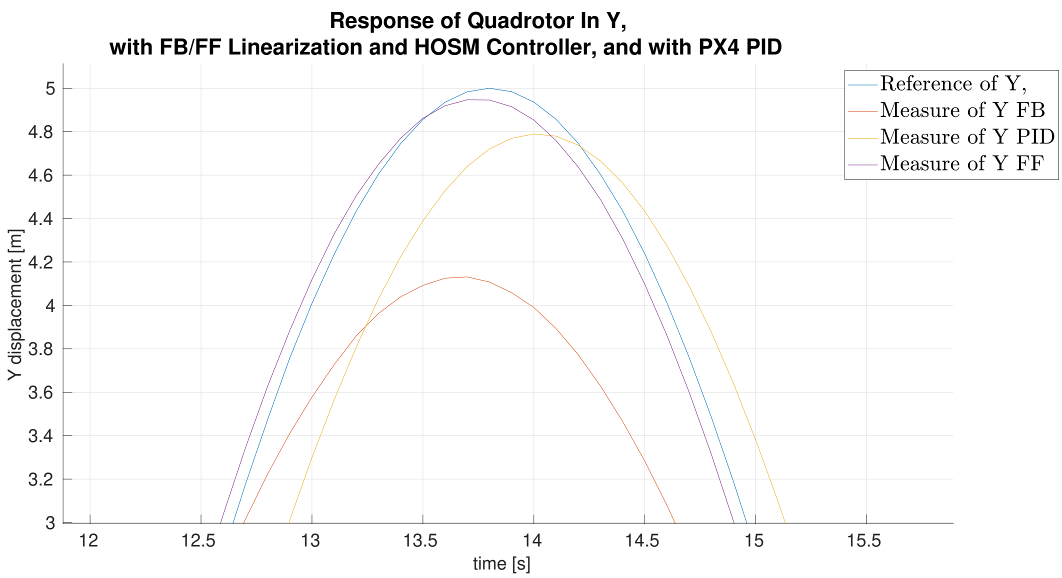
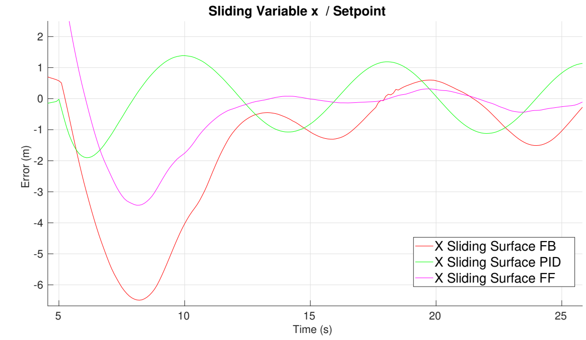
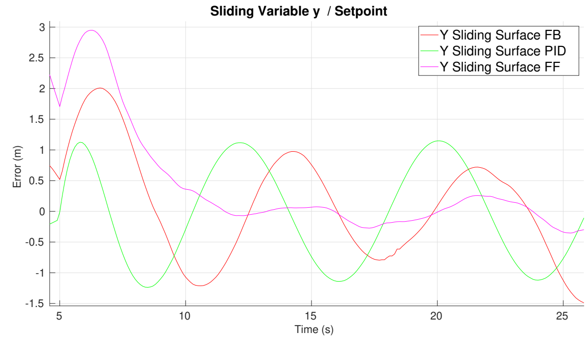


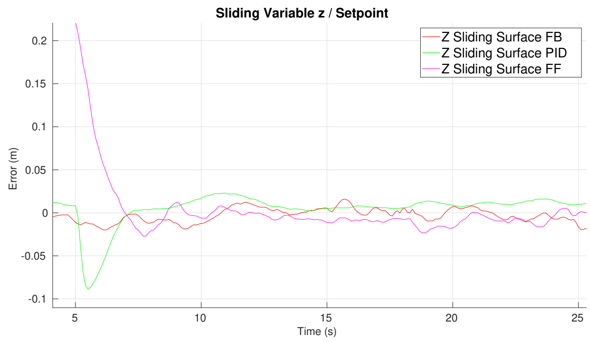
Figure 6.8. Zoomed Y tracking plot



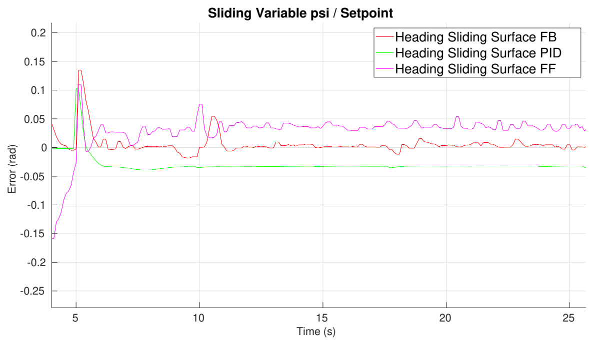
(a) X error



(b) Y error



(c) Z error



(d) Heading error

Figure 6.9. Sliding Surfaces Fast Helix

State	Index	HOSM FB	HOSM FF	PID
X	ISE RMSE	171.41 1.943	196.264 2.070	39.137 0.925
Y	ISE RMSE	51.12 1.943	108.376 1.538	37.39 0.903
Z	ISE RMSE	0.0162 0.0189	5.151 0.335	0.0131 0.0170
$\boldsymbol{\psi}$	ISE RMSE	3.217 0.895	1.443 0.1776	0.902 0.140
AVG	ISE RMSE	56.44 0.822	77.809 1.03	19.362 0.4965

Table 6.3. Performance Index during the entire Fast Helical trajectory

State	Index	HOSM FB	HOSM FF	PID
X	ISE	19.88	1.57	21.31
	RMSE	0.782	0.2185	0.806
Y	ISE	19.49	1.40	20.47
	RMSE	0.775	0.2065	0.7898
Z	ISE	0.002	0.004	0.0025
	RMSE	0.0079	0.0111	0.0087
ψ	ISE	0.0446	0.0446	0.0344
	RMSE	0.0052	0.0369	0.032
AVG	ISE	9.845	0.755	10.458
	RMSE	0.392	0.118	0.409

Table 6.4. Performance Index once the Fast Helical trajectory is tracked

Final Consideration

The HOSM control technique demonstrates to be a powerful control strategy, since it is quite robust in front of measurements errors, especially with FF linearization. This work had the only aim of implementing the controller on the PX4 architecture, but with further study and analysis the HOSM performance can be improved a lot. Given these first results, it seems a very promising controller and a good alternative to classical PID approaches.

Chapter 7

Conclusions

The main contributions of this thesis are:

- Understanding of PX4 firmware architecture, its main features and a short guide on how to make modification to it.
- A detailed analysis of PX4 control strategy.
- Implementation and testing of Feedback/Feedforward Linearization tecnique plus HOSM controller on a real-time embedded system.

Firstly, an overall description of PX4 world was done, the main characteristics and peculiarities of this system are described to the reader and a short guide that teach how to read on topic messages and how to create a new module structure in its architecture.
Then a detailed description of its control architecture was made, the PID cascade structure has been investigated in order to execute possible modification to the intrinsic controller of the PX4.
In the last part of the thesis there is the actual implementation of the new controller showing briefly all the necessary phases for the implementation and the used techniques to interface the two control architectures. Finally, the achieved results and performance are showed.
Further studies may focus on how to improve the used control strategies, for example designing other types of HOSM control aimed at chattering alleviation, or to make modification on the inner estimator in order to have the states value on an higher frequency.
Finally as future development the main idea is to test this non linear control strategy on a real quadcopter application, thus performing real world experiments.

Appendix A

PX4 Code Development Tutorial

A.1 Create uORB message

Although having a lot of built-in topics, it may be needed to add new ones [27]. To add a new topic, it is required to create a new .msg file in the "/msg" directory (Figure A.1) and add the file name to the "msg/CMakeLists.txt" list (Figure A.2).
Example:

msg > my_message.msg

```
# This is my custom message topic

uint64 timestamp                # time since system start (microseconds)

# Position in local NED frame
float32 x                       # North position in NED earth-fixed frame, (metres)
float32 y                       # East position in NED earth-fixed frame, (metres)
float32 z                       # Down position (negative altitude) in NED earth-fixed frame,

# Orientation
float32 roll_body               # body angle in NED frame
float32 pitch_body              # body angle in NED frame
float32 yaw_body                # body angle in NED frame

```

Figure A.1. New message definition

Note: in all the message files the timestamp variable must be included to synchronize the system.

msg > CMakeLists.txt

```
        led_control.msg
        log_message.msg
        logger_status.msg
        mag_worker_data.msg
        manual_control_setpoint.msg
        manual_control_switches.msg
        mavlink_log.msg
        mission.msg
        mission_result.msg
        mount_orientation.msg
        multirotor_motor_limits.msg
        my_message.msg
        navigator_mission_item.msg
```

Figure A.2. New message included into the CmakeLists

Note: a message can be used nested in other messages. For example "set-point_triplet" includes the message "position_setpoint" type.

A.2 Create Mavlink message

It is usually not needed to add new Mavlink messages, the main reason could be to introduce new sensors or devices that do not have a driver already implemented in the firmware. How to create new messages and add them in the firmware is described here [60].

A.3 Create an application

An application is the simplest function that can be implemented on PX4, on their website a brief tutorial that explain how the main APIs for PX4 work can be found, in particular how to publish/subscribe to a topic [61]. In order to create a new application the following procedure must be followed:

- Create a new directory **PX4-Autopilot/src/examples/MyTutorial**,
- Create a new C file in that directory named **MyTutorial.c** (Figure A.3),

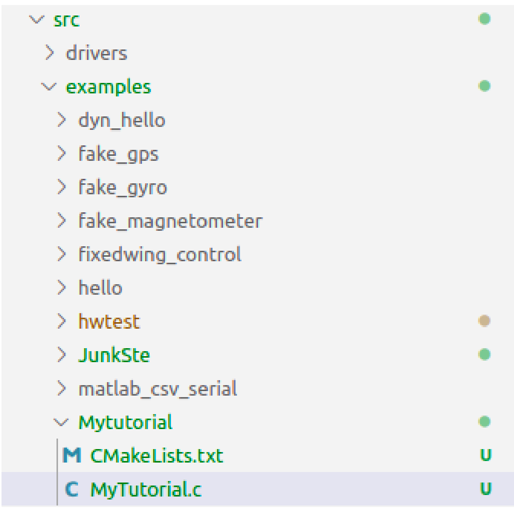


Figure A.3. Directory creation for new application

- Create and open a new cmake definition file named CMakeLists.txt (Figure A.4), src > examples > Mytutorial > CMakeLists.txt

```
############################################################################

px4_add_module(
        MODULE examples__MyTutorial     #folder path from src
        MAIN MyTutorial         #main to call from command line
        #STACK_MAIN 2000        #Size of the stack for the main function
        SRCS
                MyTutorial.c    #file c that include the main
        DEPENDS
)
        #To enable the compilation of the application into the firmware create a new
        #line for your application somewhere in the cmake file:
        #board/px4/sitl/default.cmake
```

Figure A.4. CMakeLists new application

The px4_add_module method (Figure A.4) builds a static library from a module description where:

- The MODULE block is the Firmware-unique name of the module (by convention the module name is prefixed by parent directories back to src).
- The MAIN block lists the entry point of the module, which registers the command with NuttX so that it can be called from the PX4 shell or SITL console.

- Write the a code into MyTutorial.c including all the topic and libraries needed (Figure A.5),

```c
/**
 * @file px4_simple_app.c
 * Minimal application example for PX4 autopilot
 *
 * @author Example User <mail@example.com>
 */

#include <px4_platform_common/px4_config.h>
#include <px4_platform_common/tasks.h>
#include <px4_platform_common/posix.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <poll.h>
#include <string.h>
#include <math.h>

#include <uORB/uORB.h>
#include <uORB/topics/vehicle_acceleration.h>
#include <uORB/topics/vehicle_attitude.h>
```

Figure A.5. Include section new application

- Write the main as <module name>_main (Figure A.6),

```
/*The main function must be named <module_name>_main
        and exported from the module as shown.   */
__EXPORT int MyTutorial_main(int argc, char *argv[]);

int MyTutorial_main(int argc, char*argv[]){

        /* PX4_INFO allows to print text on the console */
        int firstTutorial=1;
        PX4_INFO("Hello sky!\nWelcome on the tutorial %d",firstTutorial);
```

Figure A.6. Main function new application

- Subscribe and advertise the topics where to publish/read informations, then in order to be sure of synchronization and to avoid calculations on the wrong message, wait for new data on that topic ("POLLIN" event) before going on (Figure A.7),

```
        /* Subscribe to a topic */
        int sensor_sub_fd = orb_subscribe(ORB_ID(vehicle_acceleration));
        /* limit the update rate to 5 Hz (200 are millisecons) */
        orb_set_interval(sensor_sub_fd, 200);

        /* advertise to a topic */
        struct vehicle_attitude_s att;  //create a struct with the topic name
        memset(&att, 0, sizeof(att));   //on this struct allocate memory
        orb_advert_t att_pub = orb_advertise(ORB_ID(vehicle_attitude), &att);   //advertise on the allocated memory

        /* Wait for topic publication
        one could wait for multiple topics with this technique,
          just using one here */
        px4_pollfd_struct_t fds[] = {
                { .fd = sensor_sub_fd,   .events = POLLIN },
                /* there could be more file descriptors here, in the form like:
                 * { .fd = other_sub_fd,   .events = POLLIN },
                 */
        };
```

Figure A.7. Subscribe and Advertise to a topic new application

- Good practice is to check if some errors are present in the message lecture (Figure A.8),

```
        int error_counter = 0; //just for checking reasons
        /* wait for sensor update of 1 file descriptor for 1000 ms (1 second) */
                int poll_ret = px4_poll(fds, 1, 1000);

        /* handle the poll result */
                if (poll_ret == 0) {
                        /* this means none of our providers is giving us data */
                        PX4_ERR("Got no data within a second"); //print an error message
                        /*Warnings and errors are additionally added to the ULog and shown on Flight Review*/
         }else if (poll_ret < 0) {
                        /* this is seriously bad - should be an emergency */
                        if (error_counter < 10 || error_counter % 50 == 0) {
                                /* use a counter to prevent flooding (and slowing us down) */
                                PX4_ERR("ERROR return value from poll(): %d", poll_ret);
                        }

                        error_counter++;
```

Figure A.8. Check errors on topics new application

- Create a struct to copy the subscribed message and publish it on the advertised one (Figure A.9).

```
            } else {
                /* Finally check if an update message is arrived*/
                if (fds[0].revents & POLLIN) {
                    /* obtained data for the first file descriptor */
                    struct vehicle_acceleration_s accel;
                    /* copy sensors raw data into local buffer accell defined above*/
                    orb_copy(ORB_ID(vehicle_acceleration), sensor_sub_fd, &accel);
                    PX4_INFO("Accelerometer:\t%8.4f\t%8.4f\t%8.4f",
                            (double)accel.xyz[0],
                            (double)accel.xyz[1],
                            (double)accel.xyz[2]);

                    /* set att and publish this information for other apps
                     the following does not have any meaning, it's just an example
                    */
                    att.q[0] = accel.xyz[0];
                    att.q[1] = accel.xyz[1];
                    att.q[2] = accel.xyz[2];

                    orb_publish(ORB_ID(vehicle_attitude), att_pub, &att);
                }

                /* there could be more file descriptors here, in the form like:
                 * if (fds[1..n].revents & POLLIN) {}
                 */
            }

        PX4_INFO("exiting");

    return 0;
}
```

Figure A.9. Publish on topic new application

A.4 Create a template module

An application (Section A.3) can be written to run as either a task (a module with its own stack and process priority) or as a work queue task (a module that runs on a work queue thread, sharing the stack and thread priority with other tasks on the work queue).
Modules are the fundamental blocks of PX4 architecture, they are parts of the code that runs iteratively following a particular scheduling.
The two different ways to execute a module are:

- **Work queue task**: the module runs on a shared priority queue, meaning that it does not own a proprietary stack (this is the most used case). The main advantage in using such approach is that it requires less RAM, even though the task is not allowed to sleep or poll on messages. Multiple tasks run on the same stack with single priority per work queue. Work queues are essentially used for periodic tasks, such as sensor drivers or the land detector and in particular for control loops (see Section 4.5.1).

  Procedure to follow in order to define a new work queue module:

  1. To create a work queue module it is required to specify it in the CMakeLists.txt (Figure A.10):

```
px4_add_module(
        MODULE examples__work_item
        MAIN work_item_example
        COMPILE_FLAGS
        SRCS
                WorkItemExample.cpp
                WorkItemExample.hpp
        DEPENDS
                px4_work_queue
        )
```

Figure A.10. px4 work queue

2. Then in addition to ModuleBase, the task should derive from WorkItem (included from **WorkItem** or **ScheduleWorkItem.hpp** if the scheduling cycle is specified as in *mc pos control* module (see Listing: 4))

```
class WorkItemExample : public ModuleBase<WorkItemExample>, public ModuleParams, public px4::ScheduledWorkItem
{
public:
        WorkItemExample();
        ~WorkItemExample() override;

        /** @see ModuleBase */
        static int task_spawn(int argc, char *argv[]);

        /** @see ModuleBase */
        static int custom_command(int argc, char *argv[]);

        /** @see ModuleBase */
        static int print_usage(const char *reason = nullptr);

        bool init();

        int print_status() override;

private:
        void Run() override;

        uORB::Publication<orb_test_s> _orb_test_pub{ORB_ID(orb_test)};

        uORB::SubscriptionData<sensor_accel_s> _sensor_accel_sub{ORB_ID(sensor_accel)};

        perf_counter_t  _loop_perf{perf_alloc(PC_ELAPSED, MODULE_NAME": cycle")};
        perf_counter_t  _loop_interval_perf{perf_alloc(PC_INTERVAL, MODULE_NAME": interval")};
};
```

Figure A.11. main ScheduleWorkItem

3. The specific module work queue is set in the constructor method. The ***work_ item*** example set the wq_configurations::test1 work queue as shown in Figure A.12

```
WorkItemExample::WorkItemExample() :
        ModuleParams(nullptr),
        //ScheduledWorkItem(MODULE_NAME, :: NewSchedule)
        ScheduledWorkItem(MODULE_NAME, px4::wq_configurations::test1)
{
}
```

Figure A.12. PX4 Configuration work queue

4. Implement the ScheduledWorkItem::Run() method which is the one iteratively called (Figure A.13)

```
void WorkItemExample::Run()
{
        if (should_exit()) {
                ScheduleClear();
                exit_and_cleanup();
                return;
        }

        perf_begin(_loop_perf);
        perf_count(_loop_interval_perf);

        // DO WORK

        // Example
        // grab latest accelerometer data
        _sensor_accel_sub.update();
        const sensor_accel_s &accel = _sensor_accel_sub.get();

        // Example
        // publish some data
        orb_test_s data{};
        data.timestamp = hrt_absolute_time();
        data.val = accel.device_id;
        _orb_test_pub.publish(data);

        perf_end(_loop_perf);
}
```

Figure A.13. PX4 WorkItem run()

5. Implement the task_spawn method, specifying that the module is a work queue (using the task_id_is_work_queue id).
6. Schedule the module using one of the scheduling methods (in Figure A.14 we used ScheduleOnInterval from within the init method).

```
bool WorkItemExample::init()
{
        ScheduleOnInterval(1000_us); // 1000 us interval, 1000 Hz rate

        return true;
}
```

Figure A.14. Set schedule interval

- **Tasks**: the module runs on its own task with its own stack and process priority so it is independent from any queues. The main problem is that it requires a lot of computational resources, so it is the less used module type.

  PX4 firmware has a built-in module template available on directory "PX4-AUTOPILOT/src/templates/module"; looking on PX4 website also a short introduction to a Task module is shown [62].

A.4.1 Run a new module

1. The first operation is to create the module. Any module is composed by 3 fundamental components:
   - The .cpp file, implementing the actual code.

- The .h file containing all the class declarations and libraries.
- The CMakeLists.txt file needed to build the code [63].

2. The second operation is to place the module folder inside the correct directory. The directory depends on the kind of module that is created (driver, module, example, system commands etc.), nevertheless, the starting point is always the same: “PX4-AUTOPILOT/src”.

3. The third operation is let the system knows that the new module is available. In order to do so, go into “Firmware/boards/px4” and add to the target platform “default.cmake” the name of the module within the associated list (see Figure A.15).

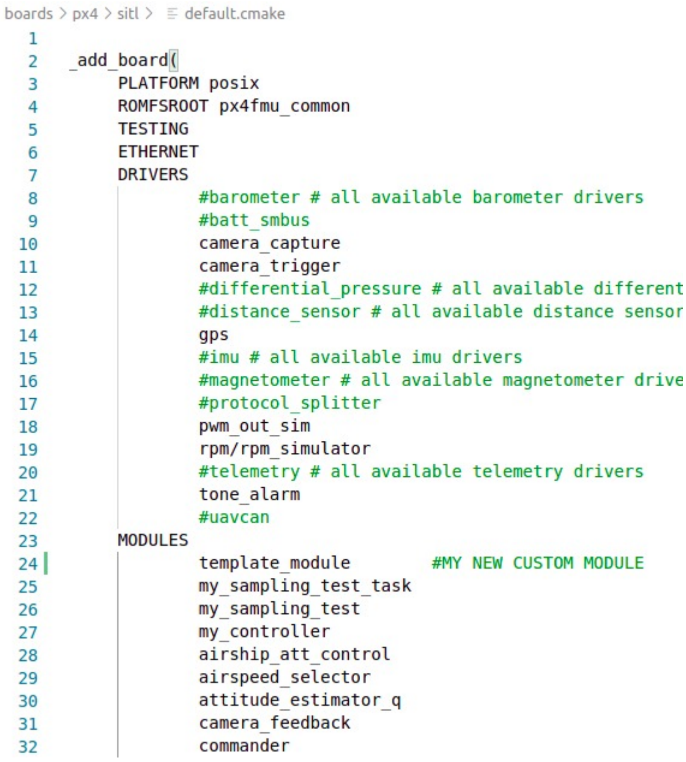


Figure A.15. Module added into default cmake

4. Finally, test the module making the firmware with a command shell (typing “**make px4_sitl gazebo**” if working in SITL).

Typing “**help**” from the shell you should see the commands that can be launched, and if correctly implemented, also the one for start the new module created (in our case “template_module”).
Note: The module can also be launched automatically when the software starts setting it in: “**ROMFS/px4fmu_common/init.d/rc.mc_apps**”, adding the line “**template_module start**”.

A.4.2 Module Structure

Most of the PX4 class modules have the methods and attributes showed in Figure A.16, and called in sequence as shown in Figure 3.2.

```
src > modules > template_module > C template_module.h > TemplateModule > _parameter_update_sub
#include <px4_platform_common/module.h>
#include <px4_platform_common/module_params.h>
#include <uORB/Subscription.hpp>
#include <uORB/SubscriptionInterval.hpp>
#include <uORB/topics/parameter_update.h>

extern "C" __EXPORT int template_module_main(int argc, char *argv[]);
using namespace time_literals;

class TemplateModule : public ModuleBase<TemplateModule>, public ModuleParams
{
public:
	TemplateModule(int example_param, bool example_flag);

	virtual ~TemplateModule() = default;

	/** @see ModuleBase */
	static int task_spawn(int argc, char *argv[]);

	/** @see ModuleBase */
	static TemplateModule *instantiate(int argc, char *argv[]);

	/** @see ModuleBase */
	static int custom_command(int argc, char *argv[]);

	/** @see ModuleBase */
	static int print_usage(const char *reason = nullptr);

	/** @see ModuleBase::run() */
	void run() override;

	/** @see ModuleBase::print_status() */
	int print_status() override;

private:
	/**
	 * Check for parameter changes and update them if needed.
	 * @param parameter_update_sub uorb subscription to parameter_update
	 * @param force for a parameter update
	 */
	void parameters_update(bool force = false);


	DEFINE_PARAMETERS(
		(ParamInt<px4::params::SYS_AUTOSTART>) _param_sys_autostart,   /**< example parameter */
		(ParamInt<px4::params::SYS_AUTOCONFIG>) _param_sys_autoconfig  /**< another parameter */
	)

	// Subscriptions
	uORB::SubscriptionInterval _parameter_update_sub{ORB_ID(parameter_update), 1_s};
```

Figure A.16. Generic hpp file for module creation

public:

- **Constructor/Deconstructor**
  Classical contructor/deconstructor for classes. In the constructor the fixed and the initial parameters used by the module are set, for example for **work queue** modules the queue priority is set. Some parameters are used to check conditions for vehicle state and are updated periodically in the run module.

- **Run()** (Figure A.17)
  It is what the module does by default without custom commands, usually it is cyclic (if a work queue) and is continuously executed in background until the module is stopped or other commands are received.

```
void TemplateModule::run()
{
	// Example: run the loop synchronized to the sensor_combined topic publication
	int sensor_combined_sub = orb_subscribe(ORB_ID(sensor_combined));

	px4_pollfd_struct_t fds[1];
	fds[0].fd = sensor_combined_sub;
	fds[0].events = POLLIN;

	// initialize parameters
	parameters_update(true);

	while (!should_exit()) {

		// wait for up to 1000ms for data
		int pret = px4_poll(fds, (sizeof(fds) / sizeof(fds[0])), 1000);

		if (pret == 0) {
			// Timeout: let the loop run anyway, don't do `continue` here

		} else if (pret < 0) {
			// this is undesirable but not much we can do
			PX4_ERR("poll error %d, %d", pret, errno);
			px4_usleep(50000);
			continue;

		} else if (fds[0].revents & POLLIN) {

			struct sensor_combined_s sensor_combined;
			orb_copy(ORB_ID(sensor_combined), sensor_combined_sub, &sensor_combined);
			// TODO: do something with the data...

		}

		parameters_update();
	}

	orb_unsubscribe(sensor_combined_sub);
}
```

Figure A.17. module run()

- **custom_command(int argc, char *argv[])** (Figure A.18)
  It allows to set up new actions for custom commands. For example for the commander module you can type: "**commander takeoff**", takeoff is a commander custom command that make the drone to take off.

```
int TemplateModule::custom_command(int argc, char *argv[])
{

	if (!is_running()) {
		print_usage("not running");
		return 1;
	}

	// additional custom commands can be handled like this:
	if (!strcmp(argv[0], "do-something")) {
		//get_instance()->do_something();
```

Figure A.18. module custom command()

- ***instantiate(int argc, char *argv[])** (Figure A.19)
  It is used only for **task** modules, not **work queue**. The method launches a new instance of the module, and enables some internal state conditions based on the given argument.

```
TemplateModule *TemplateModule::instantiate(int argc, char *argv[])
{
	int example_param = 0;
	bool example_flag = false;
	bool error_flag = false;

	int myoptind = 1;
	int ch;
	const char *myoptarg = nullptr;

	// parse CLI arguments
	while ((ch = px4_getopt(argc, argv, "p:f", &myoptind, &myoptarg)) != EOF) {
		switch (ch) {
		case 'p':
			example_param = (int)strtol(myoptarg, nullptr, 10);
			break;

		case 'f':
			example_flag = true;
			break;

		case '?':
			error_flag = true;
			break;

		default:
			PX4_WARN("unrecognized flag");
			error_flag = true;
			break;
		}
	}

	if (error_flag) {
		return nullptr;
	}

	TemplateModule *instance = new TemplateModule(example_param, example_flag);

	if (instance == nullptr) {
		PX4_ERR("alloc failed");
	}

	return instance;
}
```

Figure A.19. module instantiate()

- **task_spawn(int argc, char *argv[])** (Figure A.20)
  It creates the task ID and set its priority in the work queque or add the task to a work_queue. For **work queue** modules it simply store the new work queue instance and its ID to the relative queue (defined.

```
int TemplateModule::task_spawn(int argc, char *argv[])
{
	_task_id = px4_task_spawn_cmd("module",
				      SCHED_DEFAULT,
				      SCHED_PRIORITY_DEFAULT,
				      1024,
				      (px4_main_t)&run_trampoline,
				      (char *const *)argv);

	if (_task_id < 0) {
		_task_id = -1;
		return -errno;
	}

	return 0;
}
```

Figure A.20. module task spawn()

- **print_usage(const char *reason = nullptr)** (Figure A.21)
  It briefly explains which is the usage of the module

```
int TemplateModule::print_usage(const char *reason)
{
	if (reason) {
		PX4_WARN("%s\n", reason);
	}

	PRINT_MODULE_DESCRIPTION(
		R"DESCR_STR(
### Description
Section that describes the provided module functionality.

This is a template for a module running as a task in the background with start/stop/status functionality.

### Implementation
Section describing the high-level implementation of this module.

### Examples
CLI usage example:
$ module start -f -p 42

)DESCR_STR");

	PRINT_MODULE_USAGE_NAME("module", "template");
	PRINT_MODULE_USAGE_COMMAND("start");
	PRINT_MODULE_USAGE_PARAM_FLAG('f', "Optional example flag", true);
	PRINT_MODULE_USAGE_PARAM_INT('p', 0, 0, 1000, "Optional example parameter", true);
	PRINT_MODULE_USAGE_DEFAULT_COMMANDS();

	return 0;
}
```

Figure A.21. module print usage()

- **print_status()** (Figure A.22)
  It simply prints out if the module is running and its states conditions.

```
int TemplateModule::print_status()
{
	PX4_INFO("Running");
	// TODO: print additional runtime information about the state of the module

	return 0;
}
```

Figure A.22. module print status()

- **init()** (Figure A.23)
  It defines the scheduling method and frequency used by the **work queue** module.
  - **ScheduleOnInterval()**, schedule the Run() method at a desired frequency
  - **<topic_name>.registerCallback()**, schedule the Run() method with respect to measurement frequency data coming (data driven).
  - **ScheduleNow()**, schedule the module simply based on its queue priority

```
bool my_controller::init()
{
        // execute Run() on every sensor_accel publication
        /* if (!_vehicle_angular_velocity_sub.registerCallback()) {
                PX4_ERR("vehicle_angular_velocity callback registration failed!");
                return false;
        } */

        //Run when possible
        _time_stamp_last_loop = hrt_absolute_time();
        //ScheduleNow();
        // alternatively, Run on fixed interval
        ScheduleOnInterval(my_SAMPLING_TIME * 1e6f);    //  2000 Hz rate

        return true;
}
```

Figure A.23. module init()

private:

- **void parameters_update(bool force = false)** (Figure A.24)
  It updates the parameters used by the module.

```
void TemplateModule::parameters_update(bool force)
{
        // check for parameter updates
        if (_parameter_update_sub.updated() || force) {
                // clear update
                parameter_update_s update;
                _parameter_update_sub.copy(&update);

                // update parameters from storage
                updateParams();
        }
}
```

Figure A.24. module parameter update()

- **Parameters and Subscription/Publication** (Figure A.25)
  Set the parameters to be used and all the topic subscription/publication.

```
private:
        void Run() override;

        Takeoff _takeoff; /**< state machine and ramp to bring the vehicle off the ground without jumps */

        orb_advert_t _mavlink_log_pub{nullptr};

        uORB::Publication<takeoff_status_s> _takeoff_status_pub{ORB_ID(takeoff_status)};
        uORB::Publication<vehicle_attitude_setpoint_s>  _vehicle_attitude_setpoint_pub;
        uORB::Publication<vehicle_local_position_setpoint_s> _local_pos_sp_pub{ORB_ID(vehicle_local_position_setpoint)};

        uORB::SubscriptionCallbackWorkItem _local_pos_sub{this, ORB_ID(vehicle_local_position)};        /**< vehicle loca

        uORB::SubscriptionInterval _parameter_update_sub{ORB_ID(parameter_update), 1_s};

        uORB::Subscription _control_mode_sub{ORB_ID(vehicle_control_mode)};
        uORB::Subscription _hover_thrust_estimate_sub{ORB_ID(hover_thrust_estimate)};
        uORB::Subscription _trajectory_setpoint_sub{ORB_ID(trajectory_setpoint)};
        uORB::Subscription _vehicle_land_detected_sub{ORB_ID(vehicle_land_detected)};
        uORB::Subscription _vehicle_constraints_sub{ORB_ID(vehicle_constraints)};
```

Figure A.25. module subcription,publication

Note: an additional tutorial, with step-by-step examples, can be found in the “HoverGames” website [64].

Acronyms

API	Application Programming Interface
DOF	Degree of Freedom
ESCs	Elettronic Speed Controller
EKF	Extended Kalman Filter
FB	FeedBack linearization
FF	FeedForward linearization
GCS	Ground Control Station
QGC	QGroundController
HITL	Hardware in The Loop
HOSM	High Order Sliding Mode
ISE	Integral Square Error
NED	North East Down
PID	Proportional Integral Derivative
SITL	Simulation in The Loop
SIH	Simulation in Hardware
ROS	Robotic Operating System
RMSE	Root Main Square Error
RTOS	Real-Time Operating System
PWM	Pulse Width Modulation
RC	Remote Controller
RTL	Return To Launch
SDK	Software Development Kit
UAV	Unmanned Aerial Vehicle

VTOL	Vertical Take-Off and Landing

Bibliography

[1] Ardupilot, “Ardupilot,.” https://ardupilot.org/copter/index.html#, 2021. Accessed: 2020-04-30.

[2] dronecode, “Dronecode.” https://www.dronecode.org/, 2021. Accessed: 2020-04-30.

[3] “Dji Mavic Mini.” https://flytodiscover.it/prodotto/dji-mavic-mini-solo-drone/, 2021. Accessed: 2020-04-30.

[4] C. Gordon, “Advancing ai drone insights in the transportation and logistics industry,” 2021. Accessed: 2020-04-30.

[5] N. JOSHI, “10 stunning applications of drone technology.” https://www.allerin.com/blog/10-stunning-applications-of-drone-technology, 2021. Accessed: 2020-04-30.

[6] I. Intelligence, “Drone technology uses and applications for commercial, industrial and military drones in 2021 and the future.” https://www.businessinsider.com/drone-technology-uses-applications?r=US&IR=T, 2021. Accessed: 2020-04-30.

[7] Airobotics, “AUTOMATED DRONE APPLICATIONS FOR INDUSTRIAL PURPOSES.” https://www.airoboticsdrones.com/applications/, 2021. Accessed: 2020-04-30.

[8] D. Ackerman, “Researchers introduce a new generation of tiny, agile drones.” https://news.mit.edu/2021/researchers-introduce-new-generation-tiny-agile-drones-0302, 2021. Accessed: 2020-04-30.

[9] Aerocorner, “List of 14 Different Types of Drones Explained with Photos.” https://aerocorner.com/blog/types-of-drones/, 2021. Accessed: 2020-04-30.

[10] A. Bouguerra, D. Saigaa, K. Kara, and S. Zeghlache1, “ault-tolerant lyapunov-gain-scheduled pid control of a quadrotor uav.” https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.1076.6143&rep=rep1&type=pdf, 2015.

[11] S. Bouabdallah, A. Noth, and R. Siegwart, “Pid vs lq control techniques applied to an indoor micro quadrotor.” https://ieeexplore.ieee.org/document/1389776, 2018.

[12] K. Benkhoud and S. Bouallègue, "Dynamics modeling and advanced meta-heuristics based lqg controller design for a quad tilt wing uav." https://link.springer.com/article/10.1007/s40435-017-0325-7, 2018.

[13] "Holybro, website." http://www.holybro.com/, 2021. Accessed: 2020-04-30.

[14] "Pixhawk4, online developer guide." https://docs.px4.io/master/en/flight_controller/pixhawk4.html, 2021. Accessed: 2020-04-30.

[15] "Nuttx documentation." https://nuttx.apache.org/docs/latest/, 2021. Accessed: 2021-10-30.

[16] "Overall architecture of nuttx." https://cwiki.apache.org/confluence/display/NUTTX/NuttX+Overview, 2019.

[17] "PX4 Airframes,airframe builds." https://docs.px4.io/master/en/airframes/airframe_reference.html, 2021. Accessed: 2021-10-30.

[18] "QGroundControl, px4." https://docs.qgroundcontrol.com/master/en/, 2021. Accessed: 2020-04-30.

[19] MAVSDK, "MAVSDK, dronecode." https://mavsdk.mavlink.io/main/en/index.html, 2021. Accessed: 2020-04-30.

[20] "Companion computer for pixhawk series." https://docs.px4.io/master/en/companion_computer/pixhawk_companion.html#companion-computer-for-pixhawk-series, 2021.

[21] "Mavlink messaging, development guide." https://docs.px4.io/master/en/middleware/mavlink.html. Accessed: 2020-04-30.

[22] "PX4 Flight Modes, flying." https://docs.px4.io/master/en/flight_modes/#multicopter, 2021. Accessed: 2020-04-30.

[23] "PX4 Flight Modes, development guide." https://docs.px4.io/master/en/concept/flight_modes.html, 2021. Accessed: 2020-04-30.

[24] "PX4 RTPS, development guide." https://docs.px4.io/master/en/middleware/micrortps.html#rtps-dds-interface-px4-fast-rtps-dds-bridge, 2021. Accessed: 2021-10-30.

[25] "PX4 Architectural Overview, development guide." https://docs.px4.io/master/en/concept/architecture.html, 2021. Accessed: 2021-10-30.

[26] "PX4 modules, development guide." https://docs.px4.io/master/en/modules/modules_main.html, 2021. Accessed: 2021-10-30.

[27] "uOrb messaging, development guide." https://docs.px4.io/master/en/middleware/uorb.html, 2021. Accessed: 2020-04-30.

[28] Mavlink, "Mavlink, overview." https://mavlink.io/en/about/overview.html, 2021. Accessed: 2020-04-30.

[29] "Mavlink UDP, development guide." https://docs.px4.io/master/en/simulation/#default-px4-mavlink-udp-ports, 2021. Accessed: 2020-04-30.

[30] "List of all uorb built-in messages." https://github.com/PX4/PX4-Autopilot/tree/master/msg/, 2021.

[31] "PX4 Mixer, development guide." https://docs.px4.io/master/en/concept/mixing.html, 2021. Accessed: 2020-04-30.

[32] "PX4 Development Guide,virtual control groups." https://docs.px4.io/master/en/concept/mixing.html#virtual-control-groups, 2021. Accessed: 2021-09-30.

[33] "List of all mavlink commands." https://mavlink.io/en/messages/common.html, 2021.

[34] mavros, "mavros, px4-ros." https://docs.px4.io/master/en/ros/mavros_installation.html#ros-with-mavros-installation-guide, 2021. Accessed: 2020-04-30.

[35] S. Rapisarda, "Ros-based data structure for service robotics applications." https://webthesis.biblio.polito.it/10933/1/tesi.pdf, 2019.

[36] "Gazebo website." http://gazebosim.org/#features.

[37] "Jmavsim, development guide." https://docs.px4.io/master/en/simulation/jmavsim.html#jmavsim-with-sitl, 2021. Accessed: 2021-04-30.

[38] "PX4 SITL, development guide." https://docs.px4.io/master/en/simulation/, 2021. Accessed: 2021-10-30.

[39] G. Joachim Breitner, University of Pennsylvana; Chris Smith, "Lock-step simulation is child's play (experience report)." https://www.researchgate.net/publication/319370697_Lock-step_simulation_is_child's_play_experience_report, 2017.

[40] "Lockstep Simulation, development guide." https://docs.px4.io/master/en/simulation/#lockstep-simulation, 2020.

[41] "PX4 HITL, development guide." https://docs.px4.io/master/en/simulation/hitl.html, 2021. Accessed: 2021-10-30.

[42] "PX4 SIH, development guide." https://docs.px4.io/master/en/simulation/simulation-in-hardware.html, 2021. Accessed: 2021-10-30.

[43] "PX4 EKF Overview, development guide." https://docs.px4.io/master/en/advanced_config/tuning_the_ecl_ekf.html, 2021. Accessed: 2020-04-30.

[44] "PX4 EKF Overview, development guide." https://docs.px4.io/master/en/advanced_config/tuning_the_ecl_ekf.html#states, 2021. Accessed: 2020-04-30.

[45] "Sensor module,github." https://github.com/PX4/PX4-Autopilot/blob/master/src/modules/sensors/sensors.cpp, 2021. Accessed: 2020-04-30.

[46] "PX4 Controllers, development guide." https://docs.px4.io/master/en/modules/modules_controller.html, 2021. Accessed: 2020-04-30.

[47] "Flight Task, development guide." https://docs.px4.io/v1.12/en/concept/flight_tasks.html#flight-tasks, 2021.

[48] "PX4 Controlles Diagram, development guide." https://docs.px4.io/master/en/flight_stack/controller_diagrams.html, 2021. Accessed: 2020-04-30.

[49] S.-C. L. Jong-Woo Choi, "Antiwindup strategy for pi-type speed controller." https://ieeexplore.ieee.org/abstract/document/4801693, 2009.

[50] "Pid controller basics – eliminating derivative kick." https://barela.wordpress.com/2013/07/19/pid-controller-basics-eliminating-derivative-kick/, 2013.

[51] M. D. Brescianini, Dario; Hehn, "Nonlinear quadrocopter attitude control technical report." https://www.research-collection.ethz.ch/bitstream/handle/20.500.11850/154099/eth-7387-01.pdf, 2013.

[52] "PX4 Mixer Saturation and Airmode,advanced configurations." https://docs.px4.io/master/en/config_mc/pid_tuning_guide_multicopter.html#airmode-mixer-saturation, 2021. Accessed: 2020-04-30.

[53] "Geometry files, development guide." https://docs.px4.io/master/en/concept/geometry_files.html#how-to-add-a-new-geometry, 2021. Accessed: 2020-04-30.

[54] Philips, "Pca9685 datasheet." https://www.alldatasheet.com/datasheet-pdf/pdf/293576/NXP/PCA9685.html, 2009.

[55] M. M. Pekkaptan, "Higher order sliding mode control of a flatness-based feedforward/feedback linearized quadrotor." https://www.politesi.polimi.it/bitstream/10589/167341/3/2020_07_Pekkaptan.pdf, 2020.

[56] M. Prandini, "Nonlinear control," 2020.

[57] B. Andritsch, M. Horn, and S. Koch, "The robust exact differentiator toolbox revisited:filtering and discretization features," 2017.

[58] A. Leva, *Laboratorio di automatica.* 2005.

[59] A. R. Shahemabadi, S. B. M. Noor, and F. S. Taip, "Analytical formulation of the integral square error for linear stable feedback control system," 2013.

[60] "Mavlink messaging custom msg, development guide." https://docs.px4.io/master/en/middleware/mavlink.html#defining-custom-mavlink-messages, 2021. Accessed: 2020-04-30.

[61] "Writing your first application, development guide." https://docs.px4.io/master/en/modules/hello_sky.html, 2021. Accessed: 2020-04-30.

[62] "Module Template for full application, development guide." https://docs.px4.io/master/en/modules/module_template.html, 2021. Accessed: 2020-04-30.

[63] "Cmake add module, github px4." https://github.com/PX4/PX4-Autopilot/blob/master/cmake/px4_add_module.cmake, 2021. Accessed: 2020-04-30.

[64] "PX4 Tutorial example code, nxp hover games." https://nxp.gitbook.io/hovergames/developerguide/px4-tutorial-example-code, 2021. Accessed: 2020-04-30.