Control Systems for Multi-rotors
Principles, Modeling and Software Design

Corrado Santoro

ARSLAB - Autonomous and Robotic Systems Laboratory
Dipartimento di Matematica e Informatica - Università di Catania, Italy
santoro@dmi.unict.it

Multirotor Structure and Dynamics

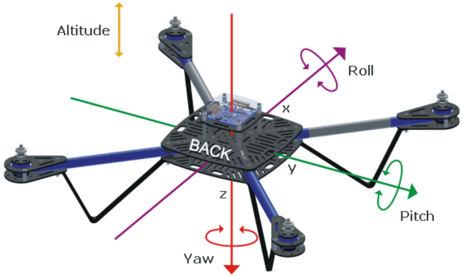
Multirotors: definition

A **multirotor** (a.k.a. "drone") is an aerial vehicle characterised by:

- An **even set** of equal *horizontal propellers* (and motors), ≥ 4, symmetrically placed in a circular shape
- A **symmetric/balanced** airframe (even if not strictly mandatory)
- **VTOL** (Vertical Take-off and Landing) capabilities
- **Four degrees of freedom**, $XYZ + Heading$
- **No critical issues** from the mechanical/aerodynamic point of view
- Total control in **software**, no mechanical parts

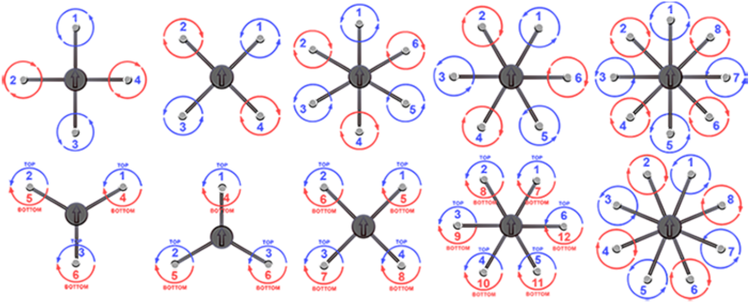
Reference System

- The **body reference system** usually employed is the one in figure
- The system also define the **Euler angles** that represents the **attitude**:
  - **roll**, ϕ
  - **pitch**, θ
  - **yaw**, ψ
- The **pose** of the multirotor is represented by:
  - $\{X, Y, Z, \phi, \theta, \psi\}$, in the **Earth frame**



Airframes and Constraints

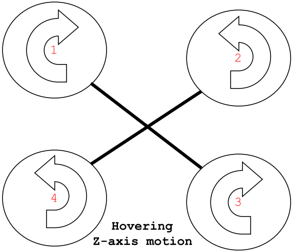
- Motors/propellers must be the **same**
- Motors/propellers must be **even** ≥ 4
- Motors/propellers must be placed in a **circular shape**
- Propellers must rotate in **opposite directions** in-pair (third Newton's Law compensation)
- Propellers must have **opposite pitches** in-pair
- The number and position of propellers define the **airframe model**



Motion

- Motion is achieved by **modulating** propeller speeds
- We can assume a *virtual pilot* able to give the commands (as in an airplane):
  - **Thrust**, the "power" to the motors (throttle control)
  - **Roll** and **Pitch**, the "control joke"
  - **Yaw**, the "pedals"
- Let us assume that these commands are *variables* belonging to the ranges:
  - $thrust\_cmd \in [0, TH_{max}]$
  - $roll\_cmd \in [-R_{max}, R_{max}]$
  - $pitch\_cmd \in [-P_{max}, P_{max}]$
  - $yaw\_cmd \in [-Y_{max}, Y_{max}]$
- These commands must be *"transferred"* to the motors on the basis of the **specific airframe**

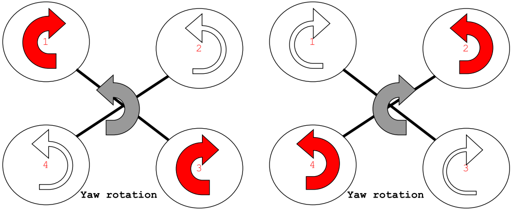
Motion: Hovering and Z-translation



Vertical motion is achieved by keeping all propeller speeds the same and proportional to a **thrust command** (we assume 1-proportionality):

$$\omega_1 = thrust\_cmd$$
$$\omega_2 = thrust\_cmd$$
$$\omega_3 = thrust\_cmd$$
$$\omega_4 = thrust\_cmd$$

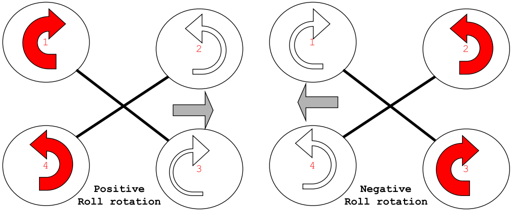
Motion: Yaw rotation in X-shaped quads



Yaw rotation is achieved by modulating propeller speeds in-pairs $1-3/2-4$, proportional to a **yaw command**:

$$
\begin{aligned}
\omega_1 &= thrust\_cmd - yaw\_cmd \\
\omega_2 &= thrust\_cmd + yaw\_cmd \\
\omega_3 &= thrust\_cmd - yaw\_cmd \\
\omega_4 &= thrust\_cmd + yaw\_cmd
\end{aligned}
$$

Motion: Roll rotation in X-shaped quads

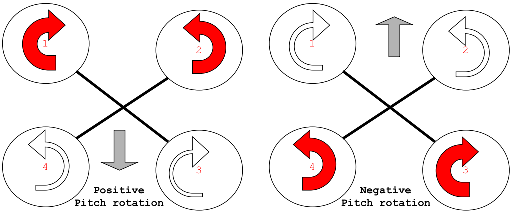


Roll rotation is achieved by modulating propeller speeds in-pairs $1-4/2-3$, proportional to a **roll command**:

$$
\begin{aligned}
\omega_1 &= thrust\_cmd - yaw\_cmd + roll\_cmd \\
\omega_2 &= thrust\_cmd + yaw\_cmd - roll\_cmd \\
\omega_3 &= thrust\_cmd - yaw\_cmd - roll\_cmd \\
\omega_4 &= thrust\_cmd + yaw\_cmd + roll\_cmd
\end{aligned}
$$

Roll rotation implies a decomposition of the thrust force: a **drag** force appears that drives the frame in a **translated flight** along Y axis

Motion: Pitch rotation in X-shaped quads

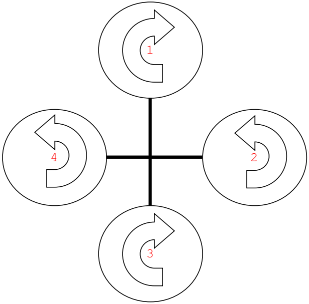


Pitch rotation is achieved by modulating propeller speeds in-pairs 1 – 2/3 – 4, proportional to a **pitch command**:

$$\begin{aligned}
\omega_1 &= thrust_cmd - yaw_cmd + roll_cmd + pitch_cmd \\
\omega_2 &= thrust_cmd + yaw_cmd - roll_cmd + pitch_cmd \\
\omega_3 &= thrust_cmd - yaw_cmd - roll_cmd - pitch_cmd \\
\omega_4 &= thrust_cmd + yaw_cmd + roll_cmd - pitch_cmd
\end{aligned}$$

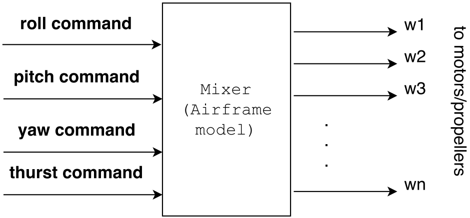
Pitch rotation implies a decomposition of the thrust force: a **drag** force appears that drives the frame in a **translated flight** along X axis

Motion in Plus-shaped quads



$$\omega_1 = thrust\_cmd - yaw\_cmd + pitch\_cmd$$
$$\omega_2 = thrust\_cmd + yaw\_cmd - roll\_cmd$$
$$\omega_3 = thrust\_cmd - yaw\_cmd - pitch\_cmd$$
$$\omega_4 = thrust\_cmd + yaw\_cmd + roll\_cmd$$

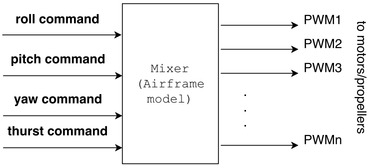
Motion: the Mixer



- The **mixer** is the software component that *translates* attitude commands to **motor commands**
- It depends airframe model and basically implements a matrix M such that

$$\begin{bmatrix} \omega_1 \\ \omega_2 \\ \cdots \\ \omega_n \end{bmatrix} = M \begin{bmatrix} roll\_cmd \\ pitch\_cmd \\ yaw\_cmd \\ thrust\_cmd \end{bmatrix}$$

The Mixer: practical aspects



- Practically, the outputs of the **mixer** are not the w_n but the **duty cycle** values of the motor PWM drivers

$$\begin{bmatrix} PWM_1 \\ PWM_2 \\ \cdots \\ PWM_n \end{bmatrix} = M \begin{bmatrix} roll\_cmd \\ pitch\_cmd \\ yaw\_cmd \\ thrust\_cmd \end{bmatrix}$$

- PWM values are then **saturated** using a technique that avoids certain side-effects

The Mixer: Saturation

- Each PWM output cannot be more than a maximum (PWM_{MAX}, i.e. the 100% duty cycle)
- Each PWM output cannot be less than the minimum value that stops the propeller: this is required to avoid multirotor **flip**
- The constraints above must not affect the displacement imposed, otherwise the control could not work

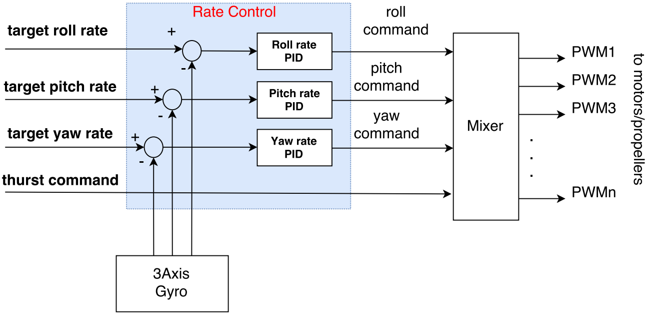
$M \leftarrow \max\{PWM_1, PWM_2, \ldots, PWM_n\}$
$m \leftarrow \min\{PWM_1, PWM_2, \ldots, PWM_n\}$
if $M > PWM_{MAX}$ **then**
    $D \leftarrow M - PWM_{MAX}$
    $PWM_i \leftarrow PWM_i - D$
    $M \leftarrow PWM_{MAX}$
    $m \leftarrow m - D$
end if
if $M - m > \Delta PWM_{MAX}$ **then**
    $m \leftarrow M - \Delta PWM_{MAX}$
end if
if $PWM_i < m$ **then**
    $PWM_i \leftarrow m$
end if

The Control System of a Multirotor

Rate and Attitude Control

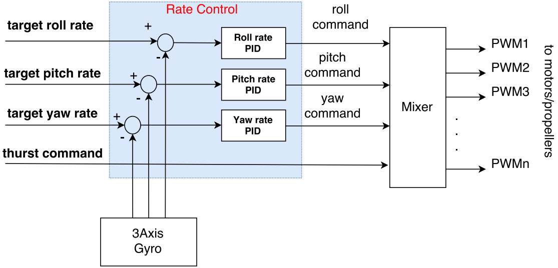
- The mixer outputs PWM values and does not have control on the real forces of the propellers
- In order to ensure **stability**, proper sensors must be employed that detects the *attitude* of the multirotor
- The control of stability is achieved by means of two control loops:
  - **Rate Control**, controls *angular speeds* $\dot{\phi}, \dot{\theta}, \dot{\psi}$, by means of a 3-axis gyro
  - **Attitude Control**, controls *Euler angles* ϕ, θ, ψ, by means of a 6-DOF or 9-DOF IMU

Rate Control: the “Acro” Mode



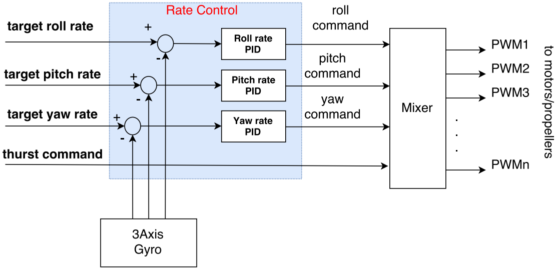
- The **Rate Control** module performs a PID control on angular rates $\dot{\phi}, \dot{\theta}, \dot{\psi}$ on the basis of:
  - **Target Rates**, given as input
  - **Current Rates**, given by the gyro
- When the target rates are given by the RC command, the mode is called **acrobatic**

Rate Control: Practical Aspects



- **Rate Controllers** are usually **PI** or **PID** controllers (the derivative part is often filtered by a LPF)
- Outputs are **saturated** to a specific value (usually 100% of PWM duty cycle)
- The **anti-wind-up** optimisation is included
- Since controllers must operate only "in flight", they are "activated" only when the **thrust** command is greater than a **threshold**

Rate Control: Implementation

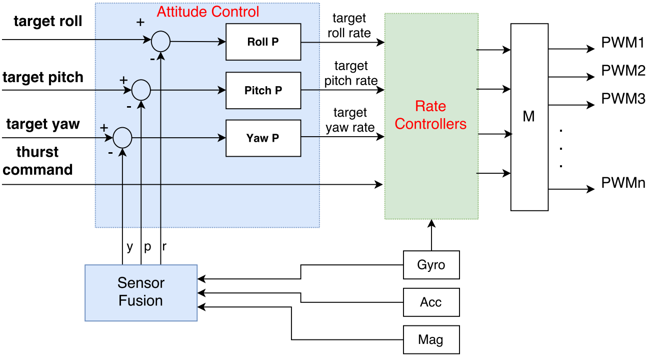


- **Rate Control** module is implemented as a **periodic task** triggered by:
  - A **Timer**, with a specific period
  - The **gyro sampling frequency**
- Periods are in the order of $200 - 500Hz$
- The code implements the classical PID algorithm with anti-wind-up

Rate and Attitude Control

- Rate control ensures that real angular rates are the one desired but do not provide guarantees on the **attitude**, i.e. that the pose of the airframe is a specific one $\{\overline{\phi}, \overline{\theta}, \overline{\psi}\}$
- Moreover, starting from an horizontal pose $\phi = 0, \theta = 0$, if a "glitch" moves suddenly the pitch angle to $\theta = 10^\circ$, the system is **stable** from the rate control point of view (there are no rotations), but (probably) the attitude is not the desired one
- An **Attitude Controller** is required, which drives the Rate Controllers and ensures the respect of a certain pose, from the Euler angles point of view
- To this aim, the **current attitude** is **estimated** by integrating data from *gyros*, *accelerometers* and *magnetometers*.

Attitude Control: Basic Schema



- **Attitude Controllers** are (usually) simple **P** controllers
- Outputs are **saturated** to a maximum angular speed, determined experimentally
- **Attitude Control** module is implemented as a **periodic task** (like the Rate Control), with a period same as to (or greater than) that of Rate Control

Attitude Estimation

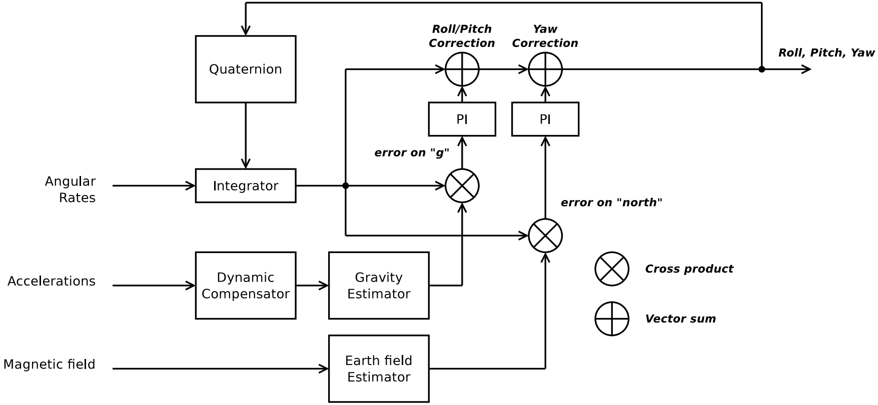
Attitude Estimator

- The most critical part of Attitude Control is the **Sensor Fusion** algorithm that implements the **Attitude Estimator**
- The literature reports a plethora of solutions:
  - Kalman Filters
  - Complementary Filters
  - Direction Cosine Matrix Algorithm
  - Gradient Descend
  - ...
- (Some) quality factors of the estimator:
  - **Resilience to vibrations**
  - **Difference w.r.t. the real attitude**
  - **Rate of convergence**

Attitude Estimator: Basics

- The basic working scheme of the estimator is the following:
  1. Wait sampling period
  2. Update Euler angles using data from gyroscopes by performing discrete integration
  3. Adjust Pitch and Roll on the basis of data from accelerometers ($\vec{g}$ vector)
  4. Adjust Yaw on the basis of data from magnetometers ($\vec{N}$ vector)
- The various algorithms differ in the way in which adjustments (steps 2 and 3) are performed

The Complementary Filter Algorithm



- The DCM is the **rotation matrix** from "Earth reference" and "Body reference" of a rigid body whose attitude is expressed by means of Euler angles θ, ϕ, ψ

Direction Cosine Matrix

$$DCM = \left(\begin{array}{ccc} c\theta c\psi & s\phi s\theta c\psi - c\phi s\psi & c\phi s\theta c\psi + s\phi s\psi \\ c\theta s\psi & s\phi s\theta s\psi + c\phi c\psi & c\phi s\theta s\psi - s\phi c\psi \\ -s\theta & s\phi c\theta & c\phi c\theta \end{array} \right)$$

$$s = \sin, \quad c = \cos$$

- A vector $v' = (x', y', z')$ in **local (body)** frame is translated into **global (Earth)** frame by multiplying it by the DCM: $v = DCM \cdot v'$

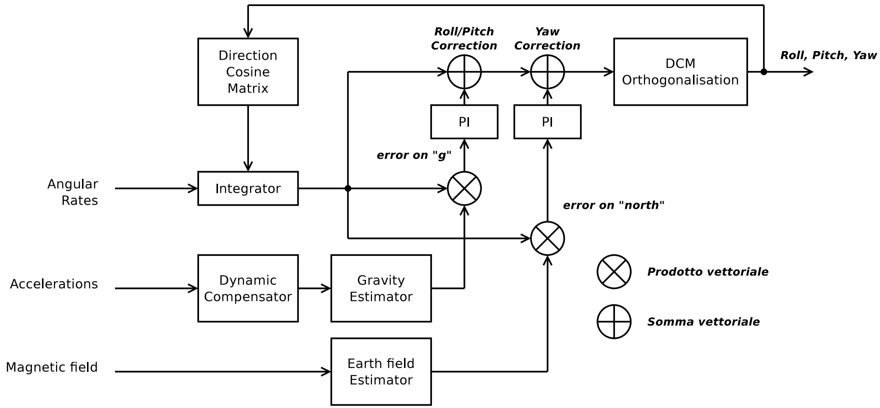
Direction Cosine Matrix

- The DCM has some properties
- It is **orthogonal**, its transpose is equal to its inverse:

$$
\begin{aligned}
DCM^{-1} &= DCM^{T} \\
DCM \cdot DCM^{T} &= I
\end{aligned}
$$

- Orthogonality implies that the columns (rows)
  - are orthogonal vectors in-pair $\rightarrow$ their **cross product is zero**
  - are vector with module equal to 1
- **Such properties must be always respected**

The DCM Algorithm

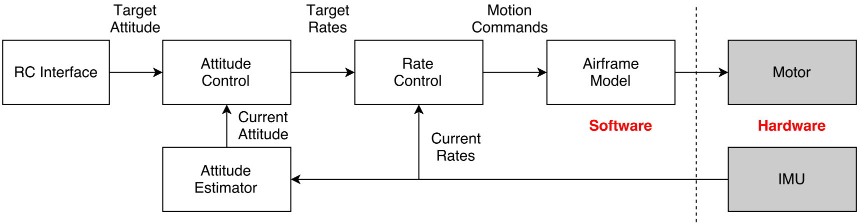


Attitude Estimators: Improving the Quality

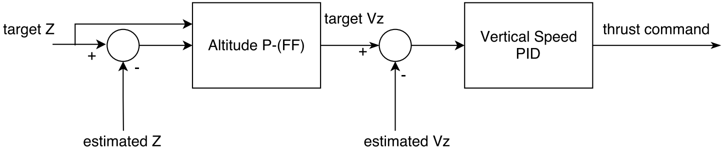
- **Sensor Calibration**
  - **Gyros:** get (and store) **offsets** b_w and remove them from measures: $\overline{\omega} = \omega - b_w$
  - **Accelerometers:** get (and store) **offsets** b_a and rotation matrix R_A remove them from measures: $\overline{a} = R_A \cdot (a - b_a)$
  - **Magnetometers:** get (and store) **offsets** b_m and rotation matrix R_M remove them from measures: $\overline{m} = R_M \cdot (m - b_m)$
- **Sensor Filtering**, low-pass filters (with order ≥ 2) are often used to reduce the effect of motor/propeller vibrations
- **Mechanical Dampers**, to decouple the flight control board from the airframe to reduce the effect of motor/propeller vibrations
- **Dynamic g compensation**, the contribution of accelerometers is weighted according to the error $|g - \|a\||$ in order to reduce the effect, on a, of body translations

Summary of basic Software Modules

In Summary... the Basic Software Modules



Other Kind of Controls



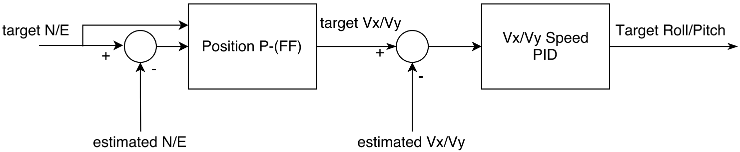
Altitude Control

- **Altitude** Z is estimated by means of proper sensors (barometer, in some case integrated with measures from accelerometers, ultrasonic sensors, etc.)
- The **Vertical Speed** V_z is determined by a derivative of the altitude
- Control is performed by means of two control loops that drive the *thrust command*
  - An inner **PI(D)** speed controller driving the thurst
  - An outer **P-(FF)** position controller driving the speed controller

Position Control

Pose Estimator

- GPS signal is used to determine (and control) the **pose** in the Earth frame
- An **EKF estimator** is used to fuse data from GPS and IMU to estimate:
  - Position $\{X, Y, Z\}$, in NED coordinates
  - Speeds $\{V_x, V_y, V_z\}$
  - Euler Angles $\{\phi, \theta, \psi\}$
- The EKF is complex and CPU-time consuming (in PX4, it is a 22-state estimator)

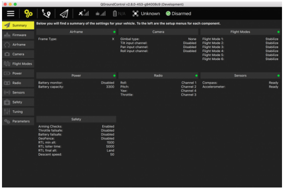


Position Control

- Position Control is performed by means of two pair of control loops (North and East) that drive the *target roll and pitch* of the **attitude controller**
  - An inner **PI(D)** speed controller driving the target attitude (roll and pitch)
  - An outer **P-(FF)** position controller driving the speed controller

Ground Control Station

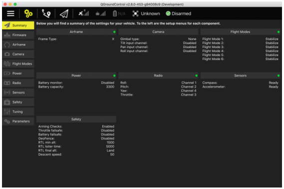
The Ground Control Station



GCS

- Most of the UAV flight stacks have the possibility of connecting a setup, telemetry and maintenance GUI called **Ground Control Station**
- It can be used for
  - Configuring the UAV and calibrating the sensor
  - Monitoring telemetry data
  - Planning the missions
  - Modifying all the parameters (gains of controllers or of the sensor fusion algorithms, sensor configuration, RC commands, etc.)

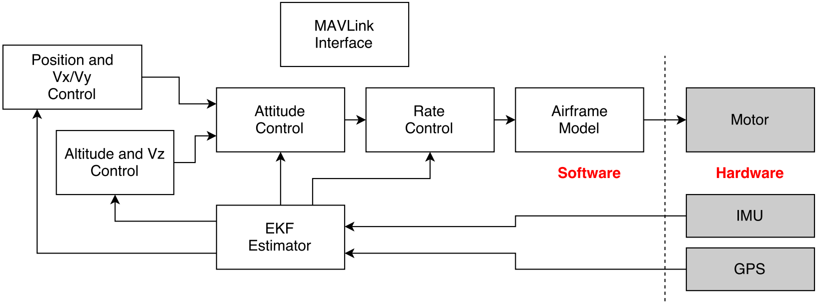
The Ground Control Station



MAVLink

- Communication between GCS and the Flight Stack is performed through a standard protocol called **MAVLink**
- It is designed to be used in **serial links** (wired or radio) or **TCP/UDP** channels
- It can be used not only for GCS-like activities but also to control the UAV through an **external on-board computer** in order to do flight tasks:
  - Arming/Disarming
  - Triggering take-off and land
  - Sending specific set-points (NED positions, or V_x, V_y, V_z speeds)
  - Sending and triggering a mission

Overall Software Modules

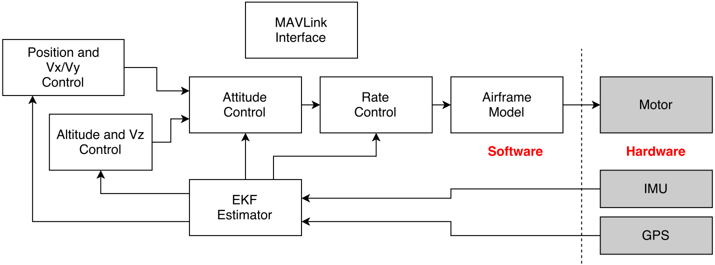


The PX4 Autopilot

The PX4 Autopilot

- The **PX4 Autopilot** is a flight control software designed to drive a large set of *autonomous vehicles*, including ground and aerial platforms
- It is entirely written in C++ and includes two basic parts:
  - **PX4 Flight Stack**: modules that implement control algorithms, estimators, etc. for manual and autonomous flight
  - **PX4 Middleware**: infrastructure for communication among all software modules of the Flight Stack
- PX4 runs on top of **NuttX**, a Unix-like real-time operating system (developed by Gregory Nutt) that provides a support for:
  - pre-emptive thread scheduling
  - device drivers
  - virtual file system
  - a minimal shell

PX4 Middleware



- Since software modules need to interact to each other, a **communication middleware** is provided called **uORB**
- It is based on a **publisher/subscriber** mechanism
- Data is identified by a **topic**, so publishing, subscribing and data copy is handled *"by topic"*
- Structures of messages used in PX4 are defined in some text files in the directory **`msg`**

The **msg** directory

```
...
output_pwm.msg
parameter_update.msg
position_setpoint.msg
pwm_input.msg
sensor_accel.msg
sensor_baro.msg
sensor_gyro.msg
sensor_mag.msg
...
```

The **sensor_mag.msg** file

```
uint64 timestamp
uint64 error_count
float32 x
float32 y
float32 z
float32 range_ga
float32 scaling
float32 temperature

int16 x_raw
int16 y_raw
int16 z_raw

uint32 device_id
```

PX4 Flight Stack

Source Directory Structure

- **drivers**, abstraction layer for physical equipment (sensors, motors, etc.), and specific device drivers for each supported equipment
- **modules**, all software modules performing the control of the vehicle
- **lib**, additional libraries for scalar and matrix math, coordinate system handling, control system modules, filters, etc.

PX4 Driver Layer

Device Drivers

- PX4 **Device drivers** must export a POSIX-compliant interface, with callbacks for functions as *open*, *close*, *read*, *ioctl*
- Devices handled are mainly **sensors** and the current version of PX4 supports a large number of them:

```
./src/drivers:
  gps                     mpu6050__
  hc_sr04                 mpu6500
  hmc5883                 mpu9250
  irlock                  ms5611
  l3gd20                  oreoled
  led                     pca8574
  lis3mdl                 pca9685
  ll40ls                  pwm_input
  lps22hb                 pwm_out_sim
  lsm303d                 sf0x
  lsm6ds33                sf10a
  mb12xx                  srf02
  md25                    srf02_i2c
  meas_airspeed           uart_esc
  mpu6000
  mpu6050
```

Multiple Sensors Handling

- The PX4 firmware is able to use **multiple sensors**, also of the same type (e.g. two or more gyros, accelerometers, magnetometers, etc.)
- Data relevant to the same sensor type are gathered altogether
- A **data quality evaluator** algorithm is employed in order to detect the “best” data read and use it in subsequent computations
- The evaluator is based on comparing the data stream with the same stream filtered by a LPF and computing the error variance

The `modules` directory

- PX4 **modules** are the main control blocks of the autopilot
- Each module is a NuttX task that is started at system startup and implements an infinite loop performing the activities:
  1. waiting for the sampling period
  2. retrieving subscribed data from uORB
  3. executing the specific computation
  4. publishing the result to uORB

PX4 Modules

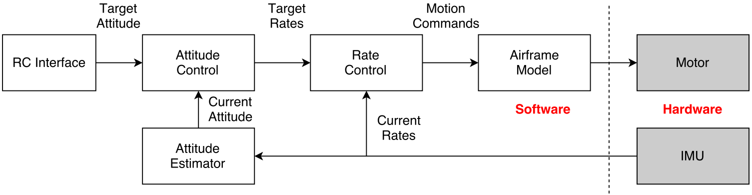
The modules directory

- Modules include:
  - **attitude_estimator_ekf**, EKF for attitude estimation
  - **attitude_estimator_q**, complementary filter for attitude estimation
  - **commander**, sensor calibration routines and GCS command handling (through MAVLink)
  - **ekf_att_pos_estimator**, EKF for position and attitude estimation
  - **fw_att_control**, attitude (and rate) controllers for fixed-wing UAVs
  - **fw_pos_control_l1**, position controllers for fixed-wing UAVs
  - **mavlink**, MAVLink protocol routines
  - **mc_att_control**, attitude (and rate) controllers for multirotor UAVs
  - **mc_pos_control**, position controllers for multirotor UAVs
  - **navigator**, controller for handling autonomous missions
  - **sdlog2**, the logger
  - **segway**, attitude controllers for a segway
  - **systemlib**, system modules, including the **mixer**

The CDrone Flight Stack

CDrone Basics

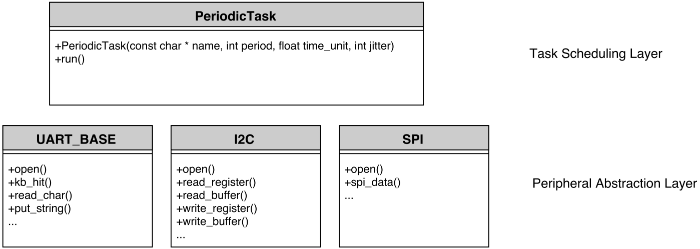
- It is a flight stack developed at the **DMI@UNICT** for educational purposes
- Initially designed in **C** for **Microchip dsPIC33F** family MCUs
- Then rewritten in **C++** and ported to the **STM32 architecture** (STM32F401RE)
- It runs in **bare metal**, with a very light HAL layer written in C++
- It includes only the **basic modules** of a UAV control system (those in figure + the MAVLink interface)



CDrone HAL

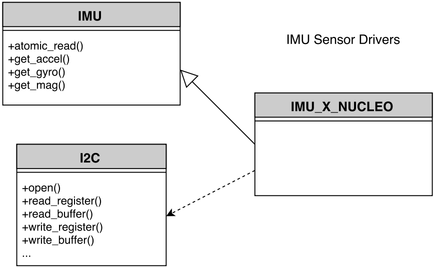
Peripheral and Task Layers

- CDrone is strongly object-based, so everything is defined as a **class**
- The **Peripheral layer** includes some (abstract and concrete) classes each representing a specific peripheral
- The **Task layer** includes the abstract class `PeriodicTask` (the base for the implementation of any periodic task) and the (non preemptive) scheduler
- Scheduling is triggered by a timer running at 400 Hz



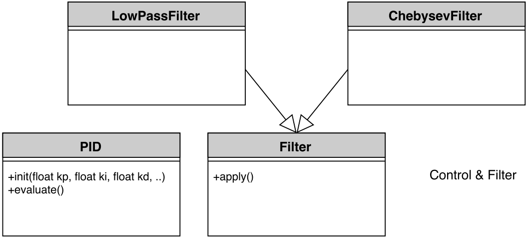
Sensor Drivers

- IMU Sensors are represented by a generic **IMU** abstract class that must be extended into the class that implements the code to handle specific sensors
- A **IMU_X_NUCLEO** class that drives (via I^2C) a X-NUCLEO-IKS01A1 add-on board



Control System Layer

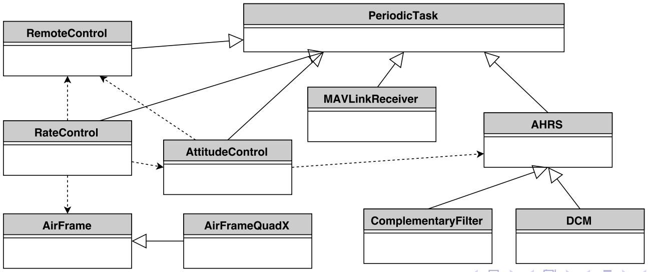
- Some (abstract and concrete) classes implementing
  - the **PID** algorithm (with derivative low-pass filter, anti-wind-up and feedforward)
  - some filters, a 2^{nd} order **LowPass** filter, and a 4^{nd} order **Chebysev** low-pass filter



Control & Filter

CDrone Flight Control Classes

- **RemoteControl**, interface with the RC
- **RateControl**, angular rate control algorithms
- **AttitudeControl**, control algorithms on Euler angles
- **AirFrame**, abstract class representing an airframe
- **AirFrameQuadX**, class representing an X-shaped quadcopter
- **AHRS**, abstract class representing a generic sensor fusion algorithm
- **ComplementaryFilter**, the complementary filter sensor fusion algorithm
- **DCM**, the DCM sensor fusion algorithm
- **MAVLinkReceiver**, MAVLink command handler



CDrone: Statistics

- **Loop-time**: 2.5ms
- **I^{2}C Sensor Polling**: 930μs
- **DCM Sensor Fusion, Rate and Attitude Control**: 410μs without FPU
- **DCM Sensor Fusion, Rate and Attitude Control**: 290μs with FPU
- **Total processing time**: 1.34ms without FPU
- **Total processing time**: 1.22ms with FPU

Control Systems for Multi-rotors
Principles, Modeling and Software Design

Corrado Santoro

ARSLAB - Autonomous and Robotic Systems Laboratory
Dipartimento di Matematica e Informatica - Università di Catania, Italy
santoro@dmi.unict.it