UNIVERSITY OF PISA
School of Engineering

M.Sc. in Robotics and Automation Engineering

Master's Thesis

Real-Time Support Framework for the Development of Unmanned Aerial Vehicles Software

Supervisor:
Prof. Giorgio Buttazzo
Dott. Mauro Marinoni

Author:
Luigi Pannocchi

Academic Year 2014/2015

Contents

List of Figures

Abstract

The objective of this thesis is to provide a set of tools to support the development of navigation, guidance, and control algorithms for unmanned aerial vehicles. More precisely, the final goal is to build a friendly programming framework for supporting hardware-in-the-loop simulations. The possibility to run the generated code directly on the target device can provide a feasible way to verify and validate the performance of the various control components without requiring the real flight tests. For this reason, this technique reduces the development time and cost, eliminates the risk of crashes, while making use of the control system that will be employed in the final implementation.

The work first presents a preliminary analysis of existing solutions and possible simulator architectures. A more specific analysis of the target device under interest is carried out. Hardware capabilities and limits are discussed to verify the feasibility of the hardware-in-the-loop simulation. The part of the firmware involved in the simulation loop has been inspected and integrated with the required features.

The contribution of this work consists in a deep analysis of the simulation issues (typically neglected in the existing solutions) and in the development of a new structure to take them into account.

The simulation environment developed under this thesis is also able to manage timing constraints specified on the computational activities. Performance tests have been run to verify the reliability of the framework. The obtained results, besides demonstrating the correct functionality of the application, showed which component could be improved in a future work.

Chapters Description

Chapter 1 The first chapter introduces the context of the application, also describing the motivations that drove the thesis. A state of the art is then presented, describing the solutions existing in the literature with their advantages and weaknesses. Starting from this point, the approach used in this work is briefly introduced.

Chapter 2 The second chapter illustrates the overall system architecture in order to acquaint the reader with all the blocks constituting the proposed framework. The main components include the AUV with its autopilot board, the Ground Station, the simulation framework and the developed application that manages the data flow between the various components.

Chapter 3 The third chapter is dedicated to the description of the simulation part of the framework realized with *Matlab/Simulink*. Before illustrating the simulation model, a review of the conventions used when representing the motion of a vehicle are reported. The equations ruling the motion of the AUV are presented together with the models of the various sensors. The chapter ends illustrating the *Simulink* blocks used to implement the mathematical models.

Chapter 4 The fourth chapter describes the features of the Ground Station Application. After a brief introduction to the application's role and its internal structure, the attention is focused on the method used to manage data flows.

Chapter 5 The fifth chapter gives an overview on the autopilot firmware and the applications running on it. Particular attention is devoted to the analysis of the mechanisms for data exchange and synchronization inside the application.

Chapter 6 The sixth chapter describes the component that manages the data flow in the structure considering timing constraints.

Chapter 7 The seventh chapter illustrates the results of the experiments carried out to test the performance of the system.

Chapter 8 The last chapter states the conclusions, summarizing the work done, the achieved results, and providing a list of possible future developments of the current solution to improve its performance.

Chapter 1

Introduction

1.1 Context of the Application

The adoption of autonomous vehicles has clearly increased over the last few decades gradually substituting manned solutions. This success is justified by the distinct advantages they provide: not only do they constitute a safer solution for various applications but they are also more efficient in accomplishing tasks.

Practical applications which see autonomous systems in action comprise operations in hazardous environments, where the human intervention is risky or even impossible. Other tasks, such as surveillance of large areas, benefit from autonomous systems since the amount of information gathered using modern sensors is far superior to the humans' capabilities and the reaction time to events is lower. To provide practical examples, apart from the well known usage for military operations, also civilian applications like agriculture, fire monitoring etc. are arousing great interest in this field. This spread towards the civilian sphere suggests that it will become a common technology, providing new issues to research and work on.

The employment of autonomous vehicles would have been unfeasible if it had not been for technological progress and continuous development of new equipment. It follows that this animated field of study entails new challenges for the engineer who has to provide feasible and efficient solutions in such a variable environment. It is thus of paramount importance to dispose of support frameworks to develop and validate the system under study. One of the most useful tool is the simulation environment which allows to obtain a feedback on the performance of the proposed solution.

Autonomous vehicles, having the autopilot computer onboard, can be included in the category of embedded systems. Indeed an embedded system is a combination of computer hardware, “embedded” into the system to be controlled, and software designed to perform a dedicated function in an environment (Figure 1.1). This structure demands for particular considerations

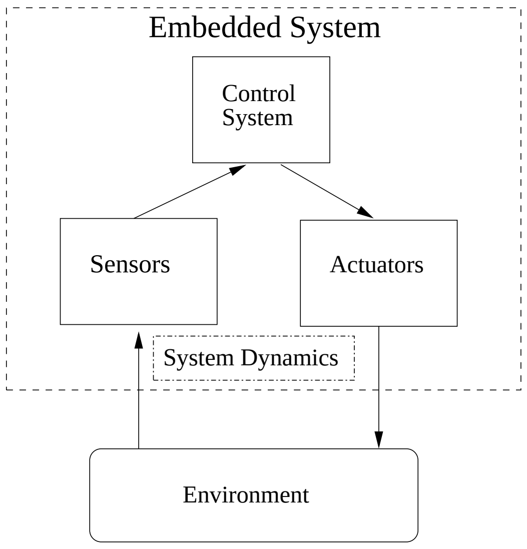


Figure 1.1: Embedded System Scheme

when dealing with the simulation step. When the interest is aimed to the development of a particular algorithm, without taking into account its practical implementation, the main requirement is the capability to reproduce a realistic simulation environment. On the other hand in this field, this approach doesn’t suffice: not only is it necessary to model the behavior of the physical part of the system, but also the computer hardware and software performing the control functions must be taken into consideration if it is required to provide a truthful result. In a simulation environment the structure shown in Figure 1.1 can be subdivided in two blocks (Figure 1.2): the part simulating the plant and the part implementing the control law. The simulation flow can be considered as a loop between these two block composed by two steps:

1. Reaction of the simulated plant, representing the system under control to external stimuli(control input and environment)
2. Computation of a control action by the controller given simulated sensor data

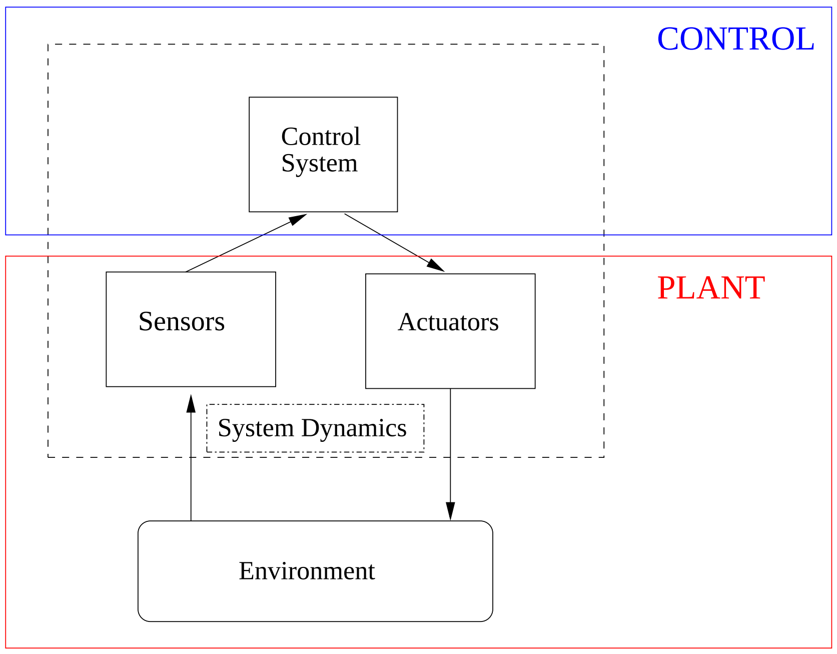


Figure 1.2: Embedded System Simulation Scheme

In the case of theoretical approach to the control problem, the part implementing the control law consists in pure code representing the theoretical aspect of the algorithm, that is, only "what" is written and not "how" is written matters. When moving a step forwards to the practical implementation there are two simulation techniques which attempt to include the architecture of the control system in the simulation loop:

1. Software in the Loop (SIL)
2. Hardware in the Loop (HIL)

The former one consists in bare code running on a computer, that is, all the simulation components are simulated on a host machine (Figure 1.3). To perform this kind of simulation it is necessary to run the code, originally hosted on the embedded system, on the machine running the simulation. Since in this case the control block is a sofware component the technique is called "Software in the Loop". The latter method makes use of a simulated plant on a host computer and the real hardware on which developed algorithms are run (Figure 1.4). The control block in this case consists in a hardware resource and then the name "Hardware in the Loop".

The advantages of a SIL implementation resides in the possibility to run it without having the physical board. Moreover it is possible to debug the applications more easily with respect to HIL case. On the other side the execution is not in realtime and it is difficult to take into account the capabilities of the end device. In chosing which way to follow it is thus necessary to consider the requirements of the particular application. . If even the way the algorithm is implemented is taken into account then the SIL simulation is an useful tool. The HIL implementation, while being slightly more complex and requiring the hardware, has the advantage of being more affine to the real world case.

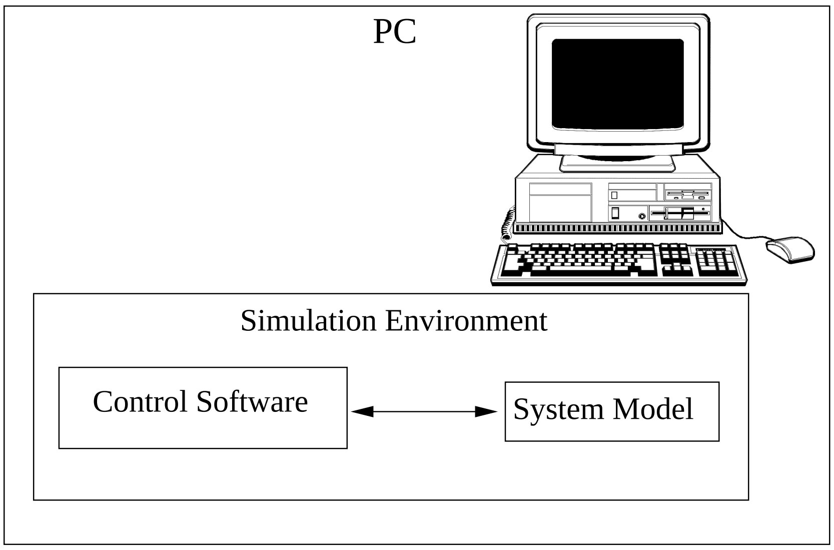


Figure 1.3: Software in the Loop framework

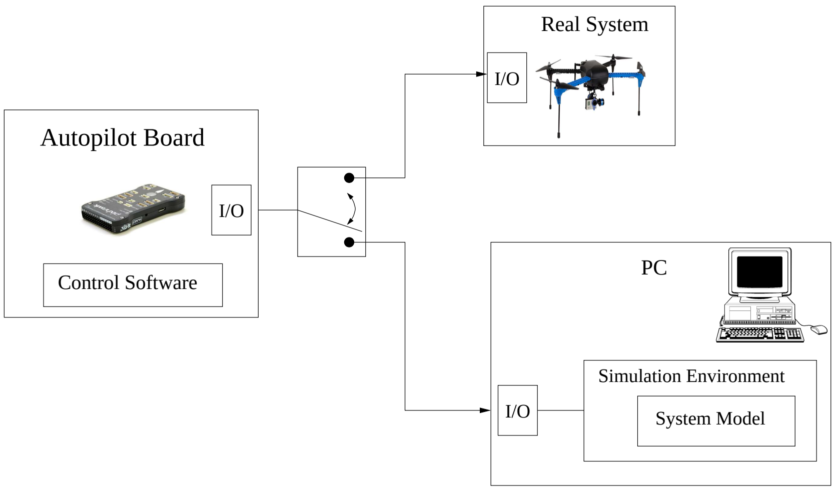


Figure 1.4: Hardware in the Loop framework

In this work the interest is oriented towards the HIL simulation. The reason of this lies in the fact that the purpose of HIL simulation is to provide a test environment for developing and evaluating new algorithms considering also the hardware which will run them. SIL simulation, in spite of a less complex setup, does not allow to inspect the code execution as will be in the real application, losing the capability to check the regularity of control loops, the presence of jitter and eventual bugs in the implementation, whuch could determine application faults etc. Summing up the key benefits of HIL simulation that have been considered are

- Realistic testing of the autopilot performance and capability.
- Possibility to simulate different environment condition.
- Simulation testing is not destructive.
- Simulation testing is safe.
- Reduced development costs.
- Fast development of new solutions with respect to real testing.

Given the capability to provide the fidelity evaluation environment in the laboratory, lot of complex embedded systems, such as guided weapon systems, vehicle's breaking systems, robotic systems, adopt this technique as a testing bed. In this particular case the interest concerns the testing of guidance, navigation and control algorithms directly on the board allowing to have a test bed for the real-time behaviour of the system. As a matter of fact, the knowledge of these information allows the designer of the control architecture to counteract eventual bad events achieving a more robust control performance. Since the objective is to accomplish a more realistic testing environment, it turns out that the development itself of such an environment requires particular attention to the time constraints in the processing of data.

1.2 Related Work

A support framework for the development of new solutions does not consist only in a simulator component but it is also necessary to include the possibility to interact with the system and have a feedback. The switch between the simulation process and the utilization in the real evironment should be as simple as possible. Some works like [4] dealt with the development of evaluation environment of complex embedded systems. Such system reaches high complexity and could be difficult to be maintained, particularly in an academical contex. Other solution like [5] are more feasible, but the low level implementation reduce the scalability of the system. A solution which is easy to maintain and update is preferable. An interesting work which is oriented towards these needs is the *PX4 Project* ([7]). It is supported by the PIXHAWK Project of the Computer Vision and Geometry Lab of ETH Zurich (Swiss Federal Institute of Technology) and by the Autonomous Systems Lab and the Automatic Control Laboratory of the same university. It is aimed to provide a complete solution for the usage and development of UAVs solutions.

1.2.1 PX4 Project

The objectives of the *PX4 Project* are summarised by its creators as:

> "PX4 is an independent, open-source, open-hardware project aiming at providing a high-end autopilot to the academic, hobby and industrial communities (BSD licensed) at low costs and high availability. It is a complete hardware and software platform, much like a computer, and can run multiple autopilot applications."

The testing environment provided by this group is made up of three components:

1. Autopilot Board with firmware
2. Ground Station application
3. Simulator for aerial vehicles

With this configuration (Figure 1.5) they claim to be able to simulate different aerial vehicles, such as multirotors and planes.

The core of the system is the ground station application, which is called *QGroundControl*. This fulfills the functions of user interface with the system, communication with the vehicle and communication with the simulator. The Ground Station application can establish a communication via TCP, UDP, serial or even wifi using radio or Xbees modules. An interesting part of this work is the communication protocol and the respective libraries developed for the data exchange with the autopilot board. This protocol is called *MAVLink*.

The autopilot, which runs a real-time operating system, is equipped with the necessary software to perform a hardware in the loop simulation.

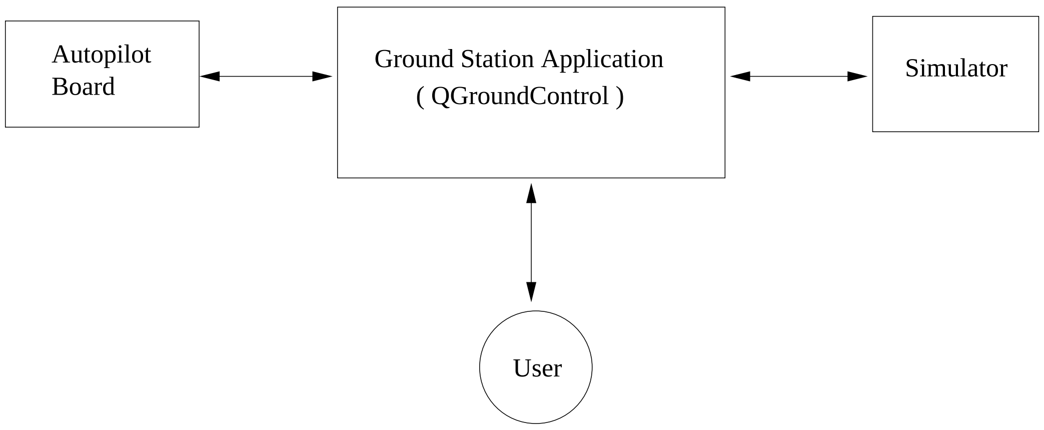


Figure 1.5: PX4 Solution for the HIL simulation environment

The *MAVLink* protocol defines the necessary data structures and the developed software applications manage the switching between “Normal Operating Mode” and “HIL mode”.

Regarding the simulator part of their system they provide several working solutions, including free open-source simulators developed by other groups and comunitites and one commercial flight simulator:

1. X-Plane ([10])
   X-Plane is a very accurate flight simulator which supports fixed wing models. To model flight dynamics it uses the “blade element theory”, that is breaking the aircraft down into many small elements and then finding the areodynamics forces on each little element. These forces are then converted into accelerations, which are then integrated to velocities and positions.

2. FligthGear ([9])
   FlightGear is an open-source flight simulator. It supports a variety of popular platforms (Windows, Mac, Linux, etc.) and is developed by skilled volunteers from around the world. Source code for the entire project is available and licensed under the GNU General Public License. The flight dynamics is obtained using parametric models which give the forces and moment acting on the vehicle independently from the geometric shape of the vehicle.

3. Java-Phyton Simulators ([11])
   These solutions are based on the same open source dynamics libraries used by FlightGear but providing simpler interfaces written usign Java and Phyton code.

1.2.2 Limits of this approach

Whereas that project consitutes a big contribution some flaws have been found which are worth to be considered.

1. Simulators

    - X-Plane
      This solution is not open source and it is not free. The fact that vehicle's model is based on the geometry makes the modeling not a easy deal for someone who is not expert in airfoils and areodinamics.
    - FlightGear
      At first this solution was considered viable as a simulation environment. It is open source and the definition of the model dynamics does not require geometrical modeling of the aircraft. Unfortunately it has a big limitation for what concern the data exchange with other applications. Indeed the frequency at which the simulated data can be provided cannot exceed the framerate. This makes impossible to reach high frequencies which could be necessary for the control algorithms. The attitude control loop of the multirotor used for the test runs at 250 Hz, while the framerate of the simulator is 60 Hz. This coupling makes this simulator unsuitable for our aims.
    - Java-Phyton Simulators
      These solutions were considered unsuitable because they were not completely developed and they were provided without documentation which makes the usage difficult and error prone.

2. Ground Station Application
   Whereas this application is very useful as graphical interface it is not optimally suited to implement a simulation loop. As will be deeply described in the following chapters, the way the data flow is managed in the application is not optimal and lacks of predictability.

3. Autopilot Firwmare
   The firmware of the Autopilot doesn't allow to perform an hardware in the loop simulation if the update frequency required by the algorithm is higher than 50 Hz. This is due to the fact that there are some update limits coded in the drivers which doesn't allow to retrieve the actuator control outputs at the required frequency.

1.3 Contribution of this Work

Using the system described above as a starting point in this work the following improvements have been proposed:

- The firmware of the Autopilot has been modified to support higher data rates.
- A new simulation environment has been proposed to tackle the problems observed with the simulators cited before.
- A new structure for the overall system has been developed to achieve better data exchange considering the different priorities of different simulation tasks.

Chapter 2

System Architecture

The design of a hardware in the loop simulation architecture requires to take into consideration different components. It is necessary to connect together the simulator application, the user interface application and the physical board. The setup proposed by this work is thus composed by a PC running the various programs and the autopilot board. The architecure shown in Figure 2.1 represents visually the interactions between the different parts.

The new idea which stands out here is to split the data flux with the autopilot board in two channels. The data flux concerned with the simulation loop should have higher priority since it is carrying information related to a simulated "real world" with real time flow, while the one related to the user interaction is less relevant in terms of realtime constraints.

The two fluxes are managed with the *Data Governor* application. This is a C/C++ program written usign the ptask libraries which carries out also the timing analysis to evaluate the performance of the system.

The overall architecture comprises the FligthGear application to reproduce a virtual environment. This is possible because of the integrated capability of *Matlab/Simulink* to communicate with it through UDP socket. This feature allows to have a visual feedback on the behavior of the vehicle.

The transmission of data with the board is accomplished over UART interface, while the data exchange between applications running on the PC is made through UDP sockets. The motivations for the usage of UDP communication protocol, instead of the possible TCP, can be summarized as follows

1. No connection establishment
   TCP uses a three-way handshake before it starts to transfer data. This preliminary routine introduces delay in the communication. UDP does not have this feature, thus it allows a more responsive messaging.

2. Smaller transmission overhead

UDP requires less overhead to perform communication, that is slender mechanisms and smaller data usage. The segment header is only 8 bytes versus the 20 of the TCP protocol.

3. Higher transmission rate
   The UDP protocol has no self-regulation of the transmission rate. The absence of the throttling mechanism present in the TCP protocol allows to guarantee transmission rates at the expense of transmission errors.

The reliability of the UDP connection can be achieved implementing checkouts on the application layer. In the following sections the single components

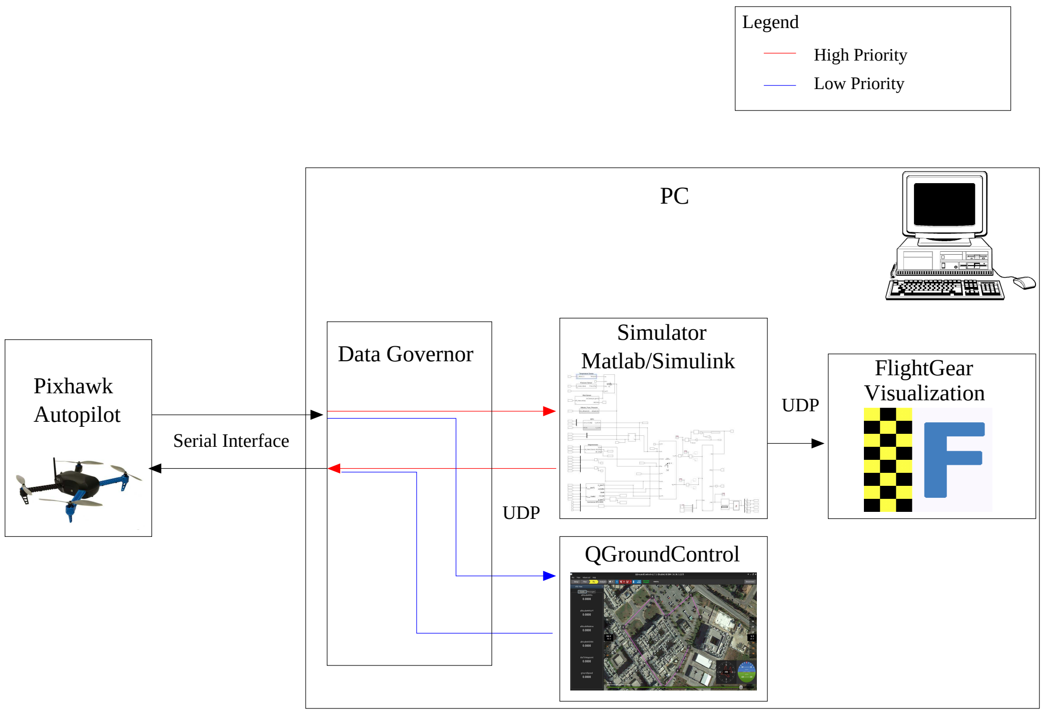


Figure 2.1: Overall System Architecture

are described briefly.

2.1 Quadrotor

The autopilot board under study is mounted on the *3DR Iris+* quadcopter (Figure 2.2) with the following characteristics:

- Autopilot: Pixhawk v2.4.5

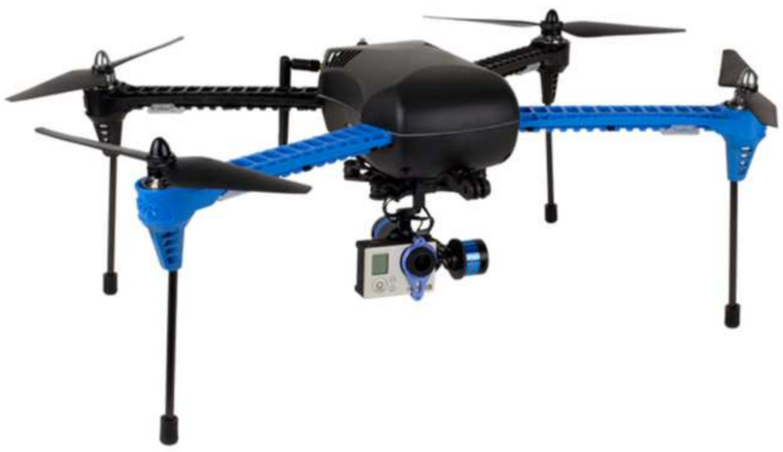


Figure 2.2: The quadcopter model equipped with the selected autopilot board

- GPS: 3DR uBlox GPS with Compass (LEA-6H module, 5 Hz update)
- Telemetry radio: 3DR Radio Telemetry v2 (915 mHz or 433 mHz)
- Motors: 950 kV
- Frame type: V
- Propellers:
  9.5 x 4.5 T-Motor multirotor self-tightening counterclockwise (2)
  9.5 x 4.5 T-Motor multirotor self-tightening clockwise (2)
- Battery: 3S 5.1 Ah 8C lithium polymer
  Low battery voltage: 10.5 V
  Maximum voltage: 12.6 V
- Battery cell limit: 3S
- Battery weight: 320 g
- Weight with battery: 1282 g
- Height: 100 mm
- Motor-to-motor: 550 mm
- Payload capacity: 400 g

- Radio range: up to 1 km
- Flight time: 16-22 minutes

2.2 Simulation Block

The simulation block is the part of the system which should reproduce the behavior of the real plant. The components which are included in it can be seen in the schematic representation in Figure 1.2. In this case the system dynamics is represented by the physics laws ruling the motion of a rigid body in space. The sensors correspond to the ones onboard the vehicle and the actuators are the motors of the quadrotor.

The simulation block is realized using *Matlab/Simulink*. The choice is motivated by several factors and it is aimed to patch the flaws of the originally proposed simulators. It constitutes a powerful instrument to model different kind of systems and the frequency at which data can be handled is not limited by other application components, except for the obvious computation time. It offers also the possibility to integrate the FlightGear engine to visualize the state of the simulation, thus satisfying all the demands to take the place of a dedicated flight simulator. Then other reasons for this choice are

- The software is well documented
- It allows fast development of new systems thanks to the application tools
- The software environment allows easy debugging of the developed applications
- The user friendly interface allows an easy maintenance and updating of the code/models
- The application is mainteined and updated thus it is realiable

The data exchange with this block is made through UDP protocol. The inputs to the block consist in the control signals to the motors evaluated by the autopilot controller and routed by the Data Governor application. The outputs of the simulation block are the simulated sensors data, which will be routed by the Data Governor application back to the autopilot board, and the vehicle navigation state for visual representation using FlightGear simulator.

2.3 Autopilot

The autopilot board is the flight computer of the aerial system (Figure 2.3). It has the function of estimating the navigation state, perform waypoints guidance and flight control. Once the state of the vehicle is estimated, using the information provided by onboard sensors, it translates the reference inputs, which can be manual human input or waypoints information, to actuator commands. The autopilot board chosen for this work is the PIXHAWK.

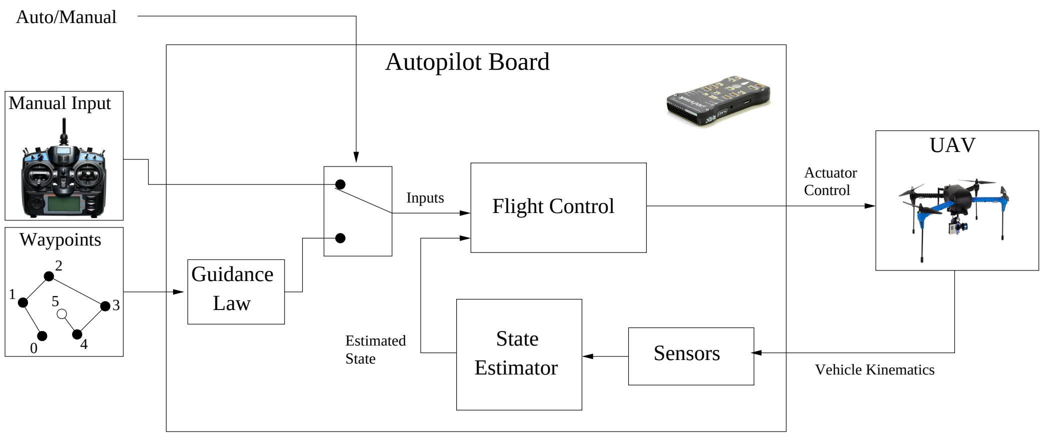


Figure 2.3: Scheme of the basic elements in a flight control system

It is a high-performance autopilot module suitable (Figure 2.4) for drones and other autonomous vehicles capable of running lightweight operating systems.

Autopilot Hardware

- Processor
  - 32bit STM32F427 Cortex M4 core with FPU
  - 168 MHz
  - 256 KB RAM
  - 2 MB Flash
  - 32 bit STM32F103 failsafe co-processor
- Sensors
  - ST Micro L3GD20H 16 bit gyroscope
  - ST Micro LSM303D 14 bit accelerometer / magnetometer

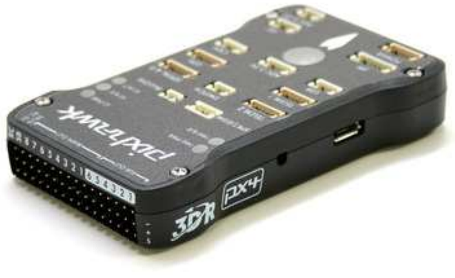


Figure 2.4: PX4 Autopilot Board

  - Invensense MPU 6000 3-axis accelerometer/gyroscope
  - MEAS MS5611 barometer

- Interfaces
  - 5x UART (serial ports), one high-power capable, 2x with HW flow control
  - 2x CAN (one with internal 3.3V transceiver, one on expansion connector)
  - Futaba S.BUS compatible input and output
  - PPM sum signal input
  - RSSI (PWM or voltage) input
  - I2C
  - SPI
  - 3.3 and 6.6V ADC inputs
  - Internal microUSB port and external microUSB port extension

2.3.1 Autopilot Operating System

The Pixhawk autopilot module runs *NUTTX*, a very efficient real-time operating system (RTOS), which provides a POSIX-style environment. The

Figure 2.5: Nuttx Logo

software can be modified and updated with an USB bootloader. The OS has been released under the permissive BSD license.

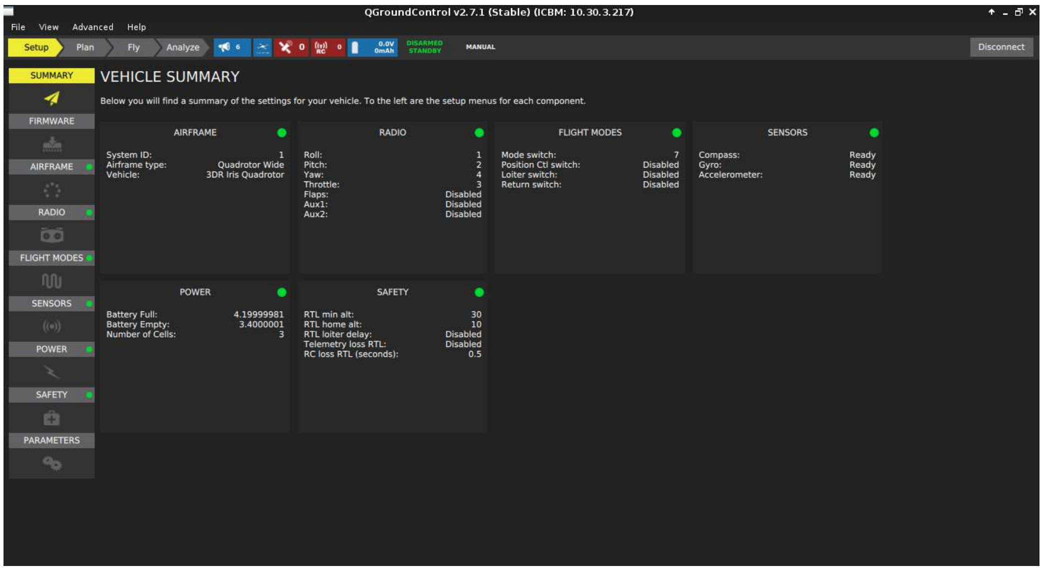
On top to the *NuttX* there are the Middleware and the Application layers. The former provides device drivers and a micro object request broker for asynchronous communication between the individual tasks running on the autopilot. The latter provides the flight control functionalities: navigation, guidance and control. The PX4 autopilot board support also other sets of application to accomplish the flight, such as the *APM Flight Stack*. For this work the *PX4 Flight Stack*, provided by the PX4 developer team, has been chosen.

2.4 Ground Station

In this work the *QGroundControl* (Figure 4.1) application has been chosen. The program is based on the Ground Station developed by the *Pixhawk Project*([8]) and now it is maintained by the community.

The ground station is a necessary component of the system because it constitutes the user interface with the vehicle. Through this application it is possible to check the configuration of the AUV (Figure 2.6), calibrate the sensors, modify the Flight Control Stack parameter (Figure 2.7) and to assign command to the vehicle. Waypoints for a mission can be assigned with a simple drag and drop on a map (Figure 2.8) and loaded on the autopilot memory. Telemetry data describing the state of the vehicle and sensors readings sent from the vehicle can be visualized in real time (Figure 2.9) and saved for post processing.

As is the Ground Station supports multiple autopilot boards like Pixhawk autopilot, PX4 autopilot, pxIMU, ArduPilotMega, SLUGS, MatrixPilot/UAVDevBoard and many more. The application can run on Linux, Mac or Windows, thus it does not force to adopt a particular system.

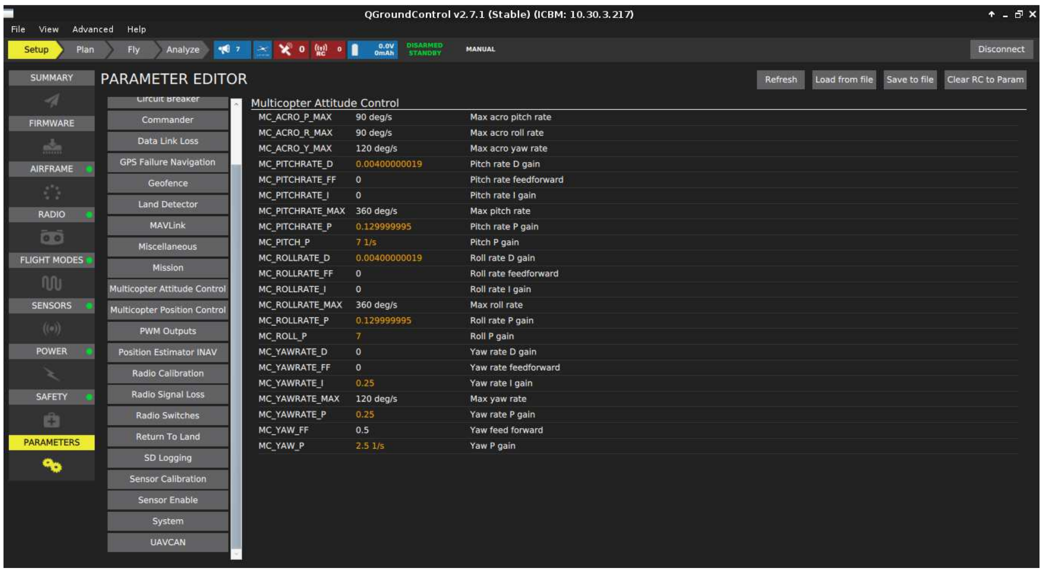


VEHICLE SUMMARY

Below you will find a summary of the settings for your vehicle. To the left are the setup menus for each component.

```
AIRFRAME
System ID:                         1
Airframe type:       Quadrotor Wide
Vehicle:          3DR Iris Quadrotor

RADIO
Roll:                              1
Pitch:                             2
Yaw:                               4
Throttle:                          3
Flaps:                      Disabled
Aux1:                       Disabled
Aux2:                       Disabled

FLIGHT MODES
Mode switch:                       7
Position Ctl switch:        Disabled
Loiter switch:              Disabled
Return switch:              Disabled

SENSORS
Compass:                       Ready
Gyro:                          Ready
Accelerometer:                 Ready

POWER
Battery Full:             4.19999981
Battery Empty:             3.4000001
Number of Cells:                   3

SAFETY
RTL min alt:                      30
RTL home alt:                     10
RTL loiter delay:           Disabled
Telemetry loss RTL:         Disabled
RC loss RTL (seconds):           0.5
```

Figure 2.6: Ground Station: Overview on the Drone State



PARAMETER EDITOR

Multicopter Attitude Control

MC_ACRO_P_MAX	90 deg/s	Max acro pitch rate
MC_ACRO_R_MAX	90 deg/s	Max acro roll rate
MC_ACRO_Y_MAX	120 deg/s	Max acro yaw rate
MC_PITCHRATE_D	0.00400000019	Pitch rate D gain
MC_PITCHRATE_FF	0	Pitch rate feedforward
MC_PITCHRATE_I	0	Pitch rate I gain
MC_PITCHRATE_MAX	360 deg/s	Max pitch rate
MC_PITCHRATE_P	0.129999995	Pitch rate P gain
MC_PITCH_P	7 1/s	Pitch P gain
MC_ROLLRATE_D	0.00400000019	Roll rate D gain
MC_ROLLRATE_FF	0	Roll rate feedforward
MC_ROLLRATE_I	0	Roll rate I gain
MC_ROLLRATE_MAX	360 deg/s	Max roll rate
MC_ROLLRATE_P	0.129999995	Roll rate P gain
MC_ROLL_P	7	Roll P gain
MC_YAWRATE_D	0	Yaw rate D gain
MC_YAWRATE_FF	0	Yaw rate feedforward
MC_YAWRATE_I	0.25	Yaw rate I gain
MC_YAWRATE_MAX	120 deg/s	Max yaw rate
MC_YAWRATE_P	0.25	Yaw rate P gain
MC_YAW_FF	0.5	Yaw feed forward
MC_YAW_P	2.5 1/s	Yaw P gain

Figure 2.7: Ground Station: Drone Parameter settings

The possibility to modify the source code, together with the modular design makes this application interesting for future developments. It would be possible to add new devices, define new protocols or simply modify the graphical interface with new widgets.

One of the features of this Ground Station consists in the implementation of the communication protocol for the data exchange with the AUV.

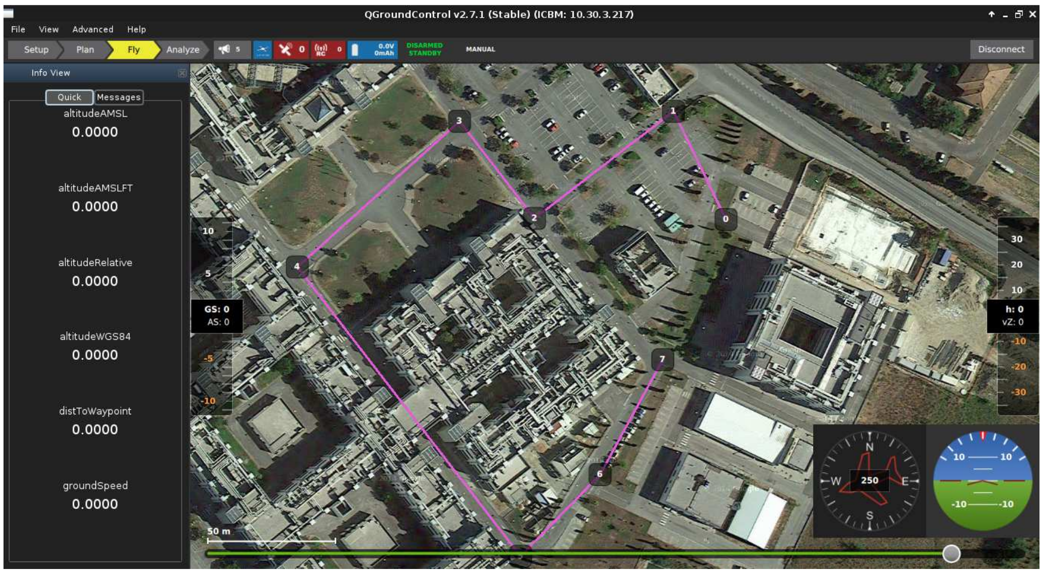


Figure 2.8: Ground Station: Map vision with waypoints

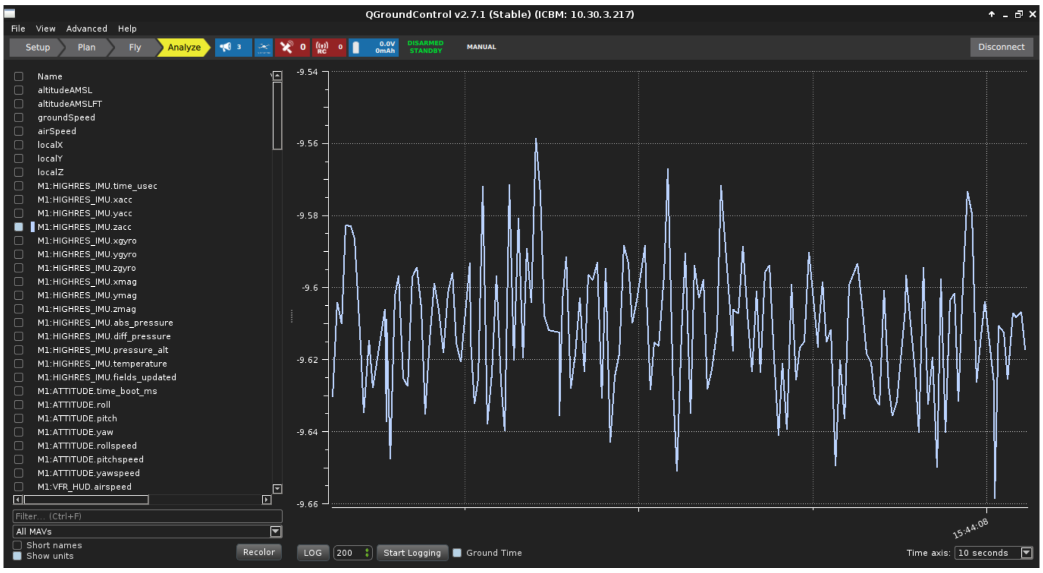


Figure 2.9: Ground Station: Plotting of received sensor data
1

This protocol is called *MAVLink* (Figure **??**), it is open-source, thus can be taylored for particular needs, and it is become common in several autopilot boards.

2.4.1 MAVLink Protocol

MAVLINK

MICRO AIR VEHICLE COMMUNICATION PROTOCOL

The *MAVLink* communication protocol has been adopted in this work because it is lightweight, so it does not constitute a possible bottleneck for the system. Moreover it simplifies the communication management thanks to the already developed and tested libraries which pack/unpack C-structs in/from mavlink messages, check the message content and detect lost messages. *MAVLink* is a header-only library, that is, it doesn't require to be compiled on the target architecture, but the inclusion of the header files sufficies. It is implemented in the Ground Control Station and it is embedded on the firmware of the autopilot board. The *Data Governor* use the MAVLink protocol to communicate with these two participants. Instead the communication with the simulation application is accomplished sending raw data (Figure 2.10).

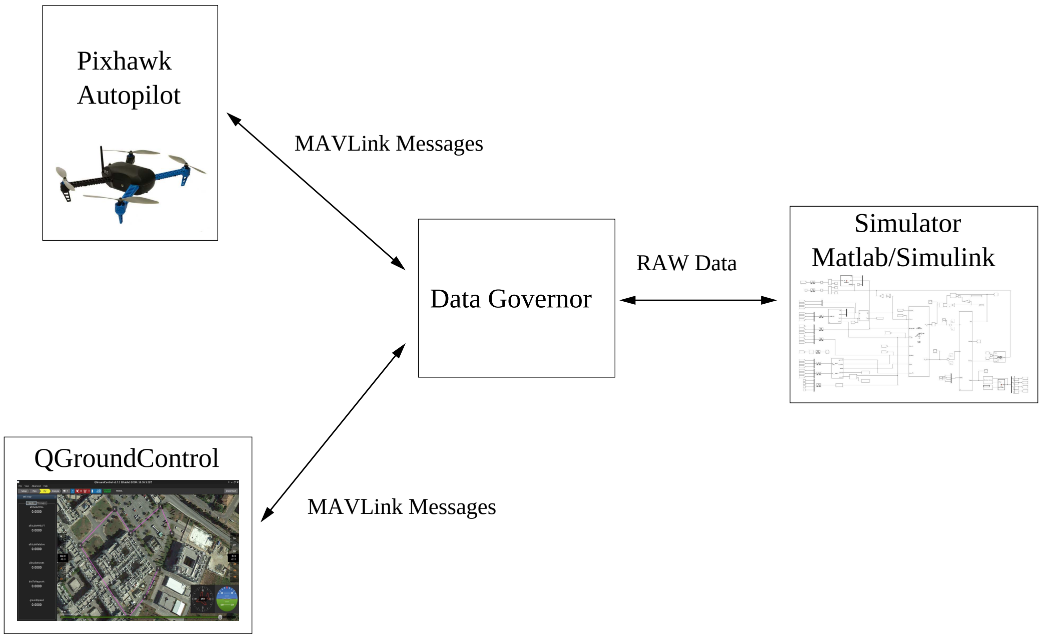


Figure 2.10: Communication Between Applications

MAVLink Features MAVLink supports fixed-size integer data types, IEEE 754 single precision floating point numbers, arrays of these data types. The list of the supported data types is constituted by:

- char - Characters / strings
- uint8_t - Unsigned 8 bit
- int8_t - Signed 8 bit
- uint16_t - Unsigned 16 bit
- int16_t - Signed 16 bit
- uint32_t - Unsigned 32 bit
- int32_t - Signed 32 bit
- uint64_t - Unsigned 64 bit
- int64_t - Signed 64 bit
- float - IEEE 754 single precision floating point number
- double - IEEE 754 double precision floating point number

MAVLink Performances The developers of the this protocol have tuned it to achieve good transmission performances while guaranteeing errors detection. The final result is a protocol with only 8 bytes of overhead and a payload capacity of 256 bytes. Some example of messages transmission performances, using different hardware, are reported in the following table.

Mavlink Transmission Performances Examples

Link Speed	Hardware	Update Rate	Payload
230400 baud	UART	250 Hz	64 bytes
115200 baud	XBeePro 2.4 GHz	50 Hz	224 bytes
115200 baud	XBeePro 2.4 GHz	100 Hz	109 bytes
57600 baud	XBeePro 2.4 GHz	100 Hz	51 bytes
9600 baud	XBeePro XSC 900	50 Hz	13 bytes
9600 baud	XBeePro XSC 900	20 Hz	42 bytes

Considering that the UART speed can be set up to 921600 baud it has been verified that it is possible to sent the required data for the simulation. Precisely it is necessary to receive 192616 bits/sec from the autopilot board and to send 232000 bits/sec to the autopilot board.

MAVLink Packet Structure The structure of the MAVLink packet(Figure 2.11) is inspired by the *CAN* and *SAE AS-4* standards.

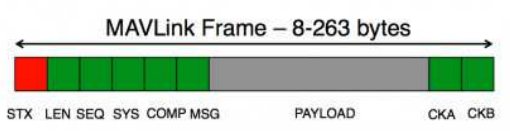


Figure 2.11: Structure of the mavlink packet

Mavlink Packet Structure

Byte Index	Content	Value	Explanation
0	Packet start Sign	0xFE	Indicates the start of a new packet.
1	Payload length	0-255	Indicates length of the following payload.
2	Packet sequence	0-255	Each component counts up his send sequence. Allows to detect packet loss.
3	System ID	1-255	ID of the SENDING system. Allows to differentiate different MAVs on the same network.
4	Component ID	0-255	ID of the SENDING component. Allows to differentiate different components of the same system, e.g. the IMU and the autopilot.
5	Message ID	0-255	ID of the message - the id defines what the payload means and how it should be correctly decoded.
6 to (n+6)	Data	0-255 bytes	Data of the message, depends on the message id.
(n+7) to (n+8)	Checksum (low byte, high byte)	ITU X.25/SAE AS-4 hash, excluding packet start sign.	

Chapter 3

Simulation Framework

3.1 Simulation Framework and Aims

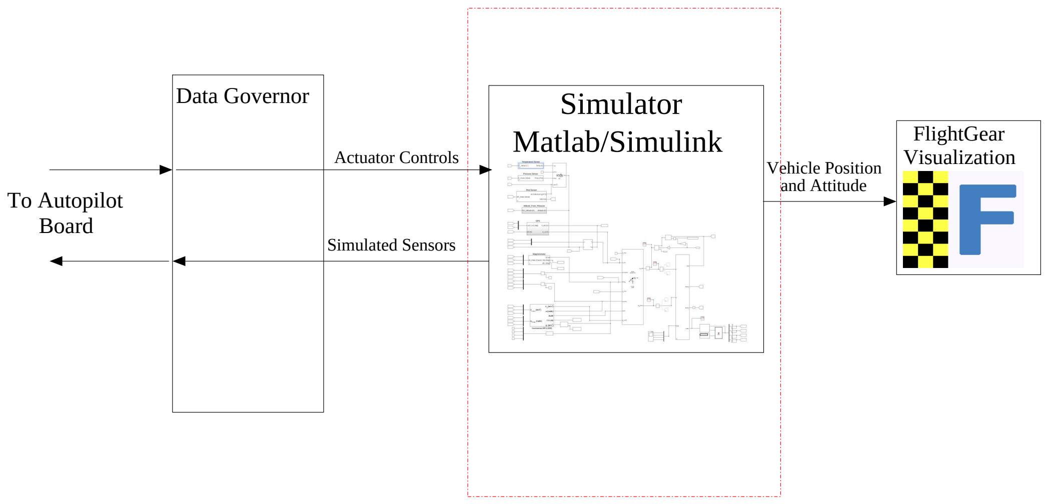


Figure 3.1: Simulation Block

The simulation part aims to provide realistic sensor data starting from the actuator commands produced by the autopilot board. These sensor data are then fed back to the autopilot board to close the simulation loop.

The simulation block (Figure 3.1) can be conceptually subdivided into 3 main parts:

1. AUV Dynamics Model
   It describes the dynamics of the vehicle and allows to compute the trajectory given the inputs to the system, that is the actuators actuator

commands. In this part it is included also the dynamics of the brushless motors which are the actuators of the UAV.

2. Sensors Model
   It gives the outputs of the sensors given the state of the vehicle and external inputs.

3. Environment Model
   This model describes the variables of the environment such as gravitational acceleration, earth magnetic field, temperature, density of the air etc.

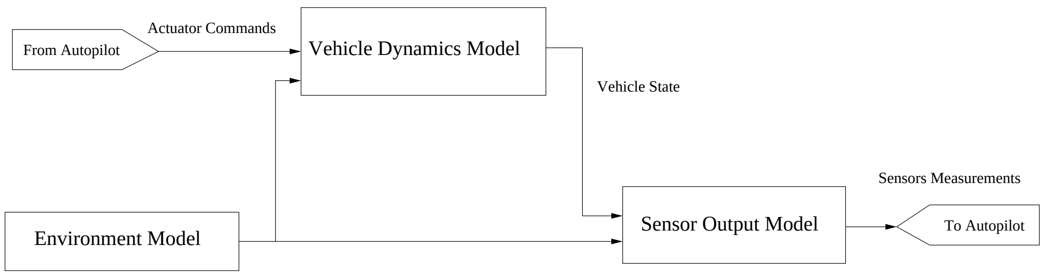


Figure 3.2: Simulation Framework

The simulation flow is represented in Figure 3.2. The step accomplished by the simulator are then

1. Use the AUV Dynamics Model to propagate the state

2. Compute the Output of the Sensors

3.2 Conventions

3.2.1 Frames of Reference

As said before the purpose of the simulator is to propagate and track the flight of the UAV given forces and moments acting on it. When dealing with moving objects in a 3D space it becomes necessary to specify the "point of view", that is the reference frame with respect to which the values are expressed. There are several frames which are commonly used in this kind of application (Figure 3.3):

1. Earth Centered Inertial Frame (ECI, $\mathcal{I}$):
   This frame has the origin at the center of the earth with the z_i axis

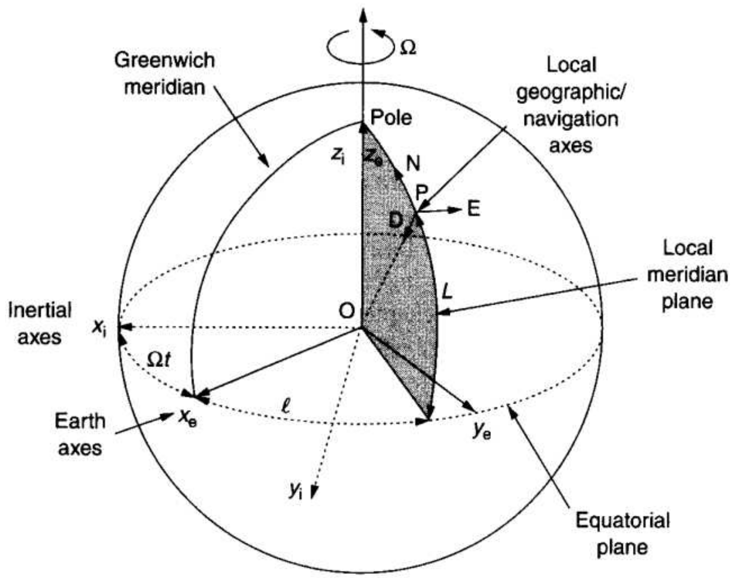


Figure 3.3: Reference Frames commonly used for navigation purposes

parallel to the rotation axis of the earth and x_i,y_i axes located in the equatorial plane. It is fixed to an inertial reference.

2. Earth Centered Earth Fixed Frame (ECEF, $\mathcal{E}$):
   This frame has the origin at the center of the earth and it is fixed to it. The z_e axis of the frame is parallel to and aligned with the rotation axis of the earth. In the equatorial plane the x_e axis locates the Greenwich meridian. The y_e axis completes the right hand system.

3. North-East-Down Frame (NED, $\mathcal{N}$):
   This frame has the origin at the vehicle center of mass with the x axis, called N, pointing North; y axis, called E, pointing East; z axis, called D, pointing Down.

4. Body Frame ($\mathcal{B}$)(Figure 3.4):
   This frame is rigidly attached to the vehicle's body. It has the x_b axis pointing forwards out the nose of the aircraft, y_b axis pointing out the right side of the aircraft and z_b axis consequently pointing down.

The previous frames are rotating one with respect to another and, for deriving the various equations, it is necessary to consider this fact. The angular speed that are usually considered are

- $\omega_{n/b}$: Angular velocity vector of the body frame with respect to the navigation frame

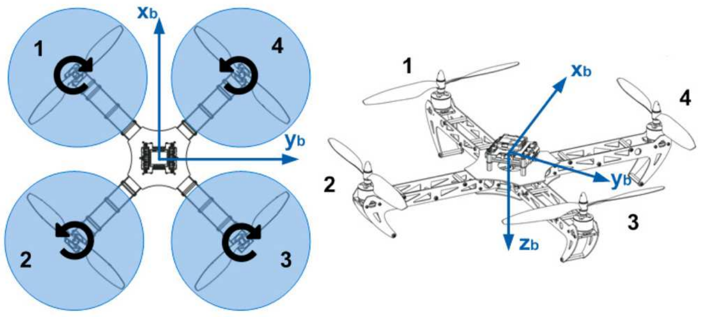


Figure 3.4: Reference Frame attached to the body

- $\omega_{e/n}$: Angular velocity vector of the navigation frame with respect to the ECEF frame
- $\omega_{i/e}$: Angular velocity vector of the ECEF frame with respect to the ECI frame, that is the earth rotation angular velocity

In describing the dynamics of a rigid body in space everything is referred to an inertial reference. From the previous list of common frames this is the case of the ECI. Theoretically it is necessary to take into account all the contributions of the varius motions in the chain of compositions from the inertial frame towards the body frame. Whereas for particular simulations where the flight covers long distance, with long flight time, it is necessary to consider every relative motion between frames to obtain an accurate simulation, nevertheless in applications where the flight involves small distances and short flight time the problem can be simplified. This is what has been done in this work. Actually the equations of motion for the UAV have been obtained considering the navigation frame NED as inertial. Indeed, flying locally, the position of the NED frame can be considered fixed with respect to the $ECEF$ frame ($\omega_{e/n} \approx 0$) and the effect of the earth rotation can be neglected for short time flights.

3.2.2 Rotations

To accomplish the simulation it is often necessary to switch between a reference frame to another. For example the computation of torque, forces and their effects is easier when accomplished considering the quantities espressed in body frame. However, at the end, the velocity of the vehicle, the force and the moments must be espressed with respect to the inertial reference frame to carry on the integration step. This passages are made using rotation matrices $C \in \Re^{3\times3}$. These matrices are function of the parameters describing the orientation between frames. In this text the convention adopted to express a rotation which brings the coordinates from frame $\mathcal{N}$ to frame $\mathcal{B}$, or equivalently, that rotates the $\mathcal{N}$ frame on the $\mathcal{B}$ frame is

$$C_n^b = \text{Rotation Matrix } (\mathcal{N}) \text{ to } (\mathcal{B}). \tag{3.1}$$

Whereas one matrix contains 9 elements only 3 parameters are necessary to specify a rotation in space. This fact is due to the orthogonality property of rotation matrices, that is

$$CC^T = I, \tag{3.2}$$

where C^T is the transpose matrix of C and I is the identity matrix. Minimal representations with 3 parameters can be obtained by using sets of 3 angles, called also "euler angles". Each angle is associated with a single rotation about one of the coordinate axes X,Y,Z. A generic rotation in space can be thought as a composition of 3 sequential single rotations. The building block of a general rotation are then the rotation matrices with respect to the three axes of a reference frame X,Y,Z.

- Rotation around X axis:

$$C_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos(\theta) & sin(\theta) \\ 0 & -sin(\theta) & cos(\theta) \end{bmatrix} \tag{3.3}$$

- Rotation around Y axis:

$$C_y(\theta) = \begin{bmatrix} cos(\theta) & 0 & -sin(\theta) \\ 0 & 1 & 0 \\ sin(\theta) & 0 & cos(\theta) \end{bmatrix} \tag{3.4}$$

- Rotation around Z axis:

$$C_z(\theta) = \begin{bmatrix} cos(\theta) & sin(\theta) & 0 \\ -sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.5}$$

There are different way of composing the rotation, indeed the only requirement is to guardantee that two successive rotation are not made about parallel axes. To give an example, a common set of angles used in the Aerospace field is the sequence *Roll, Pitch, Yaw*, corresponding respectively to rotations about the X,Y,Z axes of the body frame of a vehicle. The general rotation from frame $\mathcal{N}$ to frame $\mathcal{B}$ is then given by

$$C_n^b(\phi, \theta, \psi) = C_x(\phi)C_y(\theta)C_z(\psi). \tag{3.6}$$

Euler angles are not the only one method to represent rotations in space. There are other methods such as axis/angle and quaternions.

The axis/angle (r, θ) representation parametrizes a rotation using a versor $r \in \Re^3$ to express the axis along with the rotation is carried out and an angle θ to express the rotation angle. Quaternions (q) are another way of representing rotations. They are vectors of 4 elements which can be thought to be composed by two parts

$$q = \begin{bmatrix} q_0 \\ q_e \end{bmatrix}, \tag{3.7}$$

where $q_0 \in \Re$ is the scalar part of the quaternion vector and $q_e = [q_1, q_2, q_3]^T \in \Re^3$ is the vector part of the quaternion vector. If we want to link the geometry of rotation with the vector it is interesting to consider the connection with the axis/angle (r, θ) representation. Indeed it is found to be

$$q = \begin{bmatrix} q_0 \\ q_e \end{bmatrix} = \begin{bmatrix} cos(\theta/2) \\ rsin(\theta/2) \end{bmatrix}. \tag{3.8}$$

It is possible to notice that the vectorial part (q_e) of the quaternion is related to the axis around with the rotation is performed, while the amount of rotation is codified by the $sin(\theta), cos(\theta)$ scaling factors.

Usually the euler angles representation is used when it is necessary to interact with the user because they are easy to visualize and transmit quite direcly the geometrical information about the rotation. For a computational point of view quaternions are more efficient. Indeed in this simulation framework the integration step is carried out using the quaternion representation of orientation. The rotation matrix expressed with quaternions has the form

$$C_n^b(q) = \begin{bmatrix} q_0^2 + q_1^2 - q_2^2 - q_3^2 & 2q_1q_2 + 2q_0q_3 & 2q_1q_3 - 2q_0q_2 \\ 2q_1q_2 - 2q_0q_3 & q_0^2 - q_1^2 + q_2^2 - q_3^2 & 2q_2q_3 + 2q_0q_1 \\ 2q_1q_3 + 2q_0q_2 & 2q_2q_3 - 2q_0q_1 & q_0^2 - q_1^2 - q_2^2 + q_3^2 \end{bmatrix}. \tag{3.9}$$

In the simulink model these rotation matrices are labeled "DCM", which stands for "Direction Cosines Matrix" and they perform the rotation from the Inertial Frame $\mathcal{N}$ to the Body Frame $\mathcal{B}$.

In the following sections the parts composing the simulation framework are described in details.

3.3 UAV Dynamics Model

The UAV considered for this work is a quadrotor. In the literature concerning quadrotors there are several approaches to the vehicle dynamics modeling.

Some works are oriented towards a deep analysis and identification of the model like in [13], [15] and [16]. An accurate identification of the model is necessary to design advanced control algorithms, particularly when the vehicle on which they are going to be applied is definitive and won't be modified for a long period. Since the aim of the this work is not focused on a particular vehicle with a particular configuration, it has been chosen to deal with a simple geometrical model in order to simplify the development of the dynamical equations. The geometry of the quadrotor can be schematized as a cross-shaped frame, formed by 4 arms of equal length l, which old 4 motors. The center of mass of the vehicle, where all the mass M is lumped, is considered located in the geometrical center of the structure. The rotational inertia properties of the vehicle are described by the inertia tensor J expressed in body frame coordinates.

3.3.1 Actuators

For this kind of vehicles the actuation, thus the generation of forces and moments, is made varying the speed of the rotors ω_{m_i}. Each rotor produces a thrust T_i and a torque Q_i, whose combination generates the main thrust, the yaw torque, the pitch torque and the roll torque acting on the quadrotor. for the thrust calculation it has been used a non linear expression where the force depends quadratically on the angular speed of the rotor. Precisely the expression for the generated thrust and torque is given by

$$T_i = C_t \rho A r^2 \omega_{m_i}^2 \tag{3.10}$$
$$Q_i = C_q \rho A r^3 \omega_{m_i}^2 \tag{3.11}$$

Where

- C_t : Thrust Coefficient
  This parameter is identified with empirical experiments or using references. In this work an indicative reference value has been taken from the work of Pounds and Co. [15].

- C_q : Torque Coefficient
  Also for this parameter it is valid the identification method used for the C_t and an indicative value has been retrieved from the same work cited above.

- ρ : Air density
  Depends on the altitude and there are look-at-table or model to retrieve its value

- A : Rotor disk area
- r : Rotor disk radius
- ω_{m_i}: Angular velocity of the Rotor

Motors Dynamics The rotors are moved by DC motors. It was not the aim of this work to produce a precise dynamical model of the vehicle, thus they have been represented by a generic second order dynamical system it.

3.3.2 Equations of Motion

Having found the expression of the forces and torques acting on the structure, it is possible to write the equations of the vehicle dynamics. The expression for the linear velocity $\dot{p}^n$ and linear acceleration $\dot{v}^n$ is expressed in navigation frame. The expression of the angular acceleration $\dot{\omega}^b_{n/b}$ is given in body frame. The computation of the rotational dynamics in body frame coordinates is more computationally efficient since the inertia matrix J is constant. In the end the expressions are:

$$\dot{p}^n = v^n \tag{3.12}$$

$$\dot{v}^n = \frac{1}{M}(C^n_b F^b) - C^n_b(\omega^b_{n/b} \times v^b) + G^n \tag{3.13}$$

$$\dot{\omega}^b_{n/b} = J^{-1}\left[-\omega^b_{n/b} \times J\omega^b_{n/b} + M^b\right] \tag{3.14}$$

where F^b is the thrust generated by all the rotors expressed in Body Frame

$$F^b = \sum_{i=1}^{4} \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} T_i, \tag{3.15}$$

C^b_n is the rotation matrix which maps the coordinates from navigation frame to body frame, G^n is the gravity vector expressed in navigation frame and M^b is the resultants of the torques in body frame, which for the particular quadrotor configuration is given by

$$M^b = \begin{bmatrix} l\frac{\sqrt{2}}{2}(T_1 - T_2 - T_3 + T_4) \\ l\frac{\sqrt{2}}{2}(T_1 + T_2 - T_3 - T_4) \\ -Q_1 + Q_2 - Q_3 + Q_4 \end{bmatrix}. \tag{3.16}$$

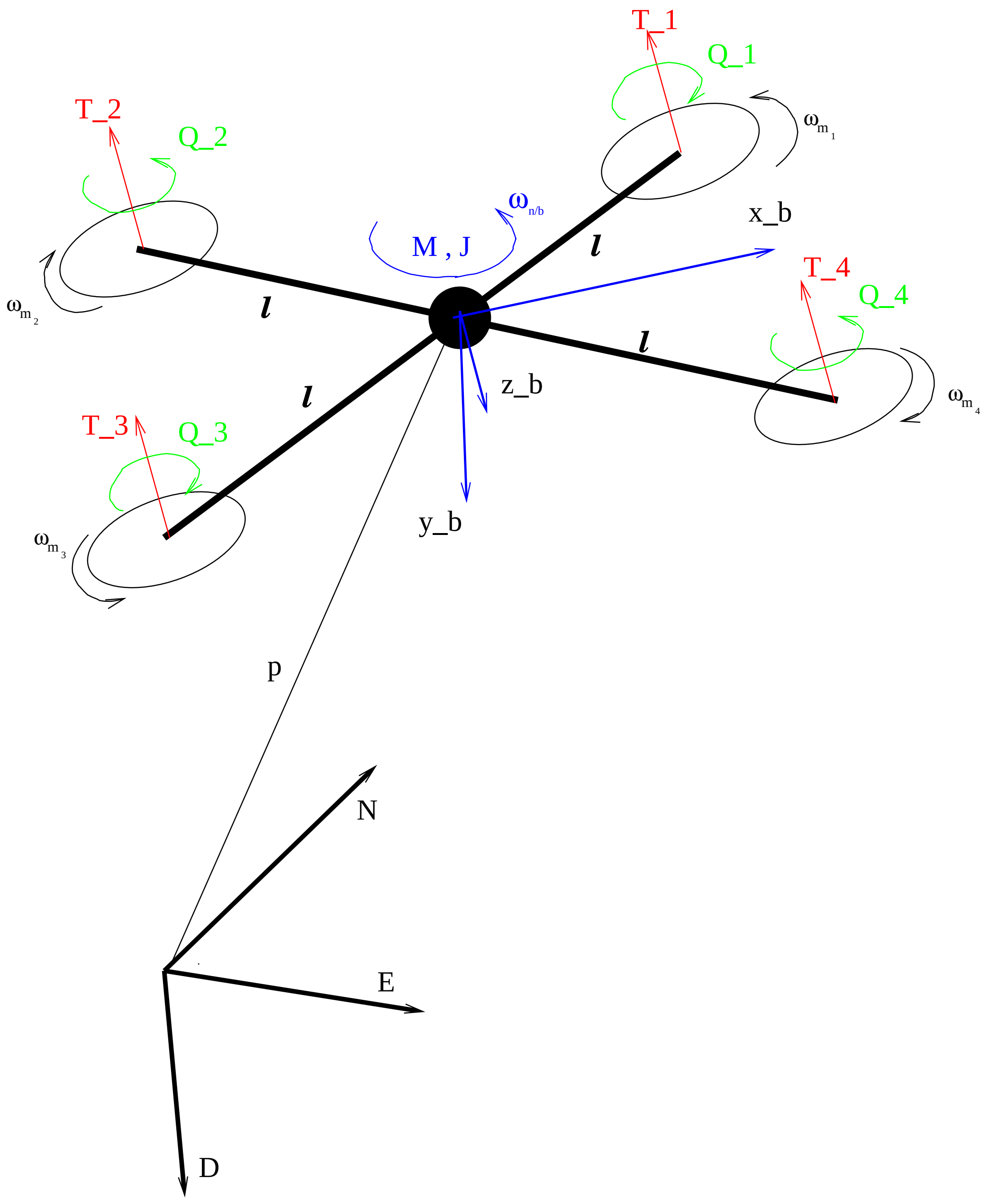


Figure 3.5: Schematical representation of the quadrotor and the forces/torques acting on it

3.3.3 Simulink Blocks

In *Simulink* the dynamics of the vehicle has been implemented with three blocks:

1. Dynamics of the motors
2. Forces/Moments computation
3. Integration of the forces/moments

The first block (Figure 3.6) has the function of taking into account the rotative dynamics of the motor. In this work no identification procedures have been run to precisely evalute the dynamics of the motors. The motors are thus represented by general second order dynamical system. The input to the motors block is the voltage in V from the battery and the vector of 4 pwd signals mapped in the interval [0, 1]. The output of the block are the 4 rotational speed in rad/s.

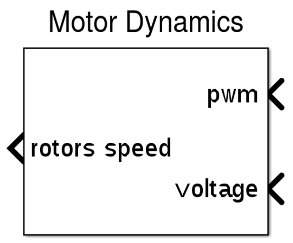


Figure 3.6: Simulink Motors describing the motor dynamics

The second block (Figure 3.7) has the function of computing the resultant force and torque acting on the UAV from the single contributions of the motors. The inputs to the block are defined as

- DCM: Rotation Matrix from the navigatio frame to the body frame
- $Velocity$: Velocity in m/s of the AUV expressed in body coordinates
- $Density$: Density of the air at the current altitude in Kg/m^3
- $Rotors$: Angular speed of the 4 propellers rad/s

The output of this block are given by

- F: Force resultant in body frame N
- Q: Torque resultant in body frame Nm
- $Thrusts$: Force exerted by each single propeller N

The third block (Figure 3.8) perform the integration of the forces/torques to obtain the state of the rigid body. The inputs to the block are

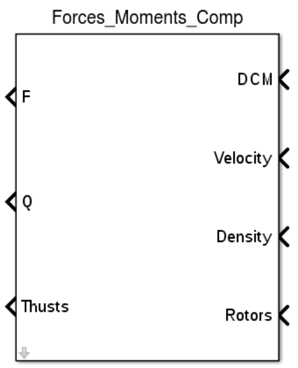


Figure 3.7: Simulink Block for the forces/torques computation

- F_{xyz}: Resultant of the forces in N applied to the rigid body and expressed in body frame coordinates
- M_{xyz}: Resultant of the torques in Nm applied to the rigid body and expressed in body frame coordinates

The outputs of the block consists in

- V_e: Velocity in m/s of the AUV in navigation frame
- X_e: Position in m of the AUV in navigation frame
- $\phi \quad \theta \quad \psi$: Roll, Pitch, Yaw angles in rad parametrizing the attitude of the AUV with respect to the navigation frame
- DCM_{be}: Direct Cosines Matrix, that is rotation matrix from the navigation frame to the body frame
- V_b: Velocity in m/s of the AUV in body frame
- ω_b: Angular velocity in rad/s of the AUV in body frame
- $d\omega_b/dt$: Angular acceleration in rad/s^2 of the AUV in body frame
- A_b: Acceleration in m/s^2 of the AUV in body frame

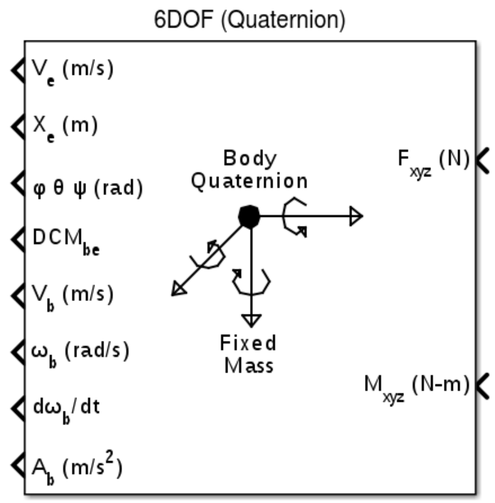


Figure 3.8: Simulink Block for the Dynamics integration

3.4 Sensors' Model

The *Simulink* programming environment offers plenty of functionalities to develop sensors model. It includes also several libraries with already developed sensor blocks. In order to run a functional simulator the sensor data used by the algorithms onboard the autopilot have been simulated. The data necessary to build the required HIL_SENSOR and HIL_GPS messages are reported in the following tables:

Data necessary to build the HIL SENSOR message

Field Name	Type	Description
xacc	float	X acceleration (m/s^2)
yacc	float	Y acceleration (m/s^2)
zacc	float	Z acceleration (m/s^2)
xgyro	float	Angular speed around X axis body frame (rad/s)
ygyro	float	Angular speed around Y axis body frame (rad/s)
zgyro	float	Angular speed around Z axis body frame (rad/s)
xmag	float	X Magnetic field ($Gauss$)
ymag	float	Y Magnetic field ($Gauss$)
zmag	float	Z Magnetic field ($Gauss$)
abs_pressure	float	Absolute pressure ($mBar$)
diff_pressure	float	Differential pressure ($mBar$)
pressure_alt	float	Altitude calculated from pressure (m)
temperature	float	Temperature (C^o)

Data necessary to build the HIL GPS message

Field Name	Type	Description
lat	int32_t	Latitude (WGS84), in degrees * 1E7
lon	int32_t	Longitude (WGS84), in degrees * 1E7
alt	int32_t	Altitude (AMSL), in meters * 1000 (positive for up)
vel	uint16_t	GPS ground speed (m/s * 100)
vn	int16_t	GPS velocity in cm/s in NORTH direction in earth-fixed NED frame
ve	int16_t	GPS velocity in cm/s in EAST direction in earth-fixed NED frame
vd	int16_t	GPS velocity in cm/s in DOWN direction in earth-fixed NED frame
cog	uint16_t	Course over ground in degrees * 100

The list of sensors that are necessary to produce the previous information are

- Accelerometers
- Gyroscopes
- Magnetometers
- Barometer
- Airspeed Sensor (for fixed wing UAV)
- Temperature Sensor

IMU model The Inertial Measurement Unit contains accelerometers and gyros. It returns the acceleration and angular velocity of the body on which is attached, with respect to the inertial frame. The accelerometer and gyros model has been provided by the *IMU* simulink block (Figure 3.9) in the *Areospace Blockset* library and allows to simulate the behavior of a 3-axis Inertial Measurement Unit considering sensors biases, measurement noise, centripetal effects, scaling factors and misalignment. The position of the sensor and of the center of mass of the vehicle are specified using the *Structural Frame*. This frame is a common manufacturer's frame of reference and is used to define points on aircrafts. In the structural frame the x axis increases from the nose towards the tail, the y axis increases from the fuselage out towards the right and the z axis is positive upwards.

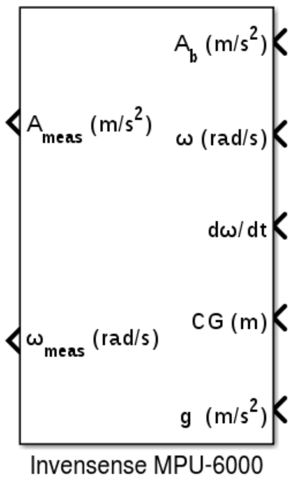


Figure 3.9: Simulink IMU block

There is a broad range of parameters which can be provided to the simulink block to simulate the features of real sensors. For the purpose of this work only a subset of the all possibilities has been specified taking into consideration the features of the real IMU onboard the autopilot board (*Inversense MPU-6000*):

- Sensor Biases A_{bias}, ω_{bias}
- Sensor Saturations
- Measurements Noise A_{noise}, ω_{noise}
- Position of the IMU in structural frame P_{imu}

The block inputs consist in

- A_b: Acceleration of the vehicle expressed in body frame in
- ω: Angular Velocity of the vehicle expressed in body frame
- $\dot{\omega}$: Angular Acceleration of the vehicle expressed in body frame
- CG: Position of the center of mass of the vehicle in structural frame
- g: Gravity expressed in body frame

The block ouptuts consist in

- A_{meas}: Measured Acceleration in body frame

- ω_{meas}: Measured Angular Velocity in body frame

For a rigid body, which is moving in space effected by a gravitational acceleration g, having acceleration A_b, angular velocity ω and angular acceleration $\dot{\omega}$ the equations ruling the IMU model will produce the following ouputs:

$$A_{meas} = A_{SFCC}\left(A_b + \omega \times (\omega \times d) + \dot{\omega} \times d - g\right) + A_{bias} + A_{noise} \quad (3.17)$$

$$\omega_{meas} = B_{SFCC}\omega + \omega_{bias} + Gs \times \omega_{gsens} + \omega_{noise}, \quad (3.18)$$

where the 3-by-3 matrices A_{SFCC}, B_{FSCC} contain the scaling factors on the diagonal and misalignement terms in the nondiagonal respectively for the accelerometers and the gyros. In the previous expressions d is the displacement between the center of mass of the rigid body and the position of the IMU and it is defined as

$$d = \begin{bmatrix} d_x \\ d_y \\ d_z \end{bmatrix} = \begin{bmatrix} -(x_{acc} - x_{CG}) \\ y_{acc} - y_{CG} \\ -(z_{acc} - z_{CG}) \end{bmatrix}. \quad (3.19)$$

The minus signs are due to the way the positions of the CG and of the sensor were given (structural reference frame).

Magnetometer Sensor The magnetometer is a device used to sense the earth magnetic field. The direction of the Earth Magnetic field at any point on the Earth is defined in terms of its orientation with respect to true north, known as the angle of “Magnetic Declination” and its angle with respect to the horizontal, the angle of “dip” (Figure 3.10). This vector in navigation coordinates is expressed as

$$H = \begin{bmatrix} H_x \\ H_y \\ H_z \end{bmatrix} = H_0 \begin{bmatrix} \cos\delta\cos\gamma \\ \cos\delta\sin\gamma \\ \sin\delta \end{bmatrix}, \quad (3.20)$$

where H_0 is the intensity of the earth magnetic field in that point. In the simulation the assumption of local displacement of the vehicle has been made, allowing to consider the vector constant.

The block inputs consist in

- *DCM*: Rotation matrix from navigation to body frame coordinates (C_n^b)
- *dip*: Angle of “dip”
- *declination*: Declination Angle

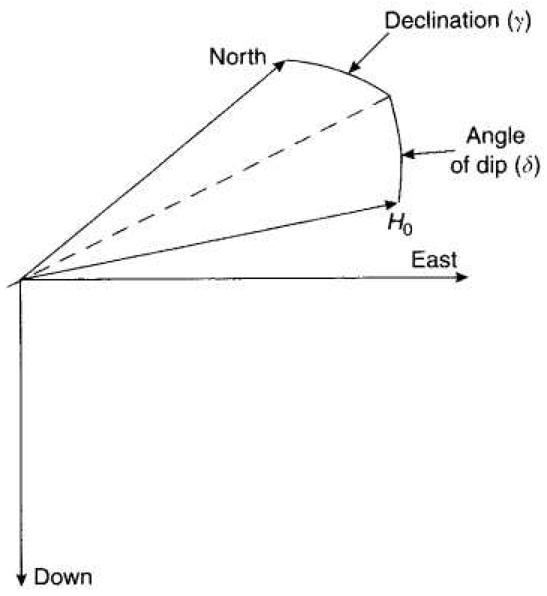


Figure 3.10: Earth Magnetic Vector components

The block ouptut consists in

- M_{meas}: Measured magnetic field in body frame

The parameter of the sensors describing measurement noises, biases, saturation can be specified through the block mask. The sensor output have been

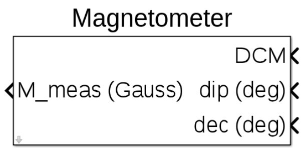


Figure 3.11: Simulink Magnetic Sensor block

computed rotating this vector from NED to Body and adding measurement noise and bias.

$$M_{meas} = C_n^b H + M_{bias} + M_{noise}, \tag{3.21}$$

where M_{noise} and M_{bias} are the respectively the measurement noises and the measurement bias.

Barometer Sensor This sensor should return the static pressure in milliBar. To produce the output of this sensor a *Simulink* block modeling the standard atmosphere is used (Figure 3.12). The ISA Atmosphere Model block implements the mathematical representation of the international standard atmosphere values for ambient temperature, pressure, density, and speed of sound for the input geopotential altitude. The input to the block (Figure

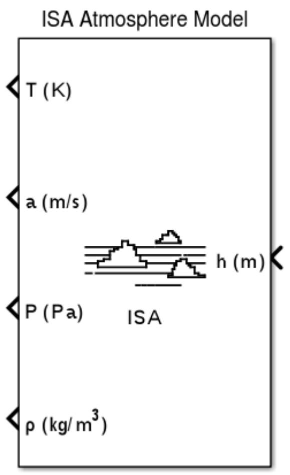


Figure 3.12: ISA Atmosphere Model Block

3.13) consists in the real pressure retrieved from the previously cited block. The ouput is the measured static pressure and has been modeled as

$$P_{meas} = P + P_{bias} + P_{noise}, \tag{3.22}$$

where P_{noise}, P_{bias} are the respectively the measurement noises and the measurement bias and the P is the real pressure value provided by the ISA model. The parameters for the sensor noise and bias can be specified through simulink interface. The output of the sensor is provided in mBar.

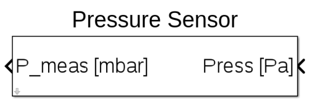


Figure 3.13: Simulink Pressure Sensor block

Airspeed Sensor The airspeed sensor is a pitot tube which measures the wind relative AUV axial speed throught the dynamic pressure readings. The simulation of this sensor consists in calculating the expected dynamic pressure related to the UAV speed. It is used with fixed wing vehicles but it has been included in the simulation framework for eventual future usages. The inputs to the block (Figure 3.14) consist in

- *AirDensity*: Air density at the current altitude in Kg/m^3
- Vel: Velocity of the vehicle in m/s

The block output consists in

- dP_{meas}: Differential Pressure in $mBar$,

and is evaluated as

$$dP_{meas} = \frac{1}{2}\rho||v||^2 + dP_{bias} + dP_{noise}. \tag{3.23}$$

The values dP_{bias}, dP_{noise} are the respectively the measurement noises and the measurement bias, ρ is the air density at that altitude and $||v||$ is the module of the vehicle's speed's. The output of this sensors is provided in mBar.

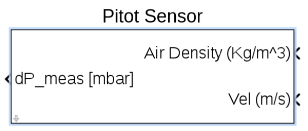


Figure 3.14: Simulink Pitot Sensor block

Temperature Sensor The temperature sensor block (Figure 3.15) simply takes the temperature T in Kelvin degrees of the atmosphere, provided by the ISA Atmosphere Model, converts it to Celsius degree and introduces measurement bias T_{bias} and noise T_{noise}.

$$T_{meas} = T + T_{bias} + T_{noise}, \tag{3.24}$$

The output of these blocks are then sent to the *Data Governor* application

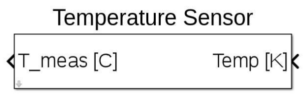


Figure 3.15: Simulink Temperature Sensor block

which encode the mavlink messages containing the *HIL_SENSOR* data.

GPS Sensor The GPS measurements on the autopilot board consists in the coordinates latitude/longitude/altitude (LLA). In matlab the sensor model has been obtained taking the local coordinates and translating them in Latitude/Longitude/Altitude usign the *Simulink* block “Flat Earth to LLA”. This block uses the WGS84 planet model to convert from local coordinates to angular coordinates. The block inputs consist in

- X_e: Coordinates in flat earth, that is the local displacement of the vehicle in the North, East, Down directions.
- h_{ref}: Reference Altitude, that is the altitude with respect to the sea level of the local earth surface.

The block ouptut consists in

- Lat_{meas}: GPS Measured Latitude
- Lon_{meas}: GPS Measured Longitude
- Alt_{meas}: GPS Measured Altitude

The parameter of the sensors describing measurement noises can be specified through the block mask. It hasn’t been introduced a measurement bias since the gps data is considered to be noisy but without long-term drift. The measurement noise is added to the local position of the AUV before the conversion in LLA coordinates. The output of this block are sent together

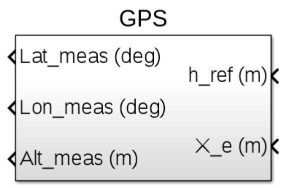


Figure 3.16: Simulink GPS Sensor block

with the rest of sensor data to the *Data Governor* where the mavlink packet *HIL_GPS* is encoded.

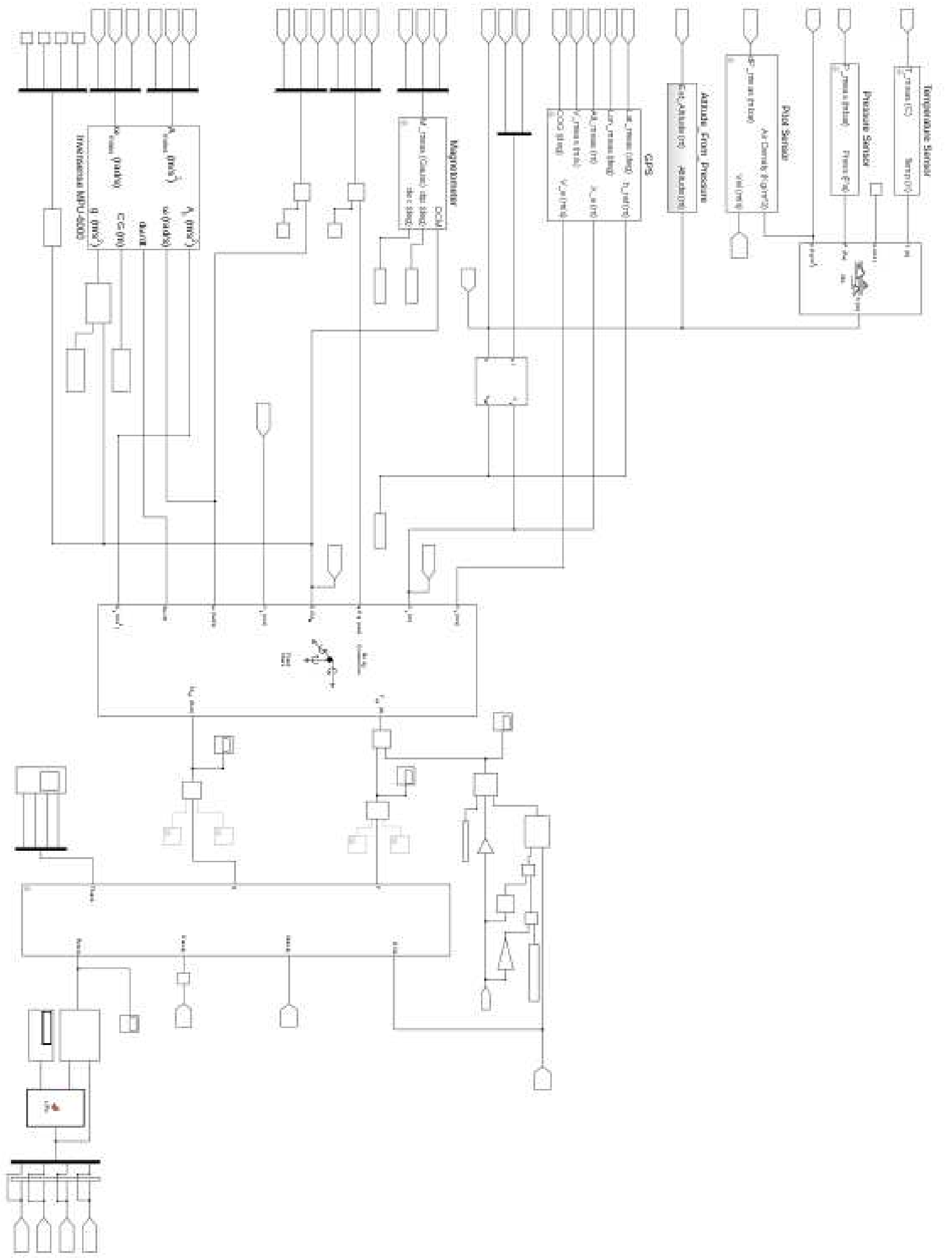


Figure 3.17: Simulink Complete Model

Chapter 4

Ground Station Application

In order to provide the functionalities of a Ground Station it is necessary to have an application which carries out the communication with the UAV. For this purpose QGroundControl (fig. 4.1) has been chosen. The fact of being open source allows to view the source code, feature which turns out to be useful when it is necessary to judge the performance of the software. This is

QGROUNDCONTROL
GROUND CONTROL STATION FOR SMALL AIR - LAND - WATER AUTONOMOUS UNMANNED SYSTEMS

Figure 4.1: Ground Station: QGroundControl Logo

an object-oriented C++/Qt application that permits to represent and control micro areal vehicles. As said in the introductory chapter, consitutes the user friendly interface with the UAV. Given this function the QGroundControl adheres to the model-view-controller (MVC) and ISO/OSI layer design patterns. This means that data, data manipulation and user interface representation are separated and that the access on hardware/communication links is abstracted from the application layer. In a MVC structure the components are classified as

- Model: Data structure / container representing a physical object (e.g. a UAV)
- View: User interface component visualizing the data of the model
- Controller: Class/functions to manipulate the model

The application manages the data flux with Link classes representing various communication means like Serial Interface or Ethernet or WiFi. The raw data is internally handled/parsed by the Protocol class and then fed into the model representing one unmanned system (UAV), which in the application is called UAS class. Because QGroundControl is a streaming/control centered application, model and controller are often combined in the same class. The structure is represented in fig. 4.2. The choice is justified by the need of

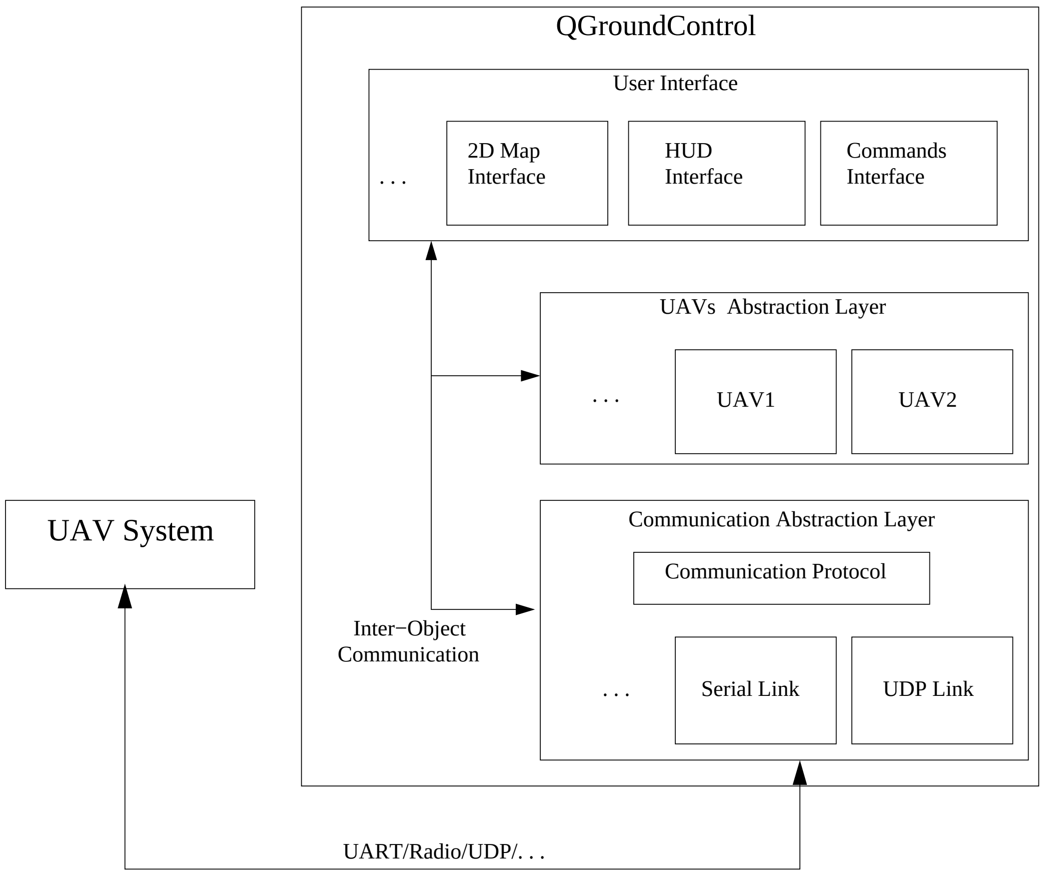


Figure 4.2: Application Structure

modularity, easiness of maintenance and update. Indeed the objectives of the application developers are summarized as:

- The stable and long term support of a standardized communication interface to the micro air vehicles
- A standardized interface of different user interface components
- Full separation of communications, protocol and user interface

- The long term support for Windows, Linux and Mac OS
- The long term openness for any type of micro air vehicle

4.1 Inter-Object Communication Approach

The software architectural pattern is tipical of user interfaces. This feature comprehends also how the communication between different components of the application is carried out and this has some implications when considering simulation requirements. When programming graphical user interfaces (GUIs) it is necessary to link actions of the user, such as clicking buttons, with routines. This connection can be made using pointers to functions. In this way it is possible to associate to an user action the address of the relative processing routine. This solution is called "callback". Qt libraries offer another method, precisely the *Signals & Slots* mechanism has been used.

4.1.1 Signals and Slots: Description

Signals and slots are used for communication between objects. The signals and slots mechanism is a central feature of Qt and probably it is the part that differs most from the features provided by other frameworks. It works like software interrupts between different objects: an event during the execution of code inside a class can trigger routines in other classes. The two components of this way of interacting are

- The *signal* is a pubblic access function associated to the change of an object value
- The *slot* is a function that is called in response to a particular signal

Signals and slots are losely coupled: a class which emits a signal neither knows nor cares which slots receive the signal. Signals and slots can take any number of arguments of any types. Using this method the implementation of communication between different components is easy. Signals are emitted by objects when they change their state in a way that may be interesting to other objects. This is all the object does to communicate. The source does not know or care whether anything is receiving the signals it emits.

Slots can be used for receiving signals, but they are also normal member functions. Just as an object does not know if anything receives its signals, a slot does not know if it has any signals connected to it. This ensures that truly independent components can be created with Qt. When a signal is emitted, the slots connected to it are usually executed immediately, just like

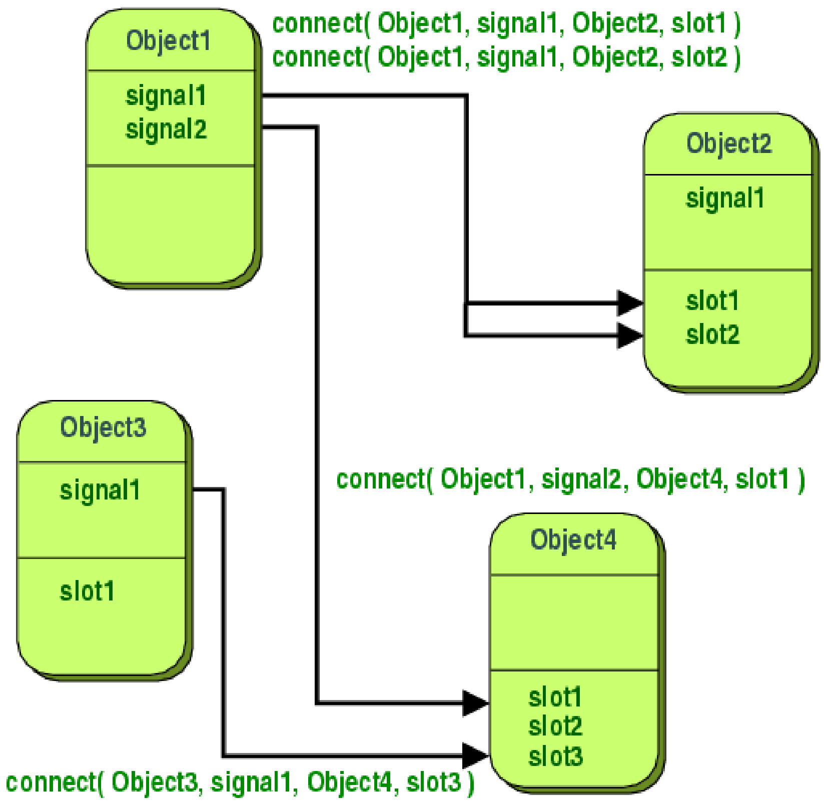

Object1
signal1
signal2
connect(Object1, signal1, Object2, slot1)
connect(Object1, signal1, Object2, slot2)
Object2
signal1
slot1
slot2
Object3
signal1
slot1
connect(Object1, signal2, Object4, slot1)
Object4
slot1
slot2
slot3
connect(Object3, signal1, Object4, slot3)

Figure 4.3: Signal/Slot connection between objects

a normal function call. When this happens, the signals and slots mechanism is totally independent of any GUI event loop. Execution of the code following the emit statement will occur once all slots have returned. If several slots are connected to one signal, the slots will be executed one after the other, in the order they have been connected, when the signal is emitted.

You can connect as many signals as you want to a single slot, and a signal can be connected to as many slots as you need. It is even possible to connect a signal directly to another signal. (This will emit the second signal immediately whenever the first is emitted.)

4.1.2 Signals and Slots: Performances

It is worth considering the perfomances of this communication method. Users of the Qt libraries asses that compared to callbacks, signals and slots are slightly slower because of the increased flexibility they provide, but the difference for real applications should be insignificant. In general, emitting a signal that is connected to some slots, is approximately ten times slower than calling the receivers directly, with non-virtual function calls. This is the overhead required to locate the connection object, to safely iterate over all connections (i.e. checking that subsequent receivers have not been destroyed during the emission), and to marshall any parameters in a generic fashion. While ten non-virtual function calls may sound like a lot, it's much less overhead than any new or delete operation, for example. As soon as you perform a string, vector or list operation that behind the scene requires new or delete, the signals and slots overhead is only responsible for a very small proportion of the complete function call costs. The same is true whenever you do a system call in a slot; or indirectly call more than ten functions. Summing up it turns out that it is accepted an overhead in the inter-object communication process in order to deal with it in a simple and flexibile way.

Whereas this consideration could be valid for a user interface application is not acceptable when dealing with realtime systems. It is necessary to assure the fulfilment of certain routines and in this way even insignificant operations triggered by the user could preempt them. For this reason in the system architecture proposed in this work the Ground Station Application has been decoupled from the simulation loop.

Chapter 5

Autopilot Software

The Pixhawk software architecture is modeled to address typical estimation and control task in deeply embedded platforms with a modular approach providing a standard programming interface. Modularity helps to develop software solutions for sophisticated problems allowing also the reusability of the code. Applications should be self contained and this is the case for the Pixhawk software where the architecture is multithreaded node-like and decouples individual applications.

The software architecture is modular and can be thought as organized into 3 layers (fig. 5.1):

1. Operating System ($NuttX$)
2. Middle Layer
3. Application Layer

In the following sections each layer is described more carefully. Understanding the functioning of the software onboard the autopilot is necessary to interact with it efficiently.

5.1 Operating System

NuttX is a real-time operating system (RTOS) with an emphasis on standards compliance and small footprint. Scalable from 8-bit to 32-bit microcontroller environments. The primary governing standards in NuttX are Posix and ANSI standards. Additional standard APIs from Unix and other common RTOS's (such as VxWorks) are adopted for functionality not available under these standards, or for functionality that is not appropriate for

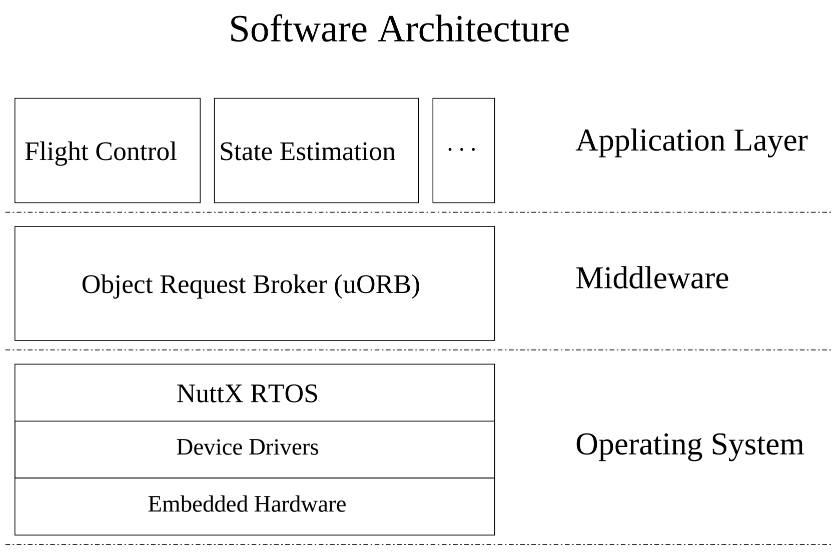


Figure 5.1: Software Architecture

deeply-embedded environments. It also offers a lightweight, bash-like shell with rich feature set and small footprint for basic user interaction. All these features make NuttX an useful interface and starting point for the developers who have a programming background on unix systems. This was one of the aims of the autopilot software developers who attempted to provide an environment to takle research topics with ease.

For what concernt the multithreading environment NuttX is fully preemptible. By defaultit performs strict priority scheduling (Fixed Priority): tasks with higher priority have exclusive access to the CPU until they become blocked. At that time, the CPU is available to tasks of lower priority. Tasks of equal priority are scheduled FIFO.

Optionally, a Nuttx task or thread can be configured with round-robin scheduler. The round-robin is similar to Fixed Priority except that tasks with equal priority share CPU time via time-slicing.

5.2 Middleware

This layer provides the necessary data structures and functions to implement inter process communication such that it is possible to develop decoupled applications. The method has been called Micro Object Request Broker (*uORB*). Understanding how different applications exchange data is of paramount importance for the timing analysis of the system and to make out how to interact efficiently with the device.

The various participants to the communication are called "nodes". Nodes which send messages are called "publishers" while the nodes which receive messages are called "subscribers". The data flows are subdivided in logical channels called "topics". In this implementation of the *uORB* each topic contains only one message type and there aren't queues.

In this messaging pattern senders do not know anything about the receivers, and at the same time the receivers simply declare their interest in receiving data belonging to the selected topics, without caring about who produces it. The resulting communication framework is then composed by nodes which remain ignorant of system topology: the applications are decoupled as required. Nodes can be publisher and subscriber at the same time (fig. 5.2). Few steps are required to set up a data exchange using this

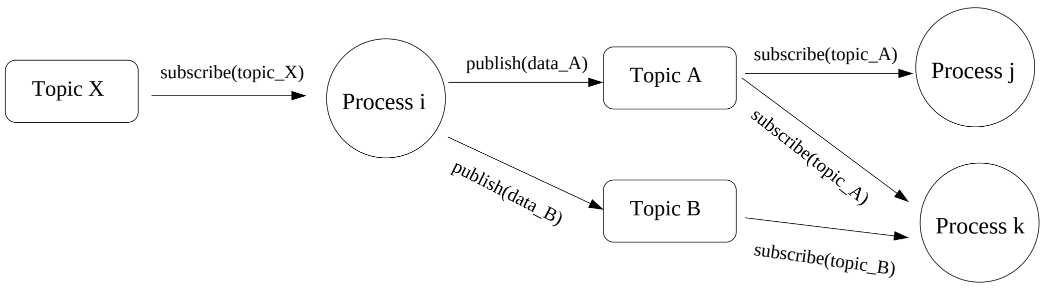


Figure 5.2: Publisher/Subscriber framework

method. This is clear looking at the publishing and subscribing procedures:

5.2.1 Publishing

In order to publish data on a topic it is necessary to follow a procedure which provides to initialize the communication channel:

1. Topic definition (only once):
   It is necessary to define the data structure (message) on that channel. Clearly the topic definition is necessary only if it is not already present in the system.

2. Topic advertisement (only once):
   Before data can be published to a topic, it must be advertised, that is the publisher has to register itself in the publishers' list of that topic.

3. Topic publication:
   Once a topic has been advertised it is possible to publish updates to it.

The publication is made with an atomical operation to guarantee the consistency of the data.

When publishing, since there is no queue for the transmitted data, the previous value is replaced and all the subscribers can only receive the last value.

5.2.2 Subscribing

The reception procedure is also subdivided in several steps:

1. Subscription to the selected topic (only once):
   The node declares its interest in the selected channel.
2. Checking for update:
   The node can verify the presence of new data on the channel. Each subscriber keeps track of the time of the last data received and thus can compare it with the current data timestamp (updated by the publisher) when checking the update.
3. Copying data from the topic:
   The data is copied in a local structure atomically.

5.2.3 Middleware as Synchonization means

This framework offers the possibility to achieve a synchronization between tasks. A subscriber that depends on publications as a source of data can wait for publications to any number of subscriptions simultaneously. This is done using the *poll()* function in the same fashion as waiting for data on a file descriptor. This works because subscriptions are actually file descriptors themselves. Since some topics are updated at a high frequency (sensors topics) it is possible to limit the rate at which subscribers receive updates on it. Summing up this architecture has particular strenghts for realtime control applications:

- The topic handle is implemented as virtual file, allowing listeners to do blocking waits on interfaces and drivers
- The read-write lock of the publication allows efficient concurrency and ensures atomic operations one the topic content
- Subscribers can ask for notification limit when retrieving data from high rate publishers.

- The asynchronous/blocking wait approach combined with the task priority setup of the operating system allows minimal latency and deterministic scheduling in the control pipeline

5.3 Application Layer

The Application Layer can be subdivided into 2 groups:

1. System Applications
2. Flight Control Applications(PX4 Stack)

5.3.1 System Applications

The system applications provide functionalities for the user and also for other running processes. There are functions to log data to SD-card, to retrieve a list of running applications and resource usage, to obtain timing statistics of running processes. A foundamental application on which is based the off-board communicationis the *mavlink application*. It manages the send/receive of MAVLink packets through the serial interface and carries out the conversion between those packets and the object request brocker structures for the inter process communication (fig. 5.3).

5.3.2 Flight Control Applications

The PX4 flight control stack is a custom, BSD licensed flight control stack, providing fully autonomous waypoint flight for multicopter and fixed wing aircraft. It uses a common codebase and common flight management code. It follows a very flexible and structured approach, which allows to run plane and multicopter controllers with the same waypoint and safety state machine handling. The applications tightly involved in the flight of the quadrotor are represented, grouped for functionality, by

- Flight Safety and Navigation
  - commander
  - navigator
- Attitude and Position Estimator
  - attitude_estimator_ekf : EKF-based attitude estimator
  - position_estimator_inav : Inertial navigation position estimator

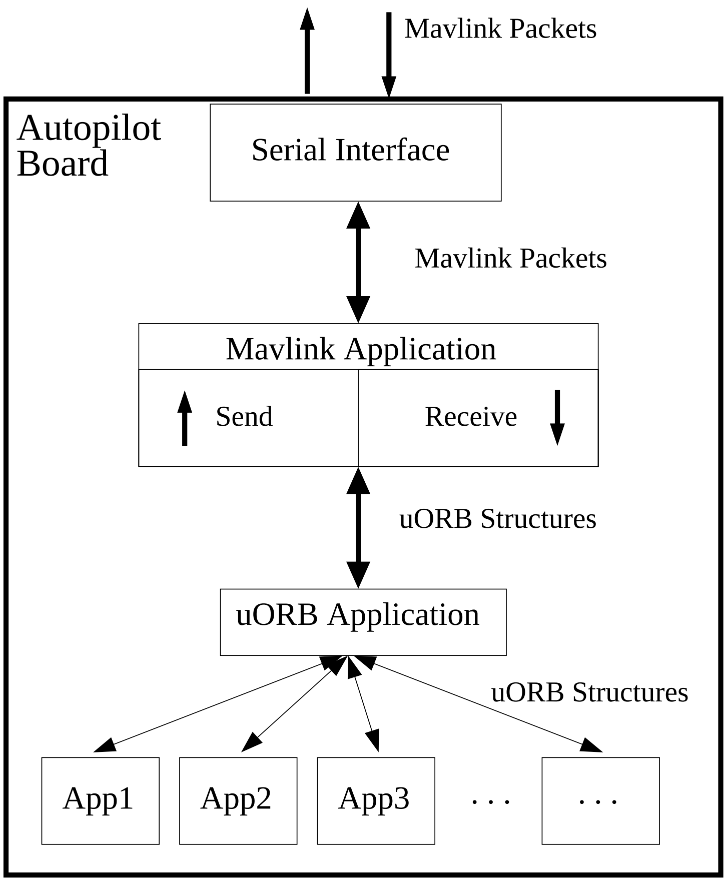


Figure 5.3: Mavlink application: data flux

- Multirotor Attitude and Position Controllers
  - Multirotor attitude controller
  - Multirotor position controller

5.3.3 Tasks Synchronization

The various tasks running onboard have precedences because each of them needs the data provided by others to execute: estimators needs new measurements to run and other applications in turn need state estimate to execute. It is thus necessary to synchronize applications and a possible approach might be to run algorithms in a single loop, such that the precedences constraints are meet automatically. This method gives birth to a coupled system with low design flexibility and cumbersome functioning in case of components running at different rate. Since the system supports multithreading, it might be possible to run the algorithms in separate tasks, each one with its own period, and check for the presence of new input data at each activation time. This solution, although constituting a decoupled system, it is not efficient. Ideed in some circumstances, depending on the order of execution of the tasks, each loop can miss the next loop making the system experience high latency (fig. 5.4 and fig. 5.5).

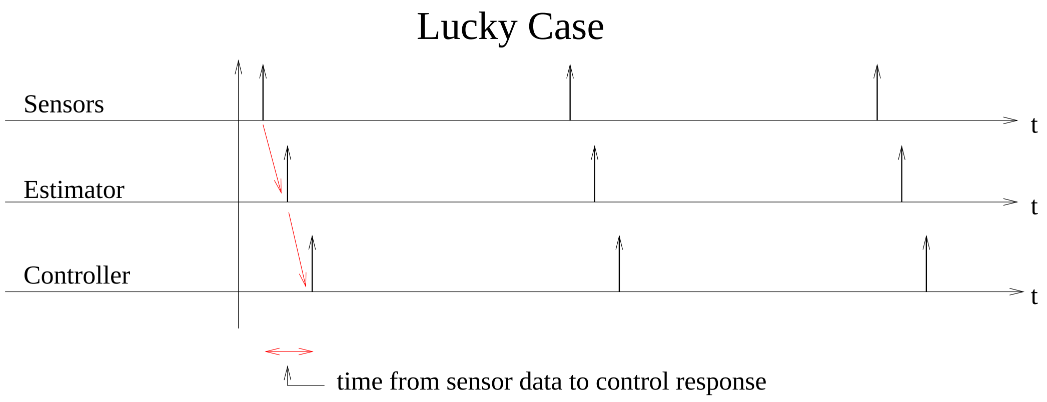


Figure 5.4: Sequence of events which determines low latency in the response

This issue is of paramount importance for the control loops performances. Given the tools offered by the OS and the *uORB* communication mechanism it is possible to tackle the control problem minimizing the latency of the system while maintaining the decoupled structure.

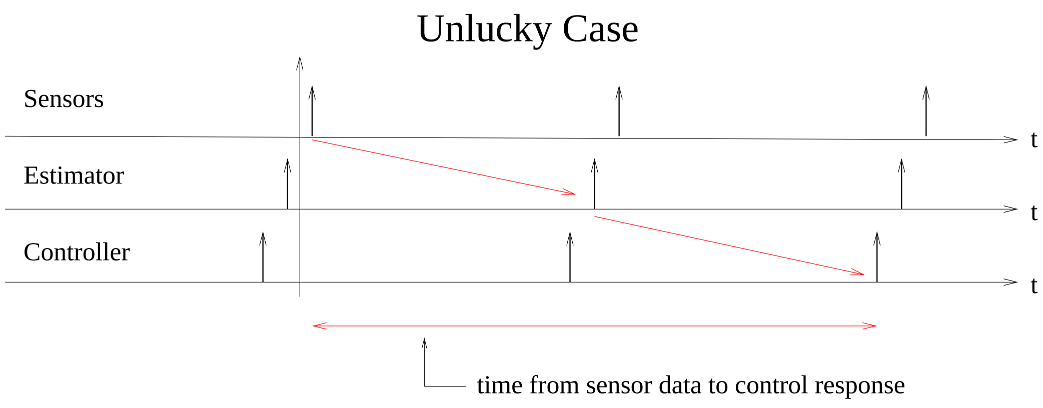


Figure 5.5: Sequence of events which determines high latency in the response

Each application, run as a separated thread, exchanges data with the others via *$uORB$*. As pointed out in the part describing the messaging structure it is possible to call a $poll()$ on the requested topic and wait for new data without usign CPU. In this way the components of a control chain are run in the same order, without the risk of bad interlacing, and thus with low latency in between.

An example of functioning is shown in fig. 5.6 where three tasks involved in the control loop are synchronized in a control chain. The starting point is the tread publishing the sensor data, which triggers the attitude estimator task waiting from them. When the estimator publishes the estimated attitude the control task is triggered. Even if, for some reason, the starting task involved with sensor data publication should be delayed, the architecture assures the maintance of the execution order as can be observed in the same figure, where a possible instance of functioning has been drawn.

Onboard the pixhawk the fast attitude control loop is achieved in this way. The loop frequency is $250Hz$. The tasks have the same priority, are scheduled FIFO and with the technique saw before it is possible to achieve the requested precedences.

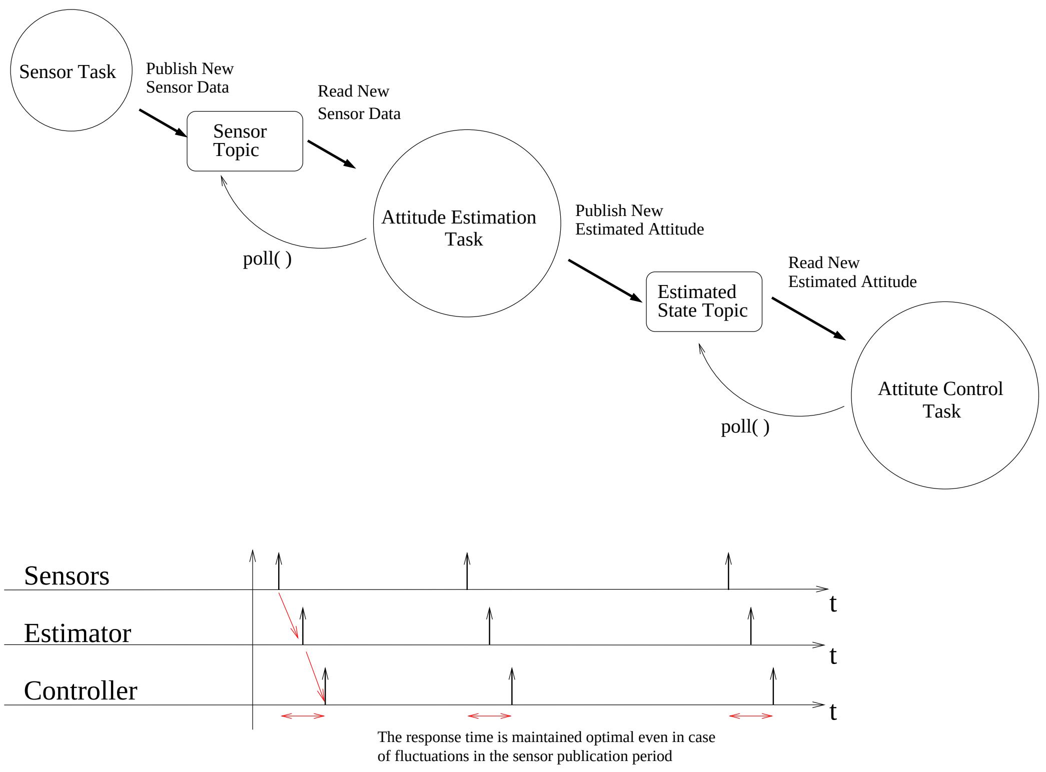


Figure 5.6: Control Chain Example

Chapter 6

Data Governor

Given the architecture described in the previous chapters it is clear that in order to close the loop between hardware and software it is necessary to consider the timing of the operations. The Graphical Interface approach of QGroundControl makes impossible to predict the time of accomplishment and puts on the same level the interaction of the user with the interface and messages exchange between simulator and autopilot board.

A viable solution for this problem consists in developing an external Application able to manage messages, decoupling the User Interface section of the framework from the Simulative part.

In order to realize this decoupling the data flow is sorted into two categories: one regarding the functioning of the Ground Station Application; the other regarding the Simulator Block.

The data stream pertinent the ground station consists in the state of the UAV, such as estimated position, estimated attitude, fightmode etc; whereas, the datastream regarding the simulator consists in sensor data (**??**) and and actuation outputs. The data exchange with the Ground Station Application allows to maintain the capability to send commands to the Autopilot Board, to visualize telemetry data in realtime, to check the state of the vehicle during the simulation.

HIL_SENSOR Message

Field Name	Type	Description
time_usec	uint64_t	Timestamp
xacc	float	X acceleration (m/s^2)
yacc	float	Y acceleration (m/s^2)
zacc	float	Z acceleration (m/s^2)
xgyro	float	Angular speed around X axis body frame (rad/s)
ygyro	float	Angular speed around Y axis body frame (rad/s)
zgyro	float	Angular speed around Z axis body frame (rad/s)
xmag	float	X Magnetic field ($Gauss$)
ymag	float	Y Magnetic field ($Gauss$)
zmag	float	Z Magnetic field ($Gauss$)
abs_pressure	float	Absolute pressure ($mBar$)
diff_pressure	float	Differential pressure ($mBar$)
pressure_alt	float	Altitude calculated from pressure (m)
temperature	float	Temperature (C^o)
fields_updated	uint32_t	Bitmask for fields that have updated since last message

HIL_GPS Message

Field Name	Type	Description
time_usec	uint64_t	Timestamp
fix_type	uint8_t	0-1: no fix, 2: 2D fix, 3: 3D fix
lat	int32_t	Latitude (WGS84), in degrees * 1E7
lon	int32_t	Longitude (WGS84), in degrees * 1E7
alt	int32_t	Altitude (AMSL), in meters * 1000 (positive for up)
eph	uint16_t	GPS HDOP horizontal dilution of position in cm (m*100)
epv	int16_t	GPS VDOP vertical dilution of position in cm (m*100)
vel	uint16_t	GPS ground speed (m/s * 100)
vn	int16_t	GPS velocity in cm/s in NORTH direction in earth-fixed NED frame
ve	int16_t	GPS velocity in cm/s in EAST direction in earth-fixed NED frame
vd	int16_t	GPS velocity in cm/s in DOWN direction in earth-fixed NED frame
cog	uint16_t	Course over groundin degrees * 100
satellites_visible	uint8_t	Number of satellites visible

This data flow should have the highest priority because is involved in the control algorithm onboard the Autopilot Board. The application not only accomplishes the data routing between participants managing priorities but also synchonizes threads and assures correct timings of the messages transmission.

Since the function of the *Data Governor* application is to assure a reliable simulation result, that is that the feedback returned is more realistic as possible, consideration have been made when dealing with possible data frequency variation. Indeed the capability to assure a constant data rate has been considered more important then exchanging totally coherent data. Following this approach in case of missing data, due to eventual delays in the execution flow, the application provides to patch the missing value inserting an artificial value equal to the previously sent data. Indeed for the autopilot computer the situation of missing data sensor is more severe that elaborating an artificial value.

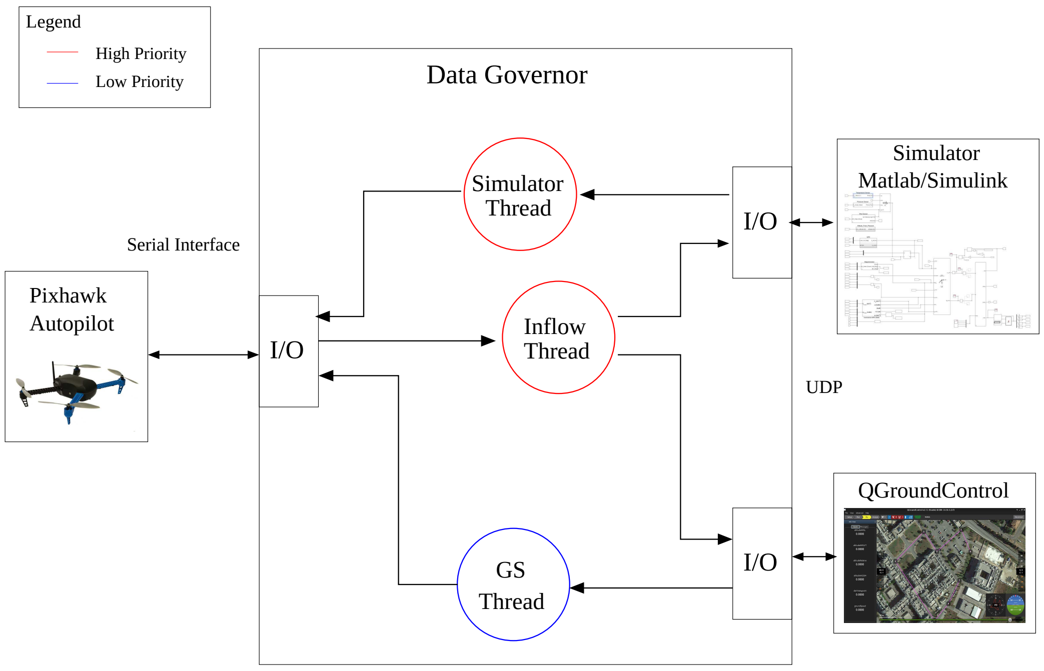


Figure 6.1: Application Structure

6.1 Application Structure

The application has been developed usign the *ptask* library [18]. This is a C library that simplifies the real-time programming in Linux, allowing to create tasks, synchronize them and perform other functions without the low level implementation details of the standard *pthread* library. The design of the application has been carried out with the goal of minimizing the resource sharing between different threads. The functionalities of the *Data Governor* are accomplished using three threads:

1. Inflow Thread:
   This thread retrieves data from the board polling the serial interface and routing the obtained packet to the destination structures:

2. Simulator Thread:
   This thread retrieves data from the the simulator application and send them back to the Autopilot Board:

3. Ground Station Thread:
   This thread retrieves data from the ground station application and send them to the Autopilot Board

```
Variables_initialization();
while not time to exit do
    fetch_message_from_autopilot_interface();
    for each received message do
        if controls for the simulator? then
            update_controls();
        else
            send_to_groundstation();
        end
    end
    send_controls_to_simulator();
    wait_for_period();
end
```

Algorithm 1: Inflow Thread

```
Variables_initialization();
while not time to exit do
    fetch_sensor_from_simulator_interface();
    update_sensors_data(); compose_mavlink_message();
    if Is the Autopilot in HIL mode? then
        send_sensorData_to_autopilot();
    end
    wait_for_period();
end
```

Algorithm 2: Simulator Thread

```
Variables_initialization();
while not time to exit do
    check_new_data_from_GS();
    send_message_to_autopilot();
end
wait_for_period();
```

Algorithm 3: Ground Station Thread

Chapter 7

Experiments

The final part of the work is spent in the testing of the overall structure. The tests consist in profiling the time characteristics of the processes involved in the simulation and verifying the fulfilment of time constraints. The characterization of the timing properties of the system is necessary to check the correct functioning and to identify the critical parts which need to be improved in future works. Using the gathered data, mean values, standard deviations and worst case execution times of the various operations are evaluated. In this way it is possible to describe statistically the timing performance of the framework.

7.1 Experiment Setup

In the experiment the autopilot board has been connected to an host pc running the *Data Governor* application and the FlightGear instance. The *Matlab/Simulink* application has been run on another machine on the LAN. A further serial connection has been established with the autopilot board to check the status of the system throught debugging shell.

Several simulation have been run and the data is collected and successively analyzed with Matlab to extract the statistical information. In the experiment the Ground Station Thread is not active, so there are only two tasks running.
The quantities of interests are:

1. Interval of time between the activation of the Inflow thread and the Simulator thread

2. Attivation Time of the single tasks

3. Time necessary to the simulator to produce sensor data

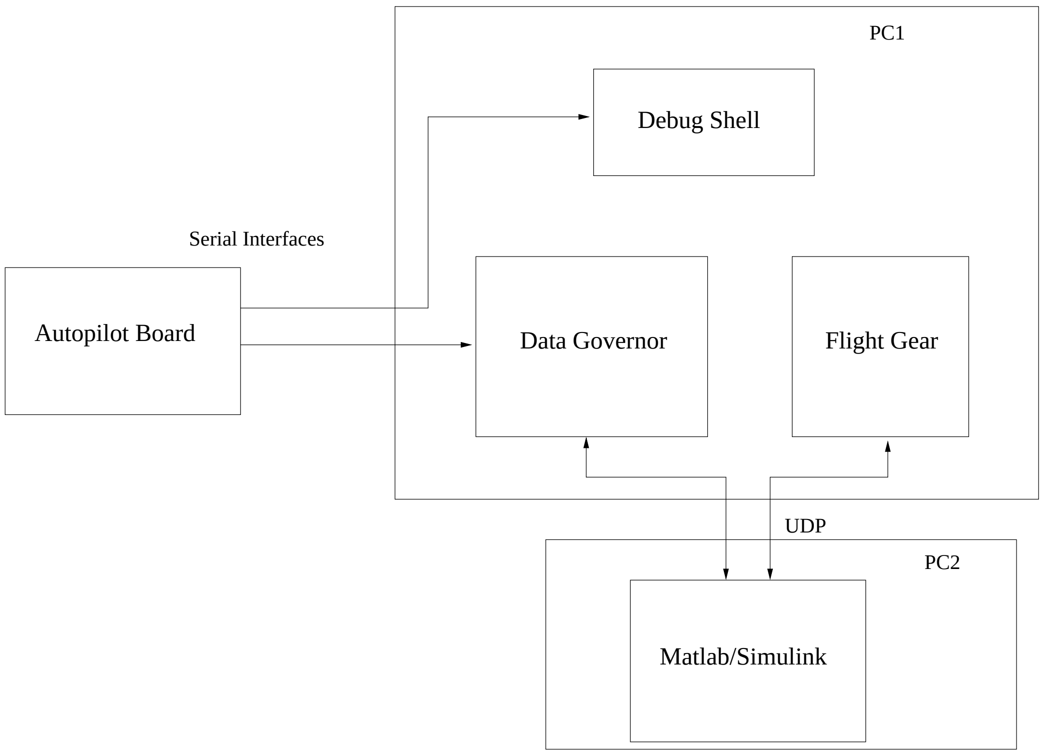


Figure 7.1: Experiment Setup

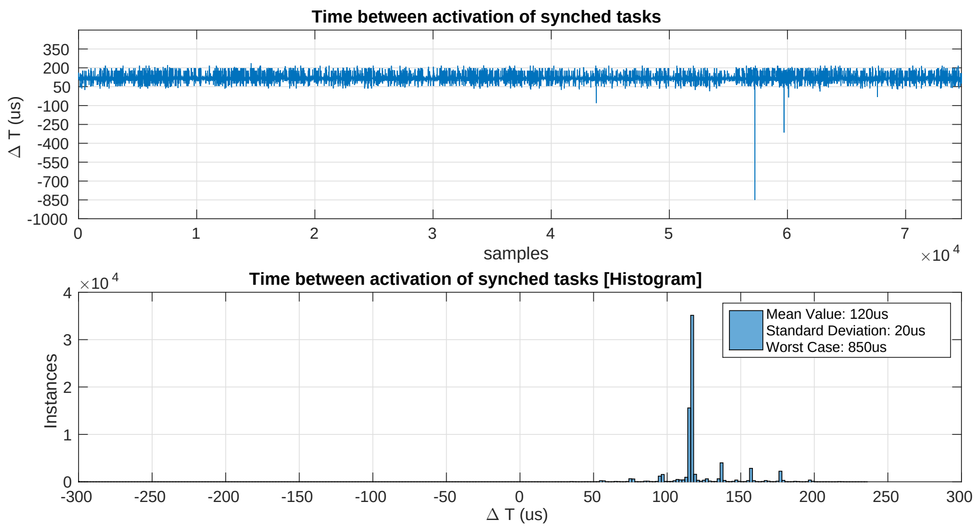


Figure 7.2: Statistic of the inter-activation time between Inflow Thread and Simulation Thread

4. Time necessary to the tasks to perform the operation in a cycle

7.2 Experiment Results

The data collected relative to the interval of time between the activation of the Inflow thread and the Simulator thread shows (Figure 7.2) that there is no drift and that the two treads are synchronized as requested. The mean value of the inter-activation time is $120us$ with a variance of $20us$. The worst case has been recorded to be $850us$.

The measurements of the activation times of the tasks show that it is possible to achieve good triggering precision (Figure 7.3 and Figure 7.4). This property is foundamental if it is necessary to guarantee the delivery of the messages in with a given frequency. The standard deviation of the measured data has been found to be $23us$ for the activation period of the Inflow Thread and $26us$ for the activation period of the Simulator Thread.

The execution time of the two threads is shown in Figure 7.5 and in Figure 7.6. These values confirm that the data exchange operations performed in the tasks' periods are accomplished without coming up against considerable delays. The time to accomplish the simulation step is reported in Figure 7.9. Even if the mean value is included in the maximum admissibile time it has been noticed that the standard deviation of $3720us$ is quite large. Indeed, as it is shown in the Figure 7.8, the inter-arrival time of sensor data from the

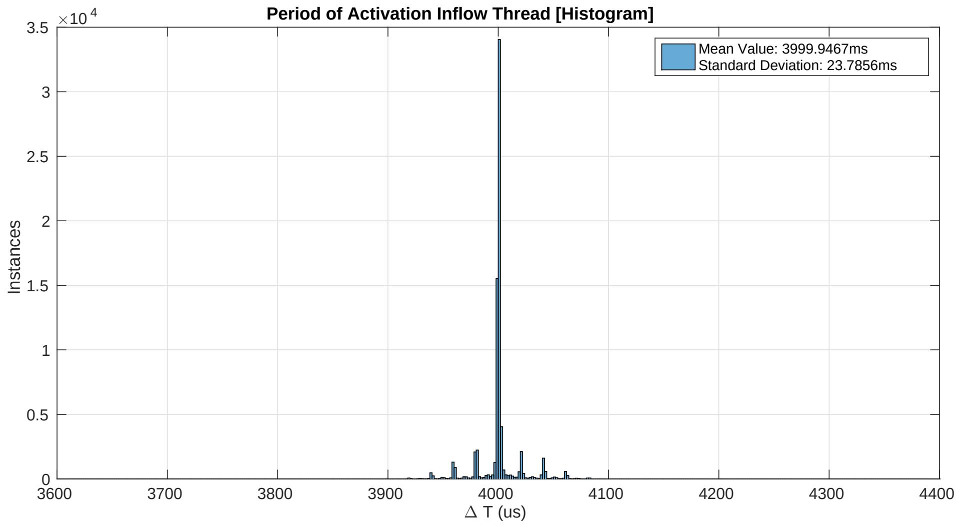


Figure 7.3: Statistic of the activation time of the Inflow Thread

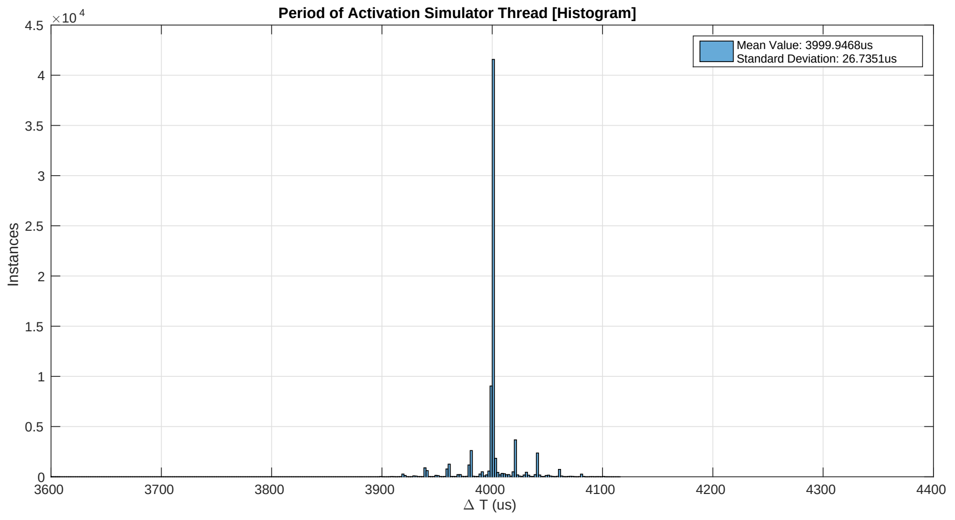


Figure 7.4: Statistic of the activation time of the Simulation Thread

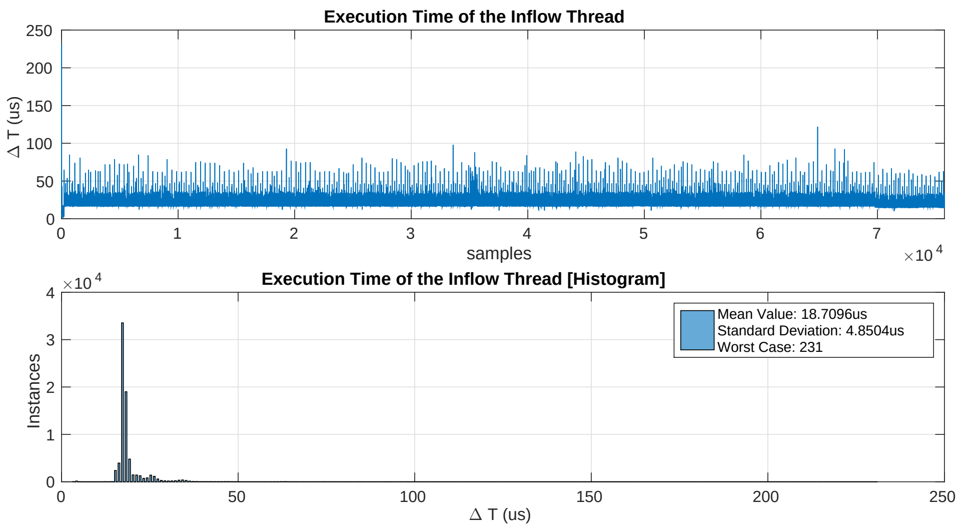


Figure 7.5: Statistic of the execution time of the Inflow Thread

simulator can be far above the requested $4000us$.

A negative result has been obtained analysing the data on the simulator response time. Unfortunately the time necessary to obtain a response from simulation is bigger than the $4ms$ sampling time of the autopilot board. The mean response time is $10.942ms$ with a standard deviation of $11.834ms$. This issue is probably due to the way simulink implements the UDP block used to communicate.

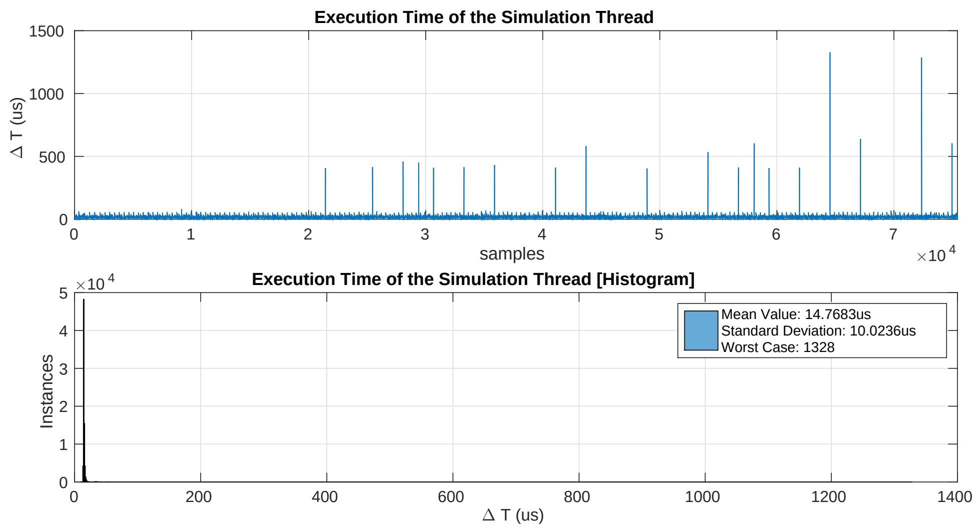


Figure 7.6: Statistic of the execution time of the Simulation Thread

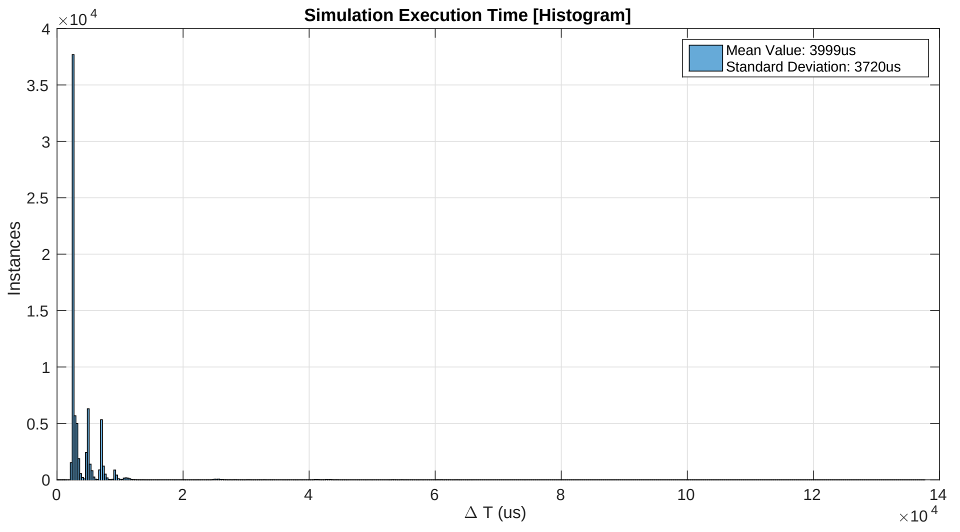


Figure 7.7: Statistic of the execution time of the Simulation Thread

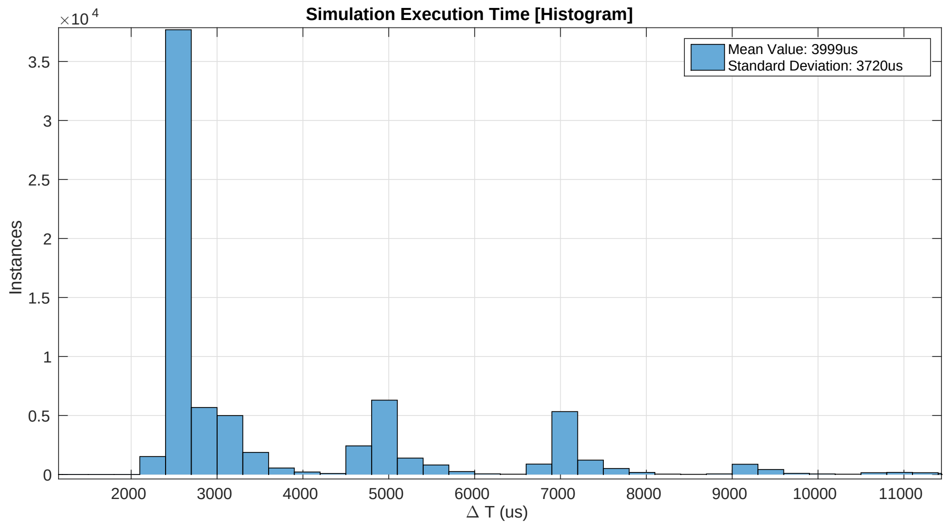


Figure 7.8: Statistic of the execution time of the Simulation Thread(Detail)

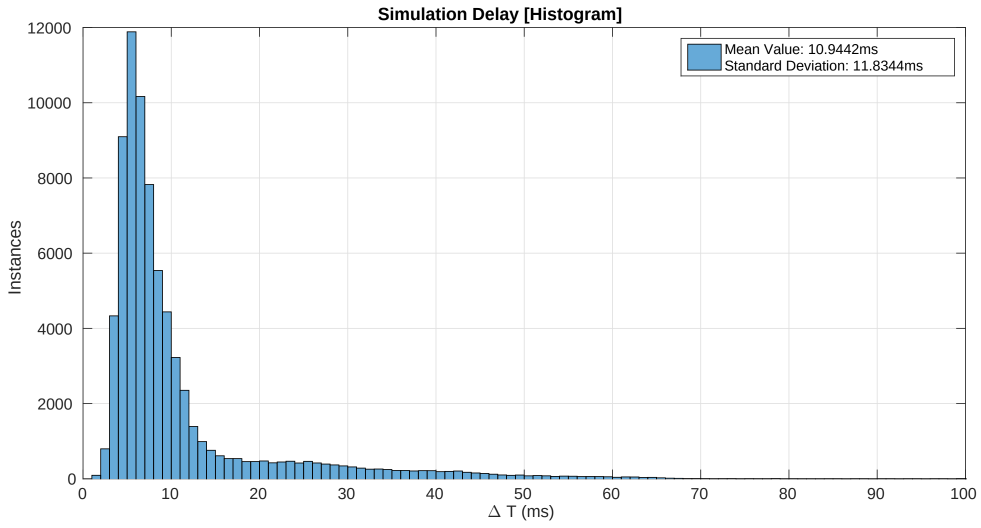


Figure 7.9: Statistic of the Response Time of the Simulator Block

Chapter 8

Conclusions

In this chapter a summary of the work done is presented in order to analize the obtained results and compare them with the initial objectives. At the end of the summary some guidelines for future works will be proposed.

In this thesis period, after an analysis of the state of the art and related works, a new structure for a hardware in the loop simulation environment has been proposed. The starting point of the work is the open-source PX4 project which offers the necessary components to start developing without rebuild everything from scratch. Given the absence of a clear documentation, the information about the functioning of the various application, firmware and tools has been retrieved mainly by direct inspection of the code.

Whereas this approach is difficult and slows down considerably the achievement of the results, it has revealed issues in the PX4 implementation. Trying to improve the current solution and to fix the detected problems a new framework has been developed. Precisely it has been considered necessary to find another solution for what concern the simulator and the architecture of the overall system.

The obtained new framework is aimed to constitute an useful tool for the development of guidance, navigation and control algorithms. Precisely it allows check the correct execution of the designed algorithms directly on the target control board. This result is possible thanks to the achieved timing precision with the time constraints are met. The ability to take the execution time into consideration allows to check the correctness of the simulation flow itself guaranteeing the feature of hardware in the loop simulator. The work ends with tests aimed to verify the capability to manage activation times and synchronization between threads with sufficient precision. The results show that the required data exchange is accomplished and thus it is a viable option for the realization of an hardware in the loop simulator.

The tests show also that the simulator component introduces a consistent and not negligible delay. The problem has been associated with the way *Matlab/Simulink* manages the UDP connection.

8.1 Future Works

The results of the tests show that the simulation approach proposed is not able to fully satisfy the requirements. Precisely the failing has been attributed to the implementation of the UDP communication in the *Matlab/Simulink* environment. It is thus necessary to reimplement the UDP communication in a more efficient method or find another solution for what concern the simulator application.

The tests that have been carried out without considering the presence of the Ground Station. It would be necessary to check the effects on the system when another tasks participate to the data exchange with the autopilot board.

Bibliography

[1] Luis Merino, Fernando Caballero, J.R. Martinez-de-Dios, Ivan Maza, Anibal Ollero *An Unmanned Aircraft System for Automatic Forest Fire Monitoring and Measurement* Journal of Intelligent & Robotic Systems January 2012, Volume 65, Issue 1, pp 533-548

[2] Andre‘ Posch, Salah Sukkarieh *UAV based search for a radio tagged animal using particle filters* Australasian Conference on Robotics and Automation (ACRA), December 2-4, 2009, Sydney, Australia

[3] Mitch Bryson, Alistair Reid, Fabio Ramos and Salah Sukkarieh *Airborne Vision-Based Mapping and Classification of Large Farmland Environments*, Journal of Field Robotics Special Issue: Visual Mapping and Navigation Outdoors Volume 27, Issue 5, pages 632655, September/October 2010

[4] WoonSik Kim, ByungSun Lee, KyungSoo Kim, TaeSoo Yang *A real-time HWIL simulation control system architecture for implementing evaluation environment of complex embedded systems*, International Conference on Advanced Communication Technology (ICACT), Seoul 2011

[5] Lorenzo Pollini, Valeria Parnenzini, Mario Innocenti *Distributed Real-Time Hardware- and Man-in-the-loop Simulation for the ICARO II Unmanned Systems Autopilot*, Latest Trends in Information Technology, 2012

[6] Aradi S., Becsi T., Gasparq P. *Experimental Vehicle Development for Testing Autonomous Vehicle Functions* 10th International Conference on Mechatronic and Embedded Systems and Applications (MESA), 2014

[7] Home - PX4 Autopilot Project *https://pixhawk.org/start*

[8] ETH PIXHAWK *https://pixhawk.ethz.ch/*

[9] FlightGear Flight Simulator *http://www.flightgear.org/* 2015

[10] X-Plane 10 Flight Simulator *http://www.x-plane.com/desktop/home/* 2015

[11] Simple multirotor simulator with MAVLink protocol support *https://github.com/DrTon/jMAVSim* 2015

[12] Qt Documentation: Signal and Slots *http://doc.qt.io/qt-5/signalsandslots.html*

[13] Gabriel M. Hoffmann, Haomiao Huang, Steven L. Waslander, Claire J. Tomlin *Quadrotor Helicopter Flight Dynamics and Control: Theory and Experiment* AIAA Guidance, Navigation and Control Conference and Exhibit 20 - 23 August 2007, Hilton Head, South Carolina

[14] Engr. M. Yasir Amir, Dr. Valiuddin Abbass *Modeling of Quadrotor Helicopter Dynamics* International Conference on Smart Manufacturing Application April. 9-11, 2008 in KINTEX, Gyeonggi-do, Korea

[15] Paul Pounds, Robert Mahony, Peter Corke *Modelling and Control of a Quad-Rotor Robot* Australian National University, Canberra, Australia CSIRO ICT Centre, Brisbane, Australia

[16] Paul Pounds, Robert Mahony, Joel Gresham *Towards Dynamically-Favourable Quad-Rotor Aerial Robots* Australian National University, Canberra, Australia CSIRO ICT Centre, Brisbane, Australia

[17] Nathan Michael, Daniel Mellinger, Quentin Lindsey, Vijay Kumar *The GRASP Multiple Micro-UAV Test Bed* IEEE Robotics & Automation Magazine September 2010

[18] Giorgio Buttazzo, Giuseppe Lipari, *Ptask: an Educational C Library for Programming Real-Time Systems on Linux* 18th Conference on Emerging Technologies & Factory Automation (ETFA), IEEE 2013