Alma Mater Studiorum · Università di Bologna

Scuola di Ingegneria e Architettura
Campus di Cesena

Corso di Laurea in Ingegneria Elettronica, Informatica e Telecomunicazioni

Sviluppo di una Architettura di Controllo per Velivoli di tipo Multi-Rotore

—

Elaborato in
Controlli Automatici

Relatore:
Dott. Roberto Naldi

Presentata da:
Alessandro Bacceli

Correlatore:
Dott. Nicola Mimmo

III Sessione
Anno Accademico 2013/2014

Al mio Amore e alla mia Famiglia...

Sommario

Con l'aumentare delle tecnologie ci stiamo dirigendo verso un'interazione virale con qualsiasi oggetto in grado di "ragionare" autonomamente. Si sente parlare sempre piú frequentemente di Human-Computer Interaction (HCI) applicato a diversi ambiti: domestico, militare, commerciale, civile, etc. La linea che separa il mondo della robotica da quello umano è sempre meno definita. Molte delle attività che fino ad un decennio fa richiedevano l'utilizzo delle risorse umane, oggi sono facilmente rimpiazzabili con apparati robotici. Quest'ultimi sono in grado di agire autonomamente, interagire e collaborare fra loro e allo stesso tempo prendere comandi da entità superiori non necessariamente umane.

Lo studio di queste tecnologie è molto importante poichè nel futuro il loro utilizzo rivoluzionerà il nostro modo di vivere, di lavorare, di interfacciarci e di comunicare.

Indice

Introduzione

In questo progetto di tesi ho portato avanti un argomento molto interessante e di estrema attualità. Il tema principale è quello della robotica applicata all' ambito dei soccorsi in situazioni di emergenza in ambienti ostili.

Ho avuto la fortuna di collaborare con un team di ricercatori impiegato nello sviluppo del progetto europeo **SHERPA**, gestito in gran parte all'interno del Center for Research on Complex Automated Systems (**CASY**) di Bologna. Il progetto nell'insieme ha come goal quello di sviluppare un'ambiente costituito da diversi strumenti robotici (Helicopter, Quadrotor, Rover...) per supportare la ricerca e il salvataggio in situazioni ostili come ad esempio quelle alpine.

L'Obiettivo all'interno del progetto di cui ho fatto parte è denominato **Obiettivo O1** ed è suddiviso in due parti:

1. gesture recognition;
2. utilizzo pratico e sicuro degli UAV da parte dell'operatore.

Entrambe le parti sono fondamentali per il risultato complessivo del progetto e verranno testate con prove di volo outdoor. La mia esperienza si è focalizzata sul secondo punto dell'Obiettivo O1 approfondendo l'ambito riguardante la sicurezza e il controllo ad alto livello.

Nel resto dell'elaborato farò spesso riferimento alla parola "sistema" dove per essa intendo il sistema implementato relativamente al secondo punto dell'Obiettivo O1.

Capitolo 1

Tecnologie di Base del Sistema

Prima di iniziare la trattazione di come il sistema è stato suddiviso, di come i suoi componenti interagiscono e di come essi si comportano, è importante conoscere le macro-tecnologie che compongono il progetto in modo da rendere la comprensione dei seguenti capitoli il più scorrevole possibile.

Il sistema che tratteremo si basa su due sottosistemi interagenti fra loro:

- **sistema volante**: l'Unmanned Aerial Vehicle (UAV);
- **sistema di terra**: Ground Control Station (GCS) + operatore umano.

Il sistema volante è formato da:

- mezzo aereo (plane, quadcopter, helicopter...);
- servizi di bordo: tutto ciò che permette al mezzo di volare (sistema di potenza elettrica, controllore, attuatori, eliche, servocomandi, ricevente);
- carichi paganti (payload): tutti gli strumenti aggiuntivi che non servono direttamente per far volare il mezzo aereo ma servono a compiere dei task per la missione specifica del mezzo (videocamera, camera termica, artva, sensori ambientali).

Il sistema di terra è formato da:

- sottosistemi di controllo (joypad, tastiera, RC pad...);
- sottosistemi di lettura e rappresentazione dati (supporti telemetrici, Ground Control Station...).

Figura 1.1: UAV di tipo Quadrotor.

Per far sì che tutto funzioni a dovere, c' è bisogno che tutte queste parti interagiscano fra loro e si coordinino a dovere. Considerando la complessità del sistema e la presenza di almeno un umano all'interno, bisogna progettare il sistema in maniera robusta per gestire condizioni critiche come quelle ambientali (neve, vento...) o dello stesso operatore (stanchezza, distrazioni...).

Definiremo meglio in questo capitolo i due blocchi più complessi all'interno del sistema ovvero gli Unmanned Aerial Vehicle (UAV) e la Ground Control Station (GCS) anche se nel resto della trattazione ci concentreremo quasi esclusivamente sullo studio del primo.

1.1 UAV: Unmanned Aerial Vehicle

1.1.1 Definizione

Un **Unmanned Aerial Vehicle (UAV)**, spesso chiamato anche drone, è un velivolo caratterizzato dall'assenza del pilota umano a bordo. Solitamente con questo termine ci si riferisce ad un oggetto formato da componenti sia hardware che software in grado di essere pilotato in remoto tramite dei supporti telemetrici e allo stesso tempo, ad esempio in caso di assenza di comandi, in grado di autopilotarsi grazie a delle leggi di controllo codificate all'interno del software.

1.1.2 Nascita, Impieghi e Tipologie

La necessità di avere a disposizione un oggetto in grado di essere pilotato a distanza nasce per scopi militari durante la Prima Guerra Mondiale nel 1916 con la progettazione di un prototipo denominato "Aerial Target" comandato con tecniche di radio controllo abbastanza primitive e basato su un sistema di giroscopi su esso montato. Questi oggetti, in questo periodo storico, erano considerati delle vere e proprie "bombe volanti" utilizzate per attaccare gli eserciti nemici.

In seguito, durante la Seconda Guerra Mondiale, cominciarono ad essere progettati aeroveicoli autocontrollati, più precisi dei primi, utilizzati come artiglieria antiaerea i quali vennero perfezionati a loro volta per affrontare sia la Guerra Fredda che quella del Vietnam.

Nel frattempo, con l'evolversi delle tecnologie, la crescita esponenziale della microelettronica e la relativa diminuzione dei costi dei componenti elettronici sempre più prestanti, gli UAV hanno acquisito moltissimi altri impieghi. Ad esempio l'ambito più comune è quello civile: essi sono impiegati per il controllo e la manutenzione delle coltivazioni (soprattutto nelle regioni Asiatiche), per la salvaguardia del territorio, per i soccorsi in scenari critici (tema principale all'interno del progetto SHERPA) e per qualsiasi tipo di monitoraggio. Ovviamente le potenzialità di questi oggetti sono infinite e nei prossimi anni subiranno una netta crescita in tutto il mondo. Non a caso tutte le maggiori compagnie internazionali hanno cominciato a puntare verso questo settore dalle mille utilità (come esempio non si può non citare il già noto Prime Air di Amazon, il drone in grado di consegnare pacchi in pochissimo tempo).

Solitamente gli UAV sono catalogati in base a dei parametri quali la quota di volo, la durata del volo e il peso massimo al decollo. Le suddivisioni inoltre vengono fatte in base al contesto nel quale essi dovranno operare: andiamo dall'UAV di piccole dimensioni, molto leggero che raggiunge quote molto basse con una scarsa autonomia di volo all'UAV molto grande e pesate in grado di raggiungere quote più elevate con a bordo diverse tipologie di carichi paganti.

1.2 GCS: Ground Control Station

1.2.1 Definizione ed Utilizzi

Il termine **Ground Control Station (GCS)** indica una struttura hardware/software in grado di controllare gli UAV a distanza. Le tecnologie alla base di una GCS sono del tutto generali: possiamo immaginarci di avere

a disposizione un qualsiasi calcolatore in grado di computare una legge di controllo ed inviare, tramite un supporto telemetrico, dei comandi all'UAV.

Questa definizione del tutto generica ci permette di lavorare su un qualsiasi tipo di tecnologia. Oggi, grazie allo sviluppo tecnologico di tablet e smartphone, abbiamo la possibilità di avere GCS a portata di mano con una potenza computazionale non indifferente.

Una GCS sarà progettata in base al contesto e alla struttura del sistema con il quale si interfaccia: se abbiamo un sistema molto distribuito, essa dovrà stabilire una connessione molto veloce ed affidabile con l'UAV. Inoltre se abbiamo un sistema progettato per l'uso comune, essa dovrà risultare user-friendly ed astrarre il più possibile dai dettagli tecnici. Ovviamente questo porta ad una gestione automatizzata di molti aspetti tecnici da parte del main program sulla GCS il quale deve comunque garantire un corretto volo all'UAV.

La NATO nel 2002, tramite lo standard STANAG 4586, ha definito delle regole riguardanti i seguenti aspetti:

1. equipaggiamento di bordo e degli operatori;
2. metodi e tempistiche di accesso ai controlli del drone;
3. parametri minimi e opzionali da tenere sempre a video sulla GCS;
4. operazioni minime eseguibili;
5. qualificazioni minime degli operatori.

Sicuramente il secondo, il terzo e il quarto punto sono fondamentali nella progettazione di un sistema GCS/UAV. Tutti i software che fungono da GCS in commercio si basano su queste regole. Tra loro troviamo software progettati a scopo militare ma anche a scopo "hobbistico" ed open source.

1.2.2 GCS all'interno del Sistema

Il nostro sistema utilizza la GCS come pannello di controllo per avere a disposizione in ogni momento i dati principali dell'UAV. Per controllare l'UAV abbiamo a disposizione un radiocomando (RC transmitter). L'accoppiamento dei due ci fornisce contemporaneamente il comando e un feedback di come sta reagendo il velivolo.

Solitamente i dati interessanti durante una prova di volo sono i seguenti:

- Altitudine;
- Angoli di Pitch, Roll e Yaw;

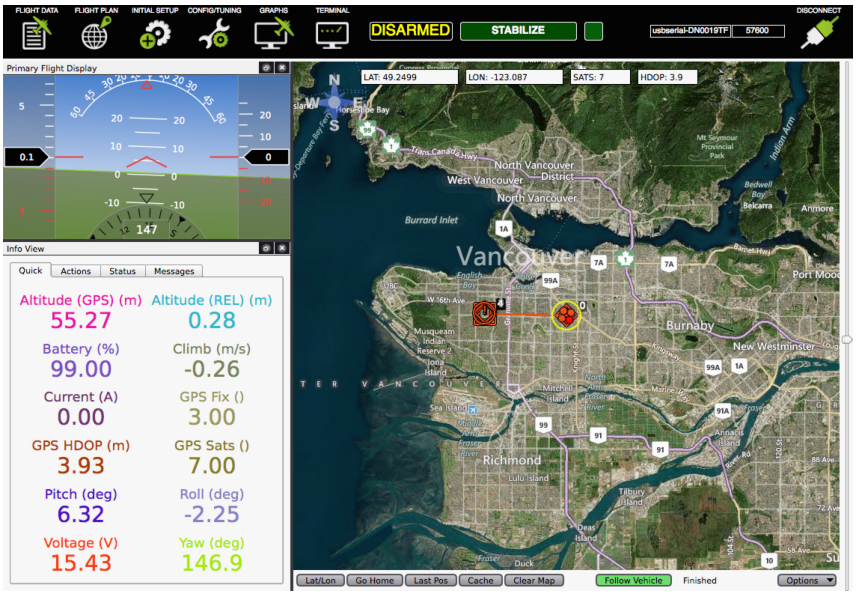

Figura 1.2: Interfaccia principale dell'APM Planner.

- Velocità di salita (Climb);
- Voltaggio;
- Corrente;
- Livello della Batteria;
- Posizione GPS.

A nostro supporto abbiamo un software chiamato **APM Planner** che può fornire molte altri dati rispetto a quelle appena citati. Ad esempio abbiamo a disposizione grafici in tempo reale dei parametri fondamentali in funzione del tempo, cosa molto comoda per avere uno storico abbastanza dettagliato soprattutto nelle fasi di test. Questo software ha anche delle funzionalità interessanti che ci permettono di comandare ad alto livello l'UAV. Nel nostro caso non faremo molto affidamento su di esse poichè dovremmo crearle noi ad hoc per le nostre esigenze. Il software APM Planner è molto utile inoltre per diversi aspetti di settaggio iniziale come la calibrazione del magnetometro, dell'accelerometro, degli ESC (Electronic Speed Controller), dell'RC transmitter, la sincronizzazione fra le due radio di comunicazione... All'interno dello stesso software vi è anche una sezione per caricare il firmware specifico all'interno del velivolo che stiamo utilizzando.

Siccome gli algoritmi di controllo utilizzati all'interno del software APM Ardupilot non tengono conto delle problematiche affrontate all'interno del progetto SHERPA, potrebbe far comodo un meccanismo in grado di comandare real time ad alto livello l'UAV eseguendo degli algoritmi progettati da noi direttamente sul calcolatore sul quale stiamo lavorando (che fungerà appunto da GCS).

Nel resto della trattazione, la GCS sarà utilizzata come un'entità passiva in grado di ricevere dati dell'UAV senza inviare ad esso nessun tipo di comando.

Capitolo 2

Il Sistema

Il sistema di cui parleremo fa riferimento a quello dell'UAV, sviluppato finora per l'obiettivo O1 del progetto SHERPA. Esso è molto complesso dal momento che è costituito da diversi componenti hardware/software che devono comunicare fra loro. Per lo sviluppo di un sistema per UAV utilizzabile in ambienti ostili, come ad esempio quelli alpini, è richiesto:

- un Sistema Operativo real time o un meccanismo che faccia in modo di porre dei limiti temporali;
- algoritmi di controllo in grado di gestire scenari critici (vento, neve...);
- una struttura sensoriale (GPS, giroscopio, accelerometro, barometro, magnetometro) molto dettagliata e precisa;
- tempi di trasmissione dei segnali (comandi, dati...) molto brevi;
- diversi sistemi di sicurezza;

In questo capitolo analizzeremo le scelte che abbiamo preso per avere a disposizione una struttura robusta, in grado soddisfare i precedenti requisiti.

2.1 Architettura

Il sistema di controllo dell'UAV è stato suddiviso in due moduli che abbiamo definito di **basso** ed **alto livello**.

È bene notare che si è scelto un sistema modulare per avere maggior flessibilità in caso di modifiche architetturali in progetti futuri: ad esempio si potrebbe spostare il modulo di alto livello su un Rover di terra e farlo comunque comunicare via canale radio con il modulo di basso livello sull'UAV senza compromette il corretto funzionamento del sistema.

Un altro punto a favore dell'architettura modulare è che avendo a disposizione due schede hardware distinte, guadagnamo in termini di efficienza. Con questa architettura è molto conveniente dividere il carico spezzando la catena di controllo sulle due piattaforme in due parti e, tramite la loro interazione, potremmo chiudere il loop di controllo.

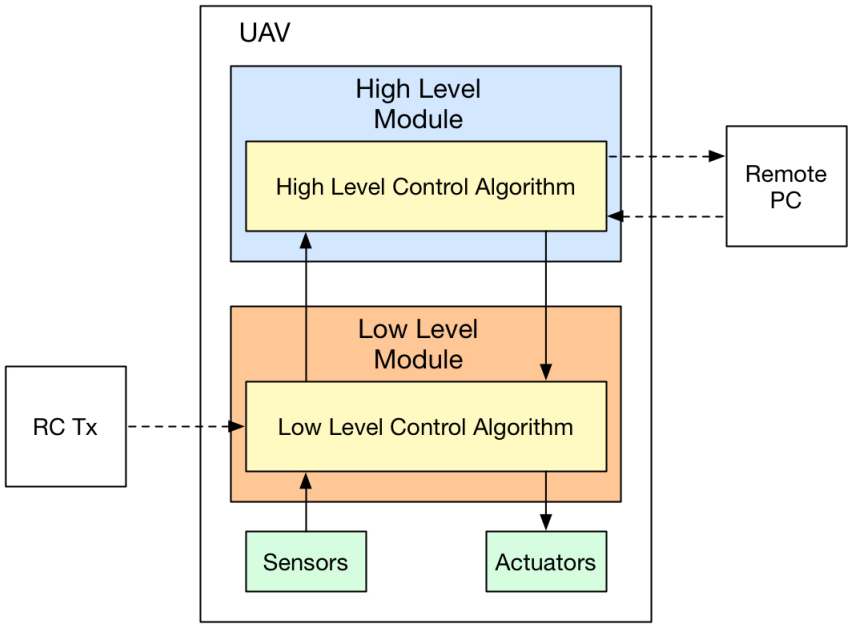

Figura 2.1: Architettura logica del sistema.

In Figura 2.1 abbiamo un modello logico del sistema che verrà analizzato nel dettaglio nelle seguenti sottosezioni.

2.1.1 Modulo di Controllo di Basso Livello

Basandoci sulla Figura 2.1 possiamo descrivere il **modulo di controllo di basso livello** come un blocco responsabile di:

- eseguire parte dell'algoritmo di controllo partendo dai dati acquisiti dal modulo di alto livello;
- gestire gli output verso gli attuatori;
- ottenere un feedback dai sensori e trasferirli al modulo di alto livello.

Per compiere le precedenti operazioni abbiamo bisogno di una buona scheda hardware con sopra un applicativo software integrato con un sistema operativo real time o, come già detto, in grado di porre dei limiti temporali ai task. Tra i vari hardware/software disponibili in commercio nel settore specifico degli UAV abbiamo optato per:

- **Scheda Hardware: Pixhawk**;
- **Sistema Operativo Real Time: PX4-NuttX**;
- **Software di Controllo: APM Ardupilot**.

2.1.1.1 Pixhawk

La scheda **Pixhawk** è stata progettata dall' **open hardware development team**. È stata realizzata prevalentemente per essere integrata all'interno di velivoli a pilotaggio remoto.

Figura 2.2: Scheda Pixhawk.

Essa è composta dalla seguenti caratteristiche tecniche:

- Microprocessore a 32-bit STM32F427 Cortex M4 core:

  - FPU: 168 MHz;
  - RAM: 256 KB;
  - Flash: 2 MB ;
  - Failsafe Co-Processor: 32 bit STM32F103.

- Sensori:
  - Giroscopio ST Micro L3GD20 3-axis 16-bit;
  - Accelerometro/Magnetometro ST Micro LSM303D 3-axis 14-bit;
  - Accelerometro/Magnetometro Invensense MPU 6000 3-axis;
  - Barometro MEAS MS5611.

- Interfacce:
  - UART 5x;
  - I2C;
  - SPI;
  - CAN 2x;
  - Spektrum DSM/DSM2/DSM-X Satellite;
  - S.BUS Futaba;
  - PPM sum signal;
  - RSSI (PWM o Voltage) input;
  - ADC inputs a 3.3V e 6.6V;
  - Porta MicroUSB.

Il suo grande numero di interfacce di I/O e le sue buone caratteristiche hardware ci sono d'aiuto nello sviluppo dei nostri progetti. Inoltre, supportando il multithreading, possiamo sfruttare a pieno le funzionalità offerte dal software APM Ardupilot. Altri punti a favore della Pixhawk sono il peso (38g) e le dimensioni (81.5 x 50 x 15.5 mm): queste caratteristiche sono ideali per un velivolo aereo.

2.1.1.2 PX4-NuttX ed APM Ardupilot

Il **PX4-NuttX** e l'**APM Ardupilot** sono rispettivamente un sistema operativo real time (RTOS) e un software di basso livello per veicoli autopilotati (sia aerei che di terra). Ora parleremo degli RTOS in generale e del NuttX. Trascureremo momentaneamente il software APM Ardupilot dal

momento che giocherà un ruolo fondamentale nel prossimo capitolo quando parleremo del progetto di un sistema di sicurezza relativo all'obiettivo O1 del progetto SHERPA.

I **real time operating system (RTOS)** sono sistemi operativi che si basano su dei vincoli di tipo temporale per svolgere determinati calcoli. In ambito industriale/robotico si fa pervasivamente uso di sistemi operativi real time dal momento che la maggior parte delle operazioni in gioco possono essere considerate accettabili solo entro un certo limite di tempo. Nel caso degli UAV, in primo luogo per una questione di sicurezza, ne abbiamo bisogno.

L'RTOS presente sulla Pixhawk e PX4-NuttX, una leggera modifica del più noto NuttX. Quest'ultimo è stato rilasciato nel 2007 da Gregory Nutt sotto licenza BSD. Tale sistema e stato realizzato in modo da essere compatibile con gli standard POSIX e ANSI e presentarsi allo stesso tempo leggero, scalabile e altamente configurabile. Oltre agli standard citati sono state integrate API da Unix ed acluni sistemi RTOS, per aggiungere funzionalita altrimenti non disponibili (come, ad esempio, la funzione fork()). Pur non supportando il concetto di processi cosi come e inteso in Linux, NuttX prevede la possibilita di creare task groups composti da un main task thread e da tutti i pthreads, creati a loro volta o dal main thread o da altri pthreads. I membri di un task group possono condividere alcune risorse per operare operazioni concorrenti.

Inoltre è un sistema adatto principalmente ai microcontrollori dal momento che ha un'architettura scalabile (supporto di sistemi dagli 8 ai 32 bit). Un altro punto a favore è sicuramente il fatto che è open source ed estremamente configurabile.

2.1.2 Modulo di Controllo di Alto Livello

Basandoci nuovamente sulla Figura 2.1, possiamo considerare il modulo di controllo di alto livello come un entità con i seguenti obiettivi:

- eseguire un algoritmo di controllo basandosi sui riferimenti che gli sono stati passati dal calcolatore remoto;
- passare il valore di uscita dall'algoritmo di controllo al modulo di basso livello;
- ottenere un feedback dal modulo di basso livello per chiudere il loop di controllo ed eventualmente comunicarlo al calcolatore remoto.

Anche in questo caso abbiamo bisogno di una buona scheda hardware con un sistema non necessariamente real time. Per questo modulo si è pensato di usare i seguenti componenti HW/SW:

- **Scheda Hardware: Odroid U3**;
- **Sistema Operativo: Lubuntu**;
- **Meta-Sistema Operativo: ROS**.

2.1.2.1 Odroid U3

Odroid U3 è una scheda hardware che appartiene alla categoria di schede denominata "single-board computer" progettata dalla Hardkernel Company. Infatti, grazie alle sue caratteristiche tecniche, è in grado di far girare un sistema operativo desktop come ad esempio una qualunque distribuzione Linux. Il nome "Odroid" deriva dall'accoppiamento della parola "Open" e "Android" che allude alla possibilità di avere un'architettura hardware open source in grado di montare il sistema operativo Android. In realtà il nome è fuorviante poichè molte caratteristiche del progetto hardware non sono pubbliche ma in possesso delle compagnie che hanno progettato la scheda. Allo stesso tempo, la scheda è usata in diversi ambiti che richiedono l'uso di un sistema operativo non mobile (come lo è per l'appunto Android) ma desktop.

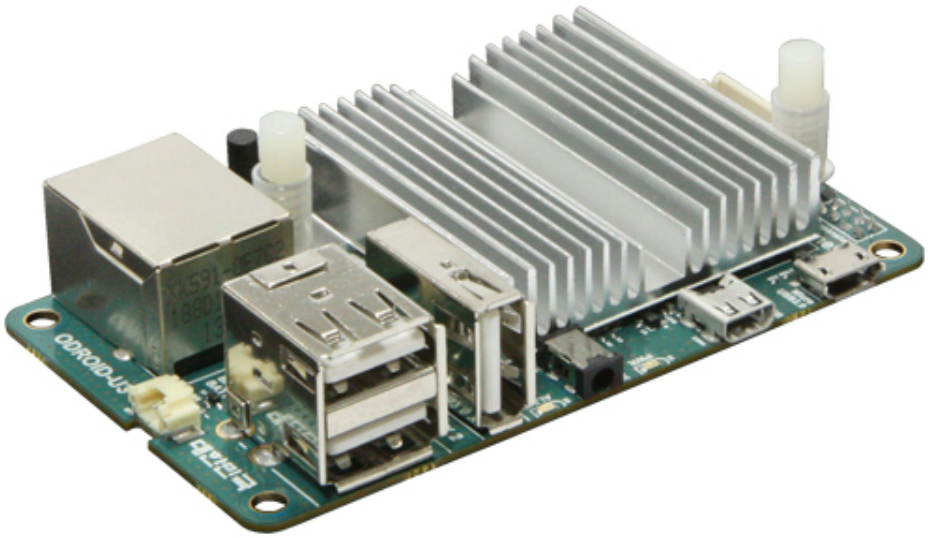

Figura 2.3: Scheda Odroid U3.

Elenchiamo brevemente le caratteristiche tecniche:

- CPU Exynos 4412 Prime quad-core a 1.7GHz;
- GPU Mali-400 MP4 quad-core a 533 MHz;

- 2GB di memoria RAM;
- MicroSD card slot e modulo eMMC;
- tre porte USB 3.0 e una porta Ethernet;
- porta micro HDMI;
- GPIO;
- UART, I2C, SPI;
- ADC.

Questo modulo sarà fondamentale per il nostro progetto poichè sarà la piattaforma fisica sulla quale gireranno le leggi di controllo ad alto livello.

Il sistema operativo che abbiamo installato è **Lubuntu**. Quest'ultimo è la versione più leggera tra le distribuzioni Linux in grado di utilizzare il minimo delle risorse hardware in modo da permettere una maggiore autonomia della batteria che alimenta la piattaforma Odroid.

2.1.2.2 ROS

Il **Robot Operating System (ROS)** è un framework ampiamente utilizzato nell'ambito della robotica che fornisce molte funzionalità che lo accomunano ad un sistema operativo (come ad esempio l'astrazione hardware, il controllo a basso livello dei dispositivi fisici, l'implementazione di funzionalità utili a servizi applicativi, la possibilità di un'interazione a "message-passing" tra i processi e la gestione di pacchetti). Il ROS è stato progettato per essere integrato su qualsiasi dispositivo fisico con sistema operativo Unix-based. A discapito di quanto si possa pensare, ROS è un meta-sistema operativo non real-time. Fortunatamente vi è la possibilità di integrare ROS con del codice in esecuzione in modalità real-time.

ROS può essere suddiviso in due parti:

1. **Core**: contiene le API per creare delle reti ROS e tutte le funzionalità che lo accomunano ad un sistema operativo. Sono inoltre contenuti in esso degli script che permettono il monitoraggio dei nodi, delle connessioni e dei messaggi all'interno della rete ROS;
2. **Packages**: sono dei moduli costruiti sopra il core, in grado di fornire diverse funzionalità. Solitamente con questo termine si fa riferimento ai nodi ROS, alle librerie di funzioni, ai files di configurazione, ai software di terze parti...

La parte principale che rende ROS un sistema molto utilizzato nel mondo della robotica è la sua architettura di comunicazione. Quest'ultima è basata sul protocollo TCP/IP. Ora cerchiamo di comprendere come avviene la comunicazione su ROS e quali sono le entità interagenti:

- **Nodi**: sono dei processi che possono eseguire qualsiasi tipo di task e scambiare informazioni con gli altri task mediante la rete ROS. Inizialmente essi si registrano ad una rete con un id univoco, dichiarando una lista di topics e servizi che hanno intenzione di utilizzare (scambiando messaggi). ROS fornisce le API necessarie per sviluppare nuovi nodi sia in C++ che in Python;

- **Master**: è un nodo speciale che viene lanciato quando viene inizializzata la rete ROS. Esso è responsabile di gestire le registrazioni dei nodi con relativi topic e servizi richiesti dal nodo in questione.

- **Messaggi**: i pacchetti che viaggiano nella rete ROS sono definiti come "messaggi ROS". Essi sono semplicemente delle strutture di dati con determinati campi tipizzati. I tipi supportati da ROS sono: booleani, unsigned e signed int (8/16/32 bits), float (32/62 bits), string, time, duration. Si possono inoltre creare delle strutture di dati contenenti questi dati primitivi.

- **Topics**: sono un sistema di trasporto basato sulla semantica di "publish and subscribe" usata per inviare messaggi. Un nodo può collegarsi ad un topic e decidere se essere un "publisher" (mittente di informazioni) o "subscriber" (destinatario di informazioni). I topic non hanno un limite di iscrizioni quindi possono accettare un numero indefinito di nodi. Proprio per come è definito il meccanismo "publish and subscribe", i nodi "publisher" non possono avere informazioni riguardo i "subscriber" e viceversa. Inoltre ogni nodo iscritto ad un determinato topic è in grado di ricevere le informazioni pubblicate da un nodo "publisher" nello stesso topic. Per interpretare correttamente i dati provenienti dai nodi, ogni topic deve definire uno standard per i messaggi da inviare e ricevere che deve essere rispettato;

- **Services**: sono un sistema molto simile ai topics ma basati su un'altra semantica denominata "request/response". In questo caso un nodo può esclusivamente ricevere informazioni da un service attraverso una richiesta. Solitamente i services sono forniti dai nodi stessi i quali possono rispondere ad una richiesta per volta. Gli standards per i messaggi di richiesta e le risposta ammissibili sono ugualmente definiti all'interno dei topics.

Per comprendere meglio come viene gestita l'interazione all'interno del sistema ROS basata su **topics**, prendiamo come esempio la figura seguente:

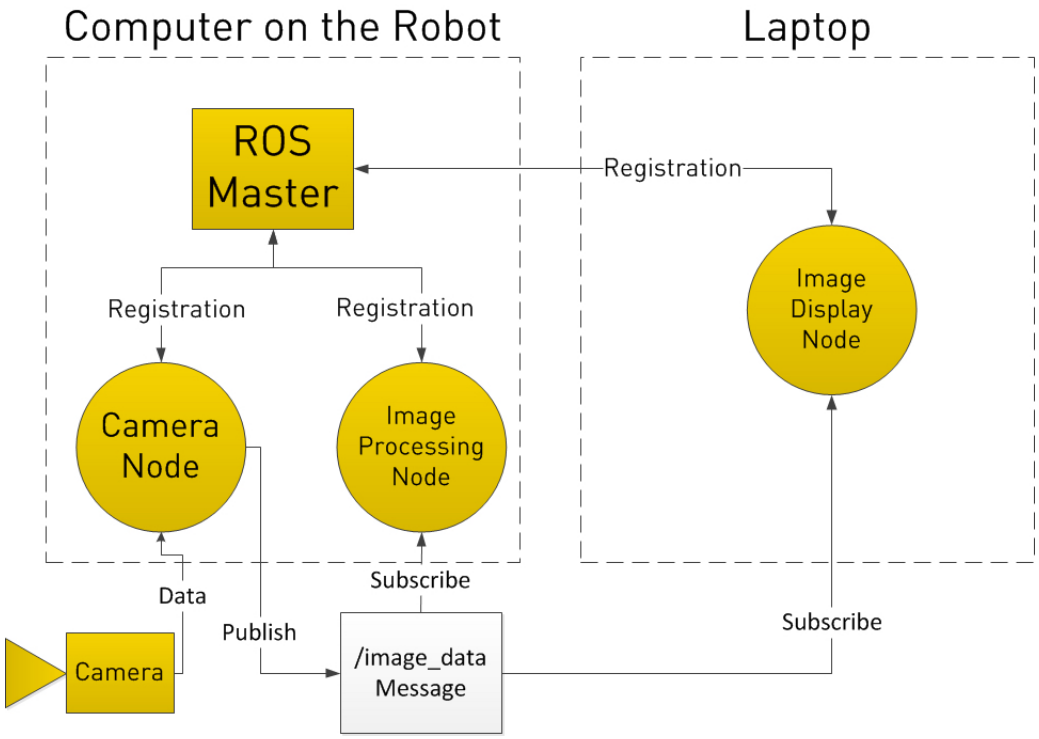

Figura 2.4: Interazione in ROS.

Facendo riferimento alla Figura 2.4 otteniamo il seguente scenario:

- sull'entità denominata "computer on the robot" viene lanciato inizialmente il nodo master;
- vengono mandati in esecuzione altri tre nodi con funzionalità differenti:
  - il primo nodo sul robot viene usato per gestire i dati relativi da una videocamera;
  - il secondo nodo sul robot viene usato per processare le immagini provenienti dalla videocamera;
  - il terzo nodo collocato su un laptop remoto viene usato per visualizzare le immagini provenienti dalla videocamera.
- tutti e tre i nodi si registrano presso il master node facendo riferimento allo stesso topic, ma con compiti diversi:

  - il primo nodo si registra come “publisher”;
  - gli altri due nodi si registrano come “subscriber”.

- nel topic viene definito come deve essere strutturato il messaggio;
- quando il primo nodo acquisisce i dati dalla videocamera, li pubblica attraverso il topic attraverso un messaggio conforme al relativo standard;
- il topic consegna i messaggi a tutti i nodi iscritti allo stesso;
- sia il secondo nodo che il terzo ricevono il messaggio contenente i dati della videocamera;
- ognuno, attraverso un suo comportamento interno, garantirà le funzionalità offerte.

Per comprendere invece come viene gestita l’interazione all’interno del sistema ROS basata sui **services**, possiamo immaginarci un modello client/server come quello utilizzato per la rete:

- un nodo “client” effettua una richiesta ad un nodo “server”, registrato come fornitore di un certo servizio;
- il nodo “server” risponde alla richiesta;
- i messaggi scambiati tra nodo “client” e nodo “server” devono essere conformi ad uno standard da definire.

La conoscenza e la padronanza con il framework ROS è indispensabile per gestire ad alto livello qualsiasi apparato robotico.

2.2 Interconnessione e Comunicazione

Per questo progetto abbiamo a disposizione un UAV di tipo Quadcopter della 3DR denominato **IRIS+**.

Figura 2.5: IRIS+ UAV Quadcopter.

Tramite le guide sul sito dell'APM siamo riusciti ad interconnettere tutti gli apparati fisici dell'IRIS+ al nostro modulo di basso livello (Pixhawk). Lo step successivo è stato quello di trovare un modo per far comunicare il modulo di basso livello con quello di alto livello e viceversa, utilizzando la porta seriale USB. Per fare ciò dobbiamo prima conoscere il protocollo di comunicazione MAVLINK.

2.2.1 MAVLINK

MAVLINK (Micro Air Vehicle Link) è un protocollo di comunicazione destinato ad UAV di piccole dimensioni. I pacchetti utilizzati nella comunicazione MAVLINK sono strutture in linguaggio C, trasmesse molto efficientemente attraverso dei canali di comunicazione seriale. I punti a suo favore sono sicuramente la velocità di comunicazione, la facilità con cui si possono realizzare nuovi messaggi ed il fatto che è open source. Per quanto riguarda la possibilità di creare nuovi messaggi, MAVLINK permette la definizione mediante file XML (chiamati "dialetti"), i quali vengono convertiti in codice sorgente in diversi linguaggi in base alle esigenze.

Ogni pacchetto MAVLINK è costituito da un massimo di 17 bytes (6 bytes per l'header, 9 bytes per il payload, 2 bytes per il checksum) ed è così strutturato:

- **Header**: intestazione del messaggio contenente i seguenti bytes
  - byte 1: **start byte** (sempre 0xFE);
  - byte 2: **payload length**, ovvero la lunghezza del messaggio vero e proprio;
  - byte 3: **sequence number**, serve per numerare i pacchetti e riordinarli lato ricevente. È utilizzato anche per rilevare eventuali pacchetti persi;
  - byte 4: **system ID**, serve per identificare il sistema che ha inviato il messaggio;
  - byte 5: **component ID**, serve per identificare il componente del sistema che ha inviato il messaggio;
  - byte 6: **message ID**, serve per identificare il tipo di messaggio;
- **Payload**: messaggio vero e proprio con lunghezza variabile, estendibile fino a 9 bytes;
- **Checksum**: 2 bytes utilizzati per il rilevamento degli errori.

I bytes di nostro maggiore interesse sono relativi al *message ID* ed ovviamente il *payload*. Analizziamo ora i tipi di messaggi più frequenti, implementati per il software APM Ardupilot, che vengono scambiati attraverso il protocollo MAVLINK prendendo come esempio la comunicazione fra il modulo di alto livello ed il modulo di basso livello :

- **MAVLINK_MSG_ID_HEARTBEAT (ID = 0)**: messaggio inviato molto frequentemente dall'alto livello per tener sotto controllo lo stato della connessione con il basso livello. L'alto livello potrebbe inviare un comando di failsafe se non riceve delle risposte costanti ai messaggi di heartbeat;
- **MAVLINK_MSG_ID_SYS_STATUS (ID = 1)**: messaggio inviato al basso livello per ottenere lo stato del sistema (LOCKED, MANUAL, GUIDED, AUTO);
- **SET_MODE (ID = 11)**: messaggio inviato al basso livello per specificare una modalità di volo da utilizzare;
- **MAVLINK_MSG_ID_MISSION_REQUEST_LIST (ID = 43)**: messaggio inviato al basso livello per comunicargli una serie di posizioni da raggiungere;

- **MAVLINK_MSG_ID_MISSION_REQUEST (ID = 40)**: messaggio inviato al basso livello per comunicargli un'unica posizione da raggiungere;
- **MAVLINK_MSG_ID_PARAM_REQUEST_READ (ID = 20)**: richiesta inviata al basso livello per ottenere alcuni parametri (altitudine, coordinate GPS, stato della batteria...);
- **MAVLINK_MSG_ID_PARAM_REQUEST_LIST (ID = 21)**: richiesta inviata al basso livello per ottenere tutti i parametri;
- **MAVLINK_MSG_ID_RC_CHANNELS (ID = 65)**: richiesta inviata al basso livello per ottenere i segnali PPM ricevuti dal trasmettitore RC suddivisi in canali;
- **MAVLINK_MSG_ID_RC_CHANNELS_OVERRIDE (ID = 70)**: messaggio inviata al basso livello per sovrascrivere i dati provenienti dal trasmettitore RC con quelli presenti nel payload dello stesso messaggio.

Ovviamente ve ne sono molti altri interessanti che possono essere trovati con una buona documentazione sul sito di riferimento del protocollo MAVLINK. Nel prossimo capitolo lavoreremo sull'ultimo dei messaggi elencati poichè ci permette di costruire un meccanismo di sicurezza all'interno del nostro progetto.

La comunicazione tra il modulo di alto livello e di basso livello di cui abbiamo appena parlato può essere messa in piedi solo se entrambi i moduli conoscono questo protocollo di comunicazione. Per quanto riguarda il modulo di basso livello sappiamo che esiste un blocco all'interno del software APM Ardupilot che permette la comunicazione tramite protocollo MAVLINK. Per quanto riguarda il modulo di alto livello abbiamo invece la possibilità di inserire un nodo ROS specializzato proprio in questo denominato **MAVROS**.

Nei prossimi capitoli utilizzeremo ricorrentemente i concetti introdotti in questi primi due capitoli per descrivere i due progetti portati avanti nell' elaborato: uno mirato alla sicurezza dell'UAV, implementato sul modulo di basso livello e uno mirato al controllo dello stesso, implementato sul modulo di alto livello.

Capitolo 3

Implementazione del Safety Switch

Per la realizzazione del progetto è fondamentale prestare attenzione alla **sicurezza** dell'UAV a disposizione. Un UAV, nel caso in cui non rispettasse le leggi di controllo e non fossero state prese alcune misure di sicurezza, potrebbe diventare molto pericoloso e affliggere danni di qualsiasi entità. Dal momento che stiamo trattando oggetti in grado di volare, il mancato funzionamento del dispositivo e l'eventuale perdita di controllo potrebbero portare a situazioni, soprattutto in scenari civili, molto pericolose. Un'altro aspetto da considerare è quello economico: stiamo trattando tecnologie molto costose che, se non utilizzate con le adeguate precauzioni, potrebbero essere facilmente danneggiate, arrecando eventuali danni pecuniari.

L'ENAC

L'Ente Nazionale per l'Aviazione Civile (**ENAC**), ovvero l'unica Autorità di regolazione tecnica, certificazione, vigilanza e controllo nel settore dell'aviazione civile in Italia, è incaricata di regolare, certificare, vigilare e controllare anche l'uso degli UAV. Dall' Aprile 2014, tra le prime nazioni al mondo, è ufficialmente in vigore in Italia una normativa volta a regolamentare l'uso di questi mezzi sia per impieghi hobbistici sia per quelli professionali.

L'ENAC ha individuato un quadrante applicativo basato su due fattori: il tipo di utilizzo (professionale o hobbistico) e il peso dell'UAV (da 0 a 25 kg e oltre i 25kg). Sulla base delle possibili combinazioni, il regolamento determina dove sia possibile volare, se sia necessaria un'autorizzazione specifica o se basti un'autocertificazione con l'obbligo o meno di assicurazione. Sicuramente non tutto è ancora chiaro e definito. La procedura di approvazione, qualora necessaria, è piuttosto complessa.

Nel nostro caso, per ottenere un'autorizzazione da parte dell'ENAC ad utilizzare gli UAV outdoor, abbiamo bisogno di un sistema dotato di alti standard di sicurezza. In questo capitolo vedremo come abbiamo pensato di svilupparlo. Prima però affronteremo una panoramica del software APM Ardupilot per avere una base su cui fondare il nostro sistema.

3.1 APM Ardupilot

Prima di partire con la parte implementativa dobbiamo analizzare il software APM Ardupilot. Iniziamo con una descrizione generale. L'APM Ardupilot, come già accennato in precedenza, è un software open source sviluppato dalla community **DIY Drones** che comprende tre differenti implementazioni relative ad altrettanti tipi di Unmanned Vehicle:

- **ArduCopter**: implementazione per dispositivi aerei ad elica come ad esempio quadcopters, helicopters...;
- **ArduPlane**: implementazione per dispositivi aerei privi di elica;
- **ArduRover**: implementazione per rover di terra;

Vi é inoltre una quarta implementazione definita come **Antenna Tracker** che permette di controllare un'antenna direzionale tramite i segnali GPS inviati da un dispositivo elettronico, come ad esempio la Pixhawk o l'APM2, la quale però non verrà trattata all'interno di questo documento. Le varie implementazioni si basano sull'utilizzo delle stesse librerie di supporto. Ciò che differenzia l'una dalle altre è la parte di controllo. Infatti ognuna prevede delle leggi di controllo basilari sulle quali sono state costruite delle modalità di volo/esplorazione molto interessanti. Facendo riferimento all'ArduCopter, ovvero il software da noi utilizzato per il nostro quadcopter, abbiamo a disposizione quattordici modalità di volo che ci permettono ad esempio di mantenere la posizione, tornare alla base di partenza, mantenere la stessa altezza, eseguire delle traiettorie circolari, eseguire acrobazie...

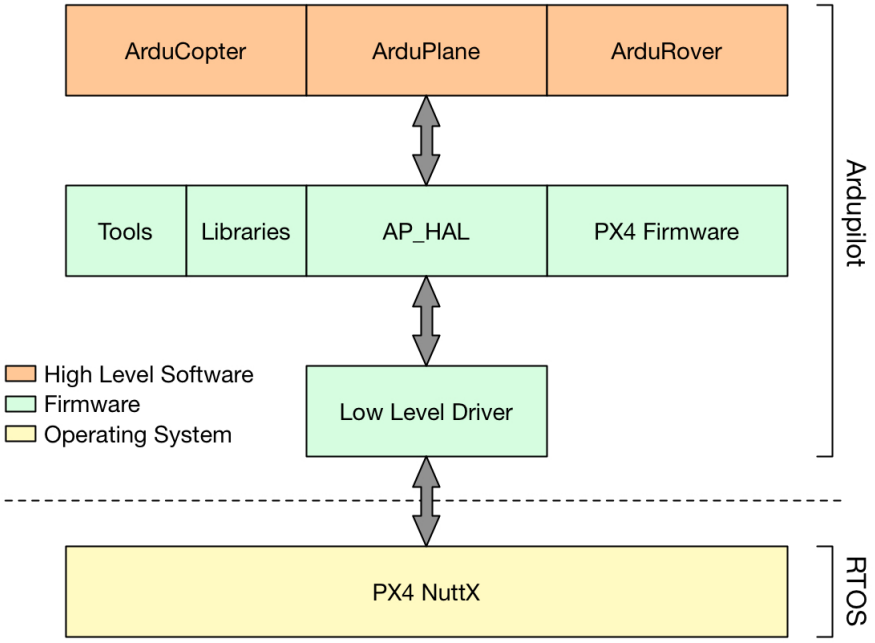

Figura 3.1: Struttura a livelli astratta del software Ardupilot e suo interfacciamento con il sistema operativo.

Grazie alla documentazione fornitaci dagli sviluppatori della community, è possibile intuire la struttura base del codice. Per far diventare il codice il più possibile manutenibile ed estensibile, gli sviluppatori della comunity DIY Drones l'hanno organizzato a livelli (Figura 3.1).

Ora entriamo un pò più nel dettaglio. Molte delle seguenti affermazioni derivano dallo studio a basso livello del codice dal momento che nella documentazione non vi sono API a supporto degli sviluppatori.

3.1.1 Low Level Driver

Il livello più basso della pila, in grado di interfacciarsi verso l'alto con il livello HAL e verso il basso con il sistema operativo NuttX, è quello dedicato ai **driver di basso livello** (**Low Level Driver**). L'interfacciamento con il mondo esterno viene quindi gestito da questo strato.

I driver di basso livello sono stati definiti nel package relativo allo strato HAL specifico della scheda utilizzata (del quale ci occuperemo nella prossima sottosezione) e sono:

- **AnalogIn**: driver relativi ai dispositivi di input analogici con i quali possiamo rilevare valori di corrente, tensione, timestamp di eventi particolari...;

- **GPIO**: driver relativi a dispositivi di I/O general purpose con i quali possiamo interagire tramite comandi di read, write, toggle...;

- **RCInput**: driver relativi a dispositivi radio in ingresso alla scheda con i quali possiamo interagire tramite comandi di read su determinati canali ed eseguire operazioni di modifica;

- **RCOutput**: driver relativi a dispositivi radio in uscita dalla scheda con i quali possiamo interagire tramite comandi di write su determinati canali ed eseguire operazioni di modifica;

- **Scheduler**: driver relativi allo scheduler con il quale possiamo interagire usando funzioni che ci permettono di ottenere, aggiungere o modificare dati riguardanti le code di scheduling;

- **Storage**: driver relativi ai supporti di memorizzazione sia volatili (EEPROM, FRAM) che non (Flash) con i quali possiamo interagire in lettura/scrittura;

- **UARTDriver**: driver relativi all'interfacciamento seriale utilizzando il protocollo UART tramite funzioni di lettura/scrittura.

Nella nostra struttura useremo principalmente i driver RCInput riguardanti l'accesso all'RC receiver. Riprenderemo in seguito l'analisi di questa classe quando parleremo del progetto del "safety switch".

3.1.2 AP_HAL

Nonostante questo blocco si trovi nello stesso livello di librerie, tools e PX4 Firmware abbiamo preferito dedicargli una sottosezione a parte poichè è fondamentale per la comunicazione fra i due mondi (quello di basso livello e quello di alto livello) del software APM Ardupilot. È stato denominato **AP_HAL** (che sta per **ArduPilot Hardware Abstraction Layer**) poichè costituisce l'unico modo che abbiamo per accedere ai driver di basso livello per poi manipolare i dispositivi fisici.

Questo livello è potentissimo poichè ci permette di astrarre completamente dai concetti di basso livello che compongono tutta la struttura, il che ci permette ad esempio di ricevere dati dai sensori o inviare dati agli attuatori

tramite la mediazione con i low level driver utilizzando dei semplici comandi che sono riutilizzabili all'interno di ognuna delle tre piattaforme.

Analizzando i sorgenti scopriamo che vi è un file denominato AP_HAL il quale è composto da una serie di associazioni a diverse tipologie di file tra le quali ve ne è una molto interessante ovvero l'interfaccia HAL. Quest'ultima definisce una serie di campi che permettono l'accesso alle interfacce di basso livello.

Inoltre vi sono diverse specializzazioni della seguente interfaccia con relativa implementazione specifica per ogni scheda che supporta il software APM Ardupilot situati nel package che assume il seguente aspetto: `libraries/AP_HAL_xxx/`, dove "xxx" indica il nome della scheda. Nel nostro caso faremo riferimento al package `libraries/AP_HAL_PX4/` dato che stiamo utilizzando una scheda Pixhawk. Il tutto è riassunto in Figura 3.2.

Il concetto principale è che in una classe di alto livello viene definito un riferimento ad un oggetto di tipo AP_HAL denominato per l'appunto ***hal*** che viene instanziato una sola volta alla creazione delle stesse. L'uso di questo tipo di meccanismo si avvicina molto al pattern creazionale denominato **Singleton** e ci permette di poter risalire in seguito al riferimento della classe AP_HAL senza dover instanziare ogni volta un nuovo oggetto. Quindi, per quanto detto, possiamo schematizzare la struttura di questo layer (limitandoci a descrivere le interfacce e classi principali) atrraverso la Figura 3.2. I campi pubblici che troviamo definiti all'interno dell'interfaccia principale HAL, che non sono altro che riferimenti alle interfacce di basso livello, sono poi inizializzati nella classe di specializzazione in modo coerente con gli aspetti tecnici della scheda in questione. Una breve definizione di ciò che i campi stanno ad indicare è la seguente:

- **UARTDriver* uart0**: riferimento all'oggetto Serial0 tramite la libreria ArduPilot FastSerial nella quale viene ridefinito l'Arduino Core Serial Object responsabile della trasmissione seriale tramite protocollo UART definito nella classe `ardupilot/libraries/AP_HAL/UARTDriver.h`;
- **UARTDriver* uart1**: riferimento all'oggetto Serial1 (vedi uart0);
- **UARTDriver* uart2**: riferimento all'oggetto Serial2 (vedi uart0);
- **UARTDriver* uart3**: riferimento all'oggetto Serial3 (vedi uart0);
- **I2CDriver* i2c**: riferimento all'oggetto che permette l'utilizzo del protocollo di comunicazione seriale I2C definito nella classe `ardupilot/libraries/AP_HAL/I2CDriver.h`;

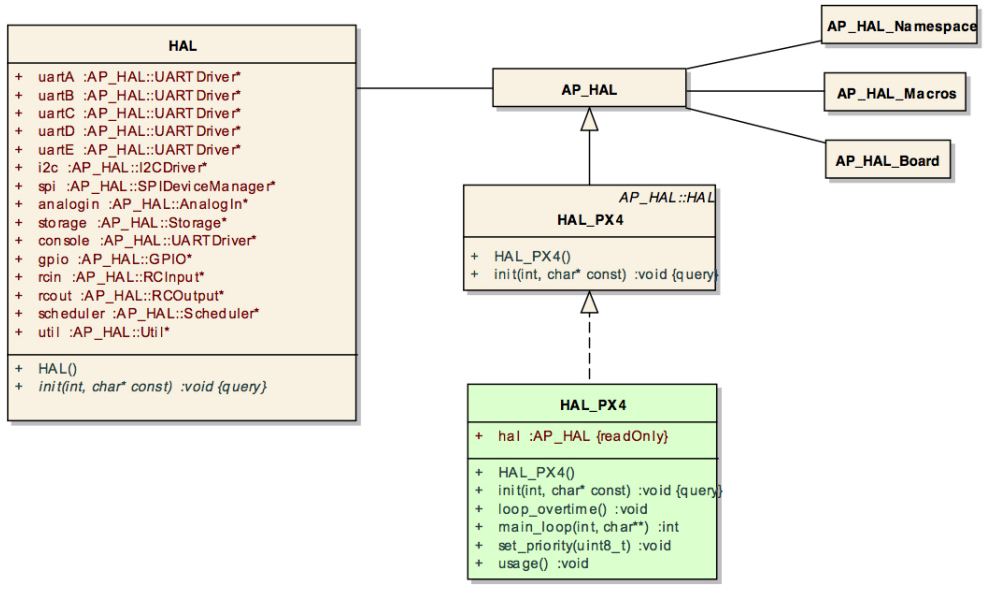

Figura 3.2: Struttura dell'HAL basata sugli aspetti di nostro interesse.

- **SPIDriver* spi**: riferimento all'oggetto che permette l'utilizzo del protocollo di comunicazione seriale SPI definito nella classe `ardupilot/libraries/AP_HAL/SPIDriver.h`;
- **AnalogIn* analogin**: riferimento all'oggetto che gestisce i segnali analogici in ingresso alla scheda definito nella classe `ardupilot/libraries/AP_HAL/AnalogIn.h`;
- **Storage* storage**: riferimento all'oggetto che gestisce la memorizzazione dei dati all'interno della scheda definito nella classe `ardupilot/libraries/AP_HAL/Storage.h`;
- **Dataflash* dataflash**: riferimento all'oggetto che gestisce la memorizzazione dei dati in memoria Flash all'interno della scheda definito nel package `ardupilot/libraries/DataFlash/`;
- **ConsoleDriver* console**: riferimento all'oggetto/utility che permette il debug del codice filtrando segnali di warning ed errore;
- **GPIO* gpio**: riferimento all'oggetto che permette le funzionalità di default presenti nel core Arduino ovvero pinMode, digitalRead, and digitalWrite definito nella classe `ardupilot/libraries/AP_HAL/GPIO.h`;

- **RCInput* rcin**: riferimento all'oggetto che gestisce i segnali PPM in ingresso dall'RC channel definito nella classe `ardupilot/libraries/AP_HAL/RCInput.h`;

- **RCOutput* rcout**: riferimento all'oggetto che gestisce i segnali PPM in uscita dall'RC channel definito nella classe `ardupilot/libraries/AP_HAL/RCOutput.h`;

- **Scheduler* scheduler**: riferimento all'oggetto che permette di utilizzare sia le funzionalità temporali del core Arduino (millis, delay...), sia quelle introdotte nella classe `library/AP_PeriodicProcess` definito nella classe `ardupilot/libraries/AP_HAL/Scheduler.h`.

In sostanza tutti i campi esaminati sono dei riferimenti ai driver definiti all'interno della libreria Arduino e/o ridefiniti a loro volta nella libreria Ardupilot.

Analizzando il comportamento della classe *AP_HAL_xxx* relativa ad una specifica piattaforma ci accorgiamo che vengono creati un certo numero di thread a supporto delle fondamentali funzionalità che l'oggetto HAL dovrà offrire. Ovviamente questo meccanismo è verificato solo in alcune schede che supportano il multithreading come ad esempio la Pixhawk. Proprio in quest'ultima vengono creati i seguenti thread relativi all'oggetto HAL:

- l'UART e l'USB thread, incaricati di leggere e scrivere tramite l'utilizzo dei protocolli UART e USB;

- il timer thread il quale può essere usato per eseguire funzioni in maniera sequenziale ad una frequenza di 1kHz. Ad esempio all'interno del file di libreria *AP_Baro_MS5611.cpp* troviamo il seguente codice responsabile di registrare un timer callback ad una funzione da eseguire:

```
1   hal.scheduler->register_timer_process(
        AP_HAL_MEMBERPROC(&AP_Baro_MS5611::_update));
```

Listing 3.1: Timer Callback.

- l'IO thread incaricato di leggere e scrivere su microSD, EEPROM e FRAM.

All'interno del package `ardupilot/libraries/AP_HAL_xxx/` vi è una classe denominata *Scheduler.cpp* nella quale vi sono definiti tutti i thread generati dallo specifico oggetto HAL con la relativa priorità temporale.

3.1.3 Librerie, Tools e PX4 Firmware

Le librerie sono un insieme di funzioni e strutture utilizzate dalla parte di alto livello del software APM per eseguire operazioni spesso complicate e di uso frequente. Esse fanno parte del layer intermedio, posizionato tra i driver di basso livello e le Vehicle Directories.

Il PX4 Firmware è comunque una raccolta di funzioni relative esclusivamente alle piattaforme PX4, come ad esempio la Pixhawk. Spesso capita che le funzioni implementate nel package `Ardupilot/libraries/` accedino al PX4 Firmware per avere dei dettagli maggiori a livello implementativo riguardo la componentistica fisica delle schede PX4.

I tools hanno le stesse caratteristiche delle librerie e del PX4 Firmware. La loro differenza principale è che sono stati realizzati per fornire dei meccanismi per il testing e la simulazione del software.

Nonostante siano tre strumenti molto importanti, cercheremo di analizzare esclusivamente la struttura delle librerie poichè di maggior rilievo all'interno del nostro progetto rispetto ai restanti due.

Le librerie sono formate da **sketch**, file con estensione ".pde" contenenti un main program. Generalmente vengono strutturati nel seguente modo:

- vengono definiti diversi *#include* i quali servono a generare automaticamente le dipendenze e le regole di compilazione. Al momento della compilazione i file ".pde" vengono convertiti in file C++ e tramite le direttive *#include* viene creato un link con altri file;
- viene referenziata la variabile *hal*, riferita alla singola instanza dell'oggetto di tipo HAL discusso nella precedente sottosezione;
- viene definito ed implementato un metodo di inizializzazione. Questa funzione viene chiamata al momento dell'avvio della scheda e costruisce l'infrastruttura necessaria;
- vengono definiti ed implementati dei metodi per le determinate funzionalità offerte dalla classe specifica. Solitamente si occupano di aggiornare valori e di convertirli per poi passarli al blocco HAL.

3.1.4 Vehicle Directories

Il livello più alto della pila all'interno del software APM Ardupilot comprende le tre specializzazioni per le relative piattaforme: ArduCopter, ArduPlane e ArduRover. La trattazione è focalizzata sul software ArduCopter.

Innanzitutto partiamo suddividendo le modalità di volo implementate in due gruppi:

- **modalità di volo automatiche o autopilotate**: sono modalità di volo che prendono in ingresso una posizione specifica nello spazio e automaticamente la raggiungono senza ulteriori intermediazioni con il controllore che ha definito il riferimento. Come esempio troviamo la **Return To Launch (RTL)** che permette all'UAV di ritornare nella posizione di partenza, ovvero l'origine del sistema di riferimento considerato da quest'ultimo, in maniera del tutto automatica;

- **modalità di volo manuali**: sono modalità di volo che prendono costantemente degli input da un controllore (come ad esempio l'RC transmitter) espressi in termini di posizione o velocità. Grazie all'algoritmo di controllo questi riferimenti vengono trasformati in comandi di basso livello rendendo il pilotaggio il meno complicato possibile. Se proviamo a bypassare la parte di controllo ci rendiamo conto che il mezzo diventa difficilmente controllabile da un uomo. Come esempio troviamo la **Stabilize** che permette all'UAV di mantenere tutti i riferimenti fissati dal controllore in real time.

Per avere un'idea più chiara di come è organizzato il software APM ArduCopter ad alto livello, analizziamo una modalità di volo manuale prendendo come esempio la modalità "stabilize".

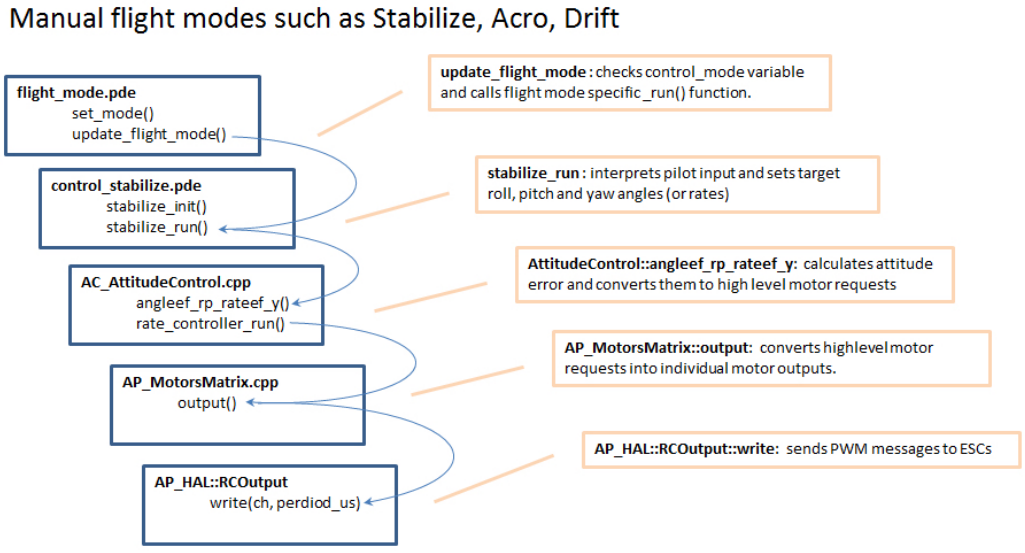

Figura 3.3: Interazione all'interno dell APM ArduCopter con riferimento ad una modalità di volo di tipo manuale.

In ogni singolo loop rappresentato nella Figura 3.3, che nel caso della scheda Pixhawk ha una frequenza di 400Hz, avvengono le seguenti operazioni:

1. viene chiamata la funzione *update_flight_mode()* definita nella classe *flight-mode.pde*. Questa funzione verifica l'ultima modalità di volo selezionata all'interno della variabile *control_mode* ed invoca il metodo $\langle flightmode \rangle\_run()$ appropriato utilizzando la classe dedicata alla $\langle flightmode \rangle$ in questione. Ad esempio se ci stiamo riferendo alla modalità *stabilize* allora verrà richiamata la funzione *stabilize_run()* all'interno della classe *control_stabilize.pde*;

2. la funzione $\langle flightmode \rangle\_run()$ è responsabile di convertire l'input del controllore in diversi parametri specifici per la legge di controllo implementata per quella modalità di volo. Inoltre quest'ultima è incaricata di passare i parametri appena convertiti all'**Attitude Controller** e/o al **Position Controller** collocati nel package `libraries/AC_AttitudeControl/`;

3. sia l'Attitude Controller che il Position Controller richiamano il loro metodo associato per passare il valore alla classe **AP_Motors** (l'Attitude Controller mediante il metodo *rate_controller_run()* mentre il Position Controller mediante il metodo *update_xy_controller()*);

4. la classe AP_Motors specifica per il tipo di motori utilizzati converte i riferimenti espressi in roll, pitch, yaw e throttle provenienti dall'AttitudeController e dal PositionController in segnali PWM da inviare ai motori. A fare ciò se ne occupa il metodo *output_armed()*. Quest'ultimo, grazie al famoso riferimento **hal**, invia il segnale ai motori tramite il comando:

```
1    hal.rcout->write();
```

Listing 3.2: Interazione con i Motori.

3.2 Safety Switch

La possibilità di avere un software molto completo, in continua evoluzione e soprattutto open source ci ha permesso di sperimentare dei meccanismi di sicurezza appropriati per procedere con le fasi di test outdoor. Il nostro fine, in termini di sicurezza, è quello di mantenere il controllo dell'UAV in qualsiasi istante e in qualsiasi circostanza. L'obiettivo che ci siamo posti,

apparentemente ovvio, non è stato implementato all'interno del firmware APM, il che costituisce una falla potenzialmente molto grande del sistema.

3.2.1 Requisiti

Sappiamo dai capitoli precedenti che il modulo di basso livello, responsabile del controllo degli attuatori, può essere controllato da tre entità:

1. il **radiocomando**;
2. il **modulo di controllo di alto livello**;
3. la **Ground Control Station** (anche se utilizzata nel nostro sistema come interfaccia dati);

Sia il modulo di controllo di alto livello che la GCS hanno priorità maggiore rispetto al radiocomando nell'APM senza la nostra modifica e, tramite l'uso del protocollo MAVLINK, inviano dei messaggi di RC_OVERRIDE per sovrascrivere le informazioni provenienti dall'RC Transmitter nel momento in cui hanno qualcosa da comunicare alla Pixhawk. Ciò significa che, se stiamo radiocomandando l'UAV, esso può smettere di eseguire i nostri comandi per iniziare ad eseguirli dalle altre due entità.

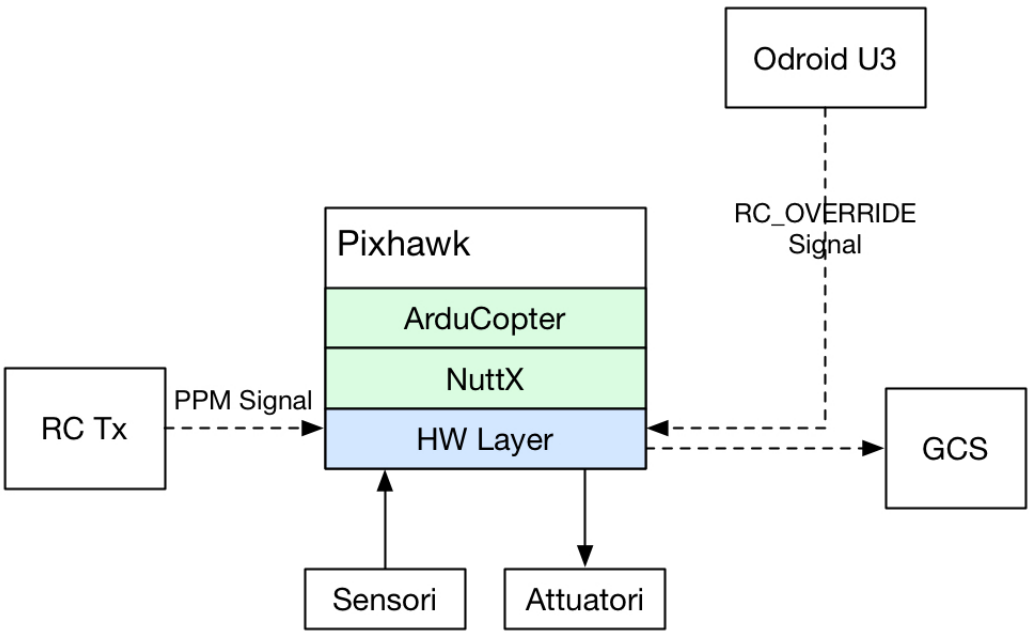

Figura 3.4: Struttura a livelli astratta del modulo di controllo di basso livello e interazione con gli altri tre moduli.

Perciò l'idea è quella di **utilizzare uno switch del radiocomando per abilitare/disabilitare questa modalità di esecuzione dei comandi basati sulla priorità**.

I requisiti sono perciò i seguenti:

- il trasmettitore RC avrà a disposizione il **canale 7 (CH7)** relativo ad uno **switch fisico** che può assumere due posizioni:
  - **0**: in questa posizione dello switch il sistema si deve comportare normalmente così come è stato descritto fino ad ora;
  - **1**: in questa posizione dello switch il sistema deve ricevere i comandi esclusivamente dall'RC transmitter cestinando a priori i dati provenienti dagli altri due moduli.
- il sistema dovrà essere progettato tenendo conto di tutta l'infrastruttura descritta finora. In particolare dovrà:
  - utilizzare il protocollo di comunicazione **MAVLINK**;
  - rendere la modifica estendibile a qualsiasi tipo di velivolo e quindi compatibile con **ArduCopter, ArduPlane, ArduRover**.
- il sistema dovrà essere efficiente e non dovà compromettere il software già sviluppato.

3.2.2 Analisi dei Requisiti

Dai requisiti emerge che dobbiamo trovare un meccanismo efficiente per tenere sotto controllo l'UAV tramite un semplice switch posizionato sull'RC transmitter. Ragionando nel dominio dei tempi, le **operazioni** che devono verificarsi in maniera sequenziale sono:

- lo switch viene portato dalla posizione 0 alla posizione 1 (o viceversa);
- l'RC transmitter invia un segnale all'RC receiver collegato con il modulo di controllo di basso livello (Pixhawk) all'interno dell'UAV;
- il loop di controllo all'interno del software Ardupilot legge i dati provenienti dall'RC transmitter e verifica se i dati relativi al CH7 sono diversi rispetto a quelli registrati in precedenza (ovvero se lo switch è stato spostato);

- in caso in cui lo switch sia rimasto nella stessa posizione non dobbiamo apportare alcune modifiche. In caso contrario dovremo verificare quale delle seguenti condizioni si è verificata e agire di conseguenza. Infine si avranno i dati filtrati;
- i dati filtrati vengono passati alle classi del livello sottostante incaricate di convertire i comandi in frequenza di rotazione per ognuno dei motori.

Gli **scenari** che si possono verificare derivano dal passaggio da uno stato all'altro dello switch sull'RC transmitter e sono due:

- **passaggio da 0 a 1**: devono essere eseguiti esclusivamente i comandi provenienti dall'RC transmitter;
- **passaggio da 1 a 0**: devono essere eseguiti tutti i tipi di comandi con priorità maggiore per i comandi provenienti dalla GCS e dal modulo di controllo di alto livello.

3.2.3 Analisi del Problema

A **livello logico**, il sistema che soddisfa i primi due requisiti sopra descritti potrebbe essere strutturato nel seguente modo:

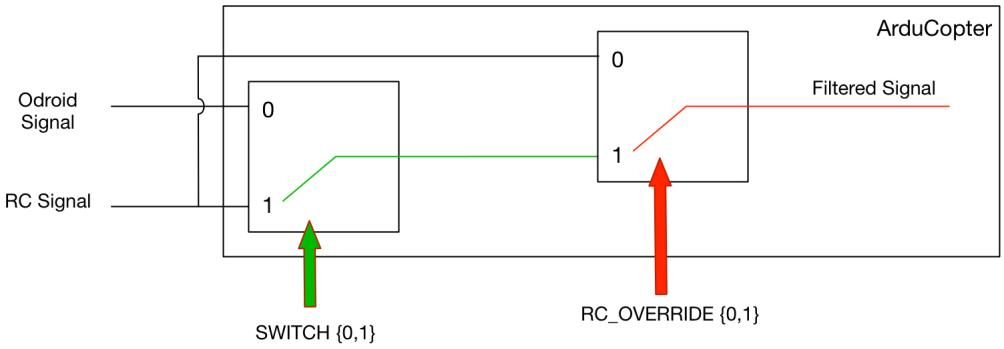

Figura 3.5: Schema logico della struttura del sistema del Safety Switch.

Per poter soddisfare il terzo requisito bisogna analizzare più attentamente l'architettura sulla quale stiamo operando. In termini matematici possiamo dire che l'efficienza di un software è inversamente proporzionale alla distanza in termini di astrazione dallo strato fisico.

Infatti ponendoci ad un alto livello di astrazione, sicuramente guadagneremo in termini di gestione del sistema, di visione d'insieme, di estendibilità e manutenibilità del codice e, cosa importantissima, possiamo tralasciare gli

aspetti tecnici relativi all'infrastruttura sulla quale il nostro software è stato mandato in esecuzione. Ponendoci invece ad un basso livello di astrazione ci dovremo preoccupare degli aspetti tecnici ma abbiamo il vantaggio di guadagnare in termini di efficienza.

Questo discorso ci porta a supporre che l'**abstraction gap** da tenere in considerazione sarà abbastanza ristretto. Inoltre potremo ridurre ancora di più questo gap per aumentare l'efficienza lavorando a livello hardware sull'RC receiver in modo da verificare i dati nel momento in cui essi vengono ricevuti. Così facendo si cadrebbe però in una condizione che non soddisferebbe né parte del secondo requisito, né parte del terzo. Infatti la modifica potrebbe non essere compatibile con tutti i tipi di veivoli e potrebbe compromettere parte del software già sviluppato.

Facendo un bilancio e analizzando i **rischi** che possiamo correre con queste due impostazione del sistema, risulta più sicuro un software leggermente meno efficiente (con differenza di tempi di reazione comunque molto piccola) rispetto ad un software potenzialmente inaffidabile poiché sono state modificate inconsciamente o erroneamente parti importanti del sistema già esistente dal momento che non vi è una precisa documentazione del software a livello di API. Fortunatamente vengono date alcune informazioni importanti per comprendere a livello logico la struttura principale del codice di tutto il software Ardupilot che ci permettono di intraprendere la prima delle due strade.

3.2.4 Progetto

Per la realizzazione del progetto abbiamo bisogno di analizzare per bene i due blocchi che abbiamo introdotto in Figura 3.5 partendo da:

1. come viene implementata la struttura di ricezione dati dell'RC receiver;
2. come viene implementata la gestione dei messaggi di RC_OVERRIDE attraverso il protocollo MAVLINK.

3.2.4.1 Struttura

Analizziamo la struttura in questione attenendoci ai due punti sopra elencati. Dal momento che le interfacce in questione sono composte dalla definizione di moltissimi campi e metodi, ci limiteremo ad elencare solo i più rilevanti nei seguenti schemi UML (Unified Modeling Language).

Iniziamo con la struttura dei canali RC. Nella parte di alto livello (facendo riferimento alla Figura: 3.1) vi è l'interfaccia *Parameters.h* la quale definisce i primi quattro canali di tipo *RC_Channel* e gli ultimi quattro di

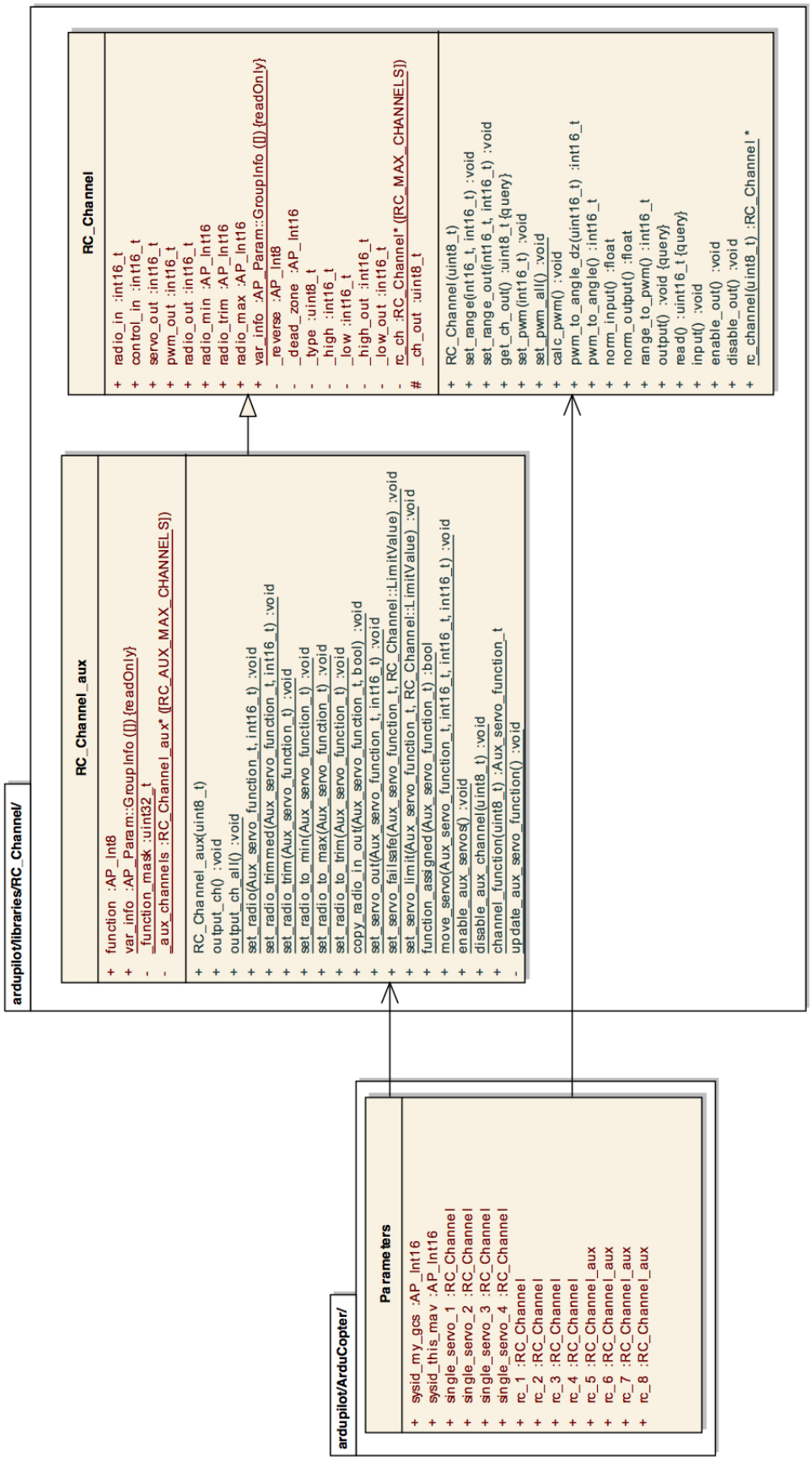

Figura 3.6: Struttura dei canali RC.

tipo *RC_Channel_aux*. Questi due tipi di dato possono essere analizzati rispettivamente attraverso le interfacce *RC_Channel.h* e *RC_Channel_aux.h*, collocate nel livello intermedio all'interno della sezione di libreria, dove la seconda è una specializzazione della prima. In esse sono definiti i campi e i metodi utilizzati per lavorare a basso livello con i canali RC. In particolare il campo *radio_in*.

L'interfaccia *Parameters.h* viene implementata dalla relativa *Parameters.pde* la quale attraverso il metodo *load_parameters()* carica tutti i parametri definiti nell'interfaccia accedendo alle funzioni di libreria. Lo scopo di questo meccanismo è che la classe principale *ArduCopter.pde* (riferendoci alla sola implementazione per veivoli multirotore) avrà in questo modo una base già avviata di tutti i componenti con la quale poter interagire, dove l'interfaccia *Parameters.h* funge da link dal momento che contiene il riferimento a tutti gli oggetti di livello intermedio.

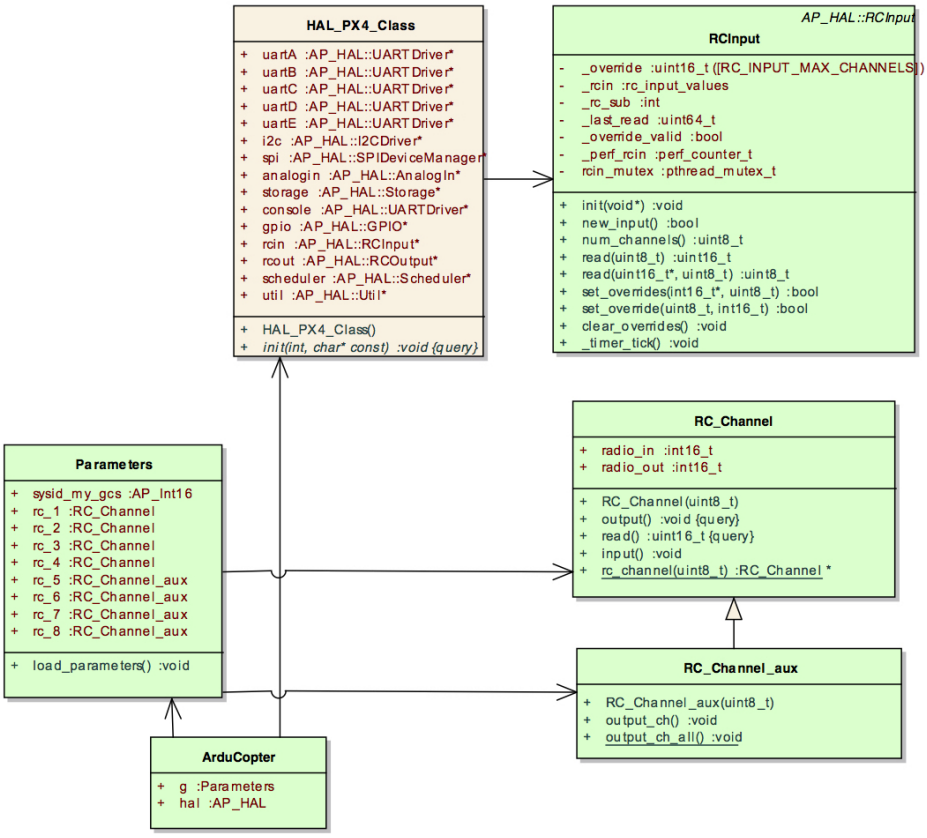

Figura 3.7: Struttura della nostra parte di interesse per l'implementazione del safety switch.

Inoltre la classe *ArduCopter.pde* ha un riferimento al famoso oggetto di tipo HAL specifico per l'architettura utilizzata. Nello schema in Figura 3.7 vediamo come sono interconnesse le parti evidenziando anche il legame che vi è fra l'oggetto HAL e l'oggetto RCInput, ultimo blocco prima dell'accesso al dispositivo fisico. Ancora una volta sono stati tralasciati molti dati interni alle classi che troviamo raffigurate poichè molto numerosi e non di nostro interesse.

Possiamo osservare quindi che vi sono due classi molto somiglianti fra loro: *RCInput* e *RC_Channel* con relativa classe specializzata *RC_Channel_aux*. La prima è un'istanza di basso livello dove sono salvati i dati veri e propri ricevuti dai dispositivi fisici, la seconda invece è un'istanza di livello più alto nella quale vi è una sorta di copia della prima nella quale vengono filtrati solo i dati in base all'utilizzo che se ne deve fare. Per fare in modo quindi di rendere l'implementazione del "safety switch" il più possibile "sicura" andremo a modificare i dati nella prima delle due classi. Inoltre scopriamo che i dati in ingresso al loop di controllo del software APM Ardupilot sono prelevati proprio da qui.

3.2.4.2 Interazione

L'interazione è molto importante in questo progetto e allo stesso tempo molto delicata poichè, non avendo piena padronanza della vasta architettura del software APM Ardupilot, una singola funzione fuori posto potrebbe compromettere il corretto funzionamento dell'UAV a prescindere dal concetto di "safety switch". Studiando a basso livello il codice ci accorgiamo che la responsabilità di controllare periodicamente l'arrivo di messaggi MAVLINK è affidata alla classe *GCS_Mavlink.pde*.

Nel caso in cui arriva un messaggio MAVLINK in ingresso, viene invocato il metodo *handleMessage(mavlink_message_t* msg)* il quale è incaricato di controllare di che tipo è. Se esso risulta essere di tipo *RC_CHANNELS_OVERRIDE* allora si deve verificare se è stato attivato il meccanismo di *safe mode* tramite il "safety switch" oppure no.

Per far ciò, abbiamo bisogno di accedere ai dati relativi all'RC transmitter che sono stati ricevuti dall'RC receiver. Come abbiamo visto dalla Figura 3.7 tutti i dati di ricezione inviati dall'RC transmitter sono periodicamente salvati nella classe **RCInput.pde** e più precisamente nel campo *radio_in*. Inoltre sappiamo che nella classe *Parameters.h* sono definiti gli otto canali dell'RC transmitter proprio di tipo **RCInput**.

Il loop si chiude grazie al campo **g** presente nella classe **ArduCopter.pde** che si riferisce appunto ad un istanza di *Parameters.pde* accedibile dalla classe *GCS_Mavlink.pde*.

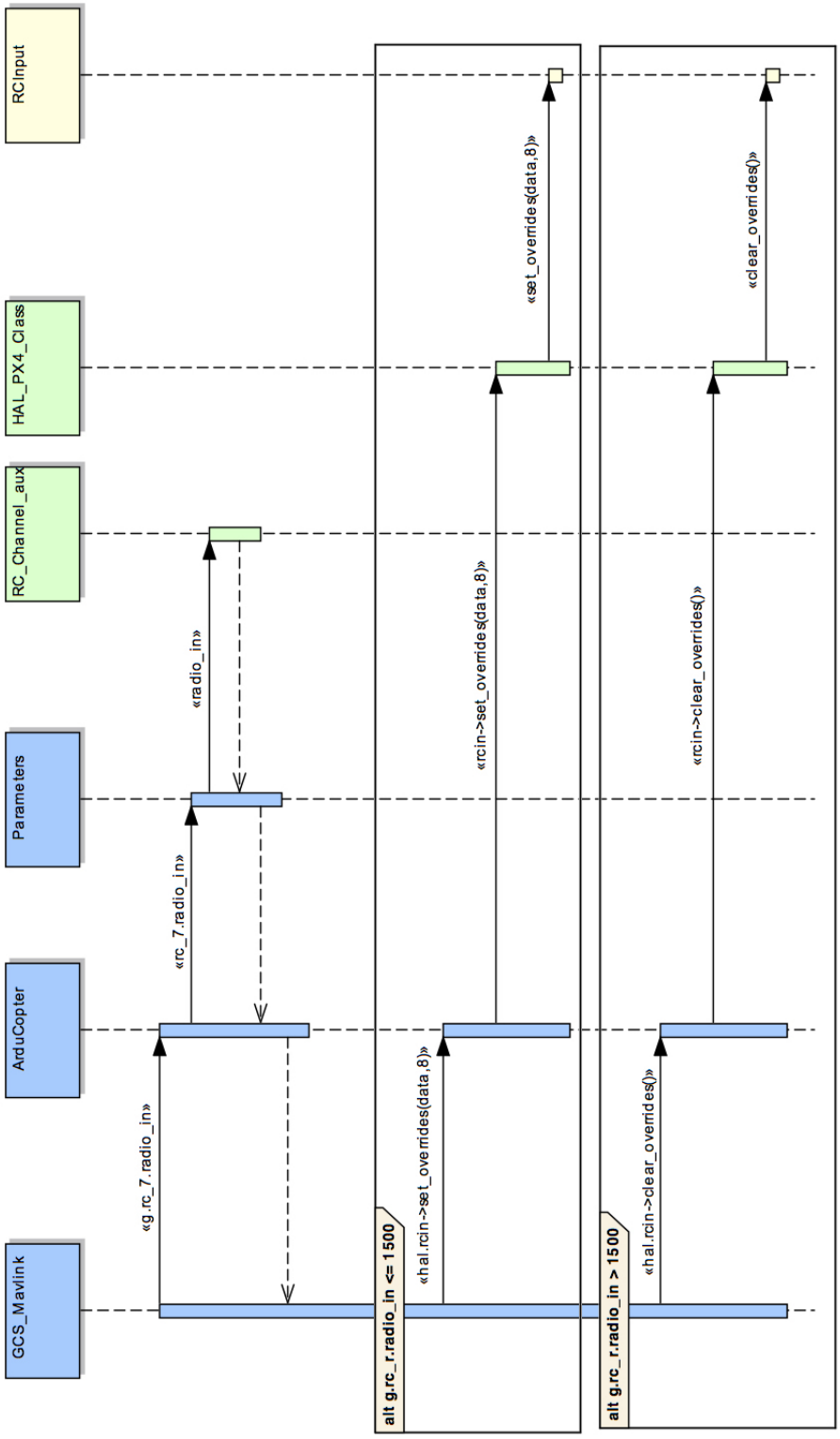

Figura 3.8: Interazione della nostra parte di interesse per l'implementazione del safety switch.

Una volta ottenuti i dati possiamo passare alla valutazione dei casi. Analizziamoli separatamente:

1. **caso *"safe mode"* disabilitato**: bisogna sovrascrivere i dati provenienti dall'RC transmitter. Quindi utilizzando il riferimento *hal* alla classe *HAL _PX4 _Class* accederemo alla struttura *rcin* ed eseguiremo il metodo *set _overrides(data,n)*, dove *data* indica un vettore di *n* elementi (nel nostro caso 8);

2. **caso *"safe mode"* abilitato**: bisogna lasciare passare esclusivamente i dati provenienti dall'RC transmitter. In questo caso riaccederemo alla stessa struttura ma questa volta eseguiremo il metodo *clear_overrides()* il quale permette di eliminare gli override provenienti da qualsiasi fonte.

3.2.4.3 Comportamento

Il sistema si comporta semplicemente muovendosi fra due stati. Il passaggio da uno stato all'altro è determinato dalle combinazioni tra l'evento di *RC_OVERRIDE* e la posizione dello switch fisico. Senza complicare troppo il discorso ecco la macchina a stati del sistema:

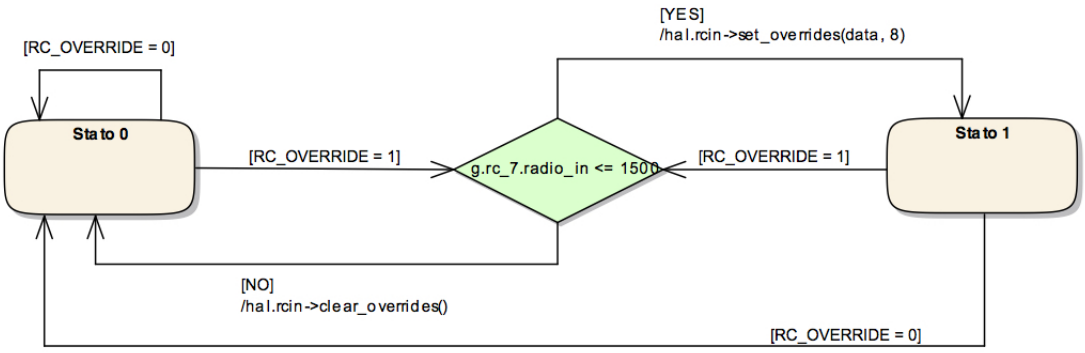

Figura 3.9: Comportamento della nostra parte di interesse per l'implementazione del safety switch.

3.2.5 Implementazione

Siamo giunti quindi alla parte implementativa del sistema del "safety switch". Iniziamo mostrando la parte di codice fondamentale per far si che

il sistema soddisfi i requisiti iniziali. La seguente modifica è stata apportata al file `ardupiot/ArduCopter/GCS_Mavlink.pde`:

```
1  case MAVLINK_MSG_ID_RC_CHANNELS_OVERRIDE: // MAV_ID: 70
2  {
3      if (g.rc_7.radio_in <= 1500) { // posizione 0
4          mavlink_rc_channels_override_t packet;
5          int16_t v[8];
6          mavlink_msg_rc_channels_override_decode(msg, &packet); //
               decodifico il messaggio mavlink e lo salvo nella struttura
               packet
7
8          v[0] = packet.chan1_raw;
9          v[1] = packet.chan2_raw;
10         v[2] = packet.chan3_raw;
11         v[3] = packet.chan4_raw;
12         v[4] = packet.chan5_raw;
13         v[5] = packet.chan6_raw;
14         v[6] = packet.chan7_raw;
15         v[7] = packet.chan8_raw;
16         hal.rcin->set_overrides(v, 8); // salvo i valori di ogni canale
               nel vettore v e lo sovrascrivo ai dati esistenti provenienti
               dall'RC transmitter
17
18          failsafe.rc_override_active = true; // abilito il failsafe
19          failsafe.last_heartbeat_ms = millis(); // salvo l'istante nel
               quale ho apportato la modifica
20         break;
21     } else {     // posizione 1
22         hal.rcin->clear_overrides(); // non sovrascrivo i valori e
               pulisco eventuali sovrascritture
23          failsafe.rc_override_active = false; // disabilito il failsafe
24          failsafe.last_heartbeat_ms = millis(); // salvo l'istante nel
               quale ho apportato la modifica
25         break;
26     }
27 }
```

Listing 3.3: Safety Switch Code.

Possiamo interpretare questo blocco di codice ricordandoci tutto ciò che abbiamo appreso in questo capitolo. Tuttavia ci sono dei dettagli da specificare. Innanzi tutto chiariamo il fatto che i dati specifici della posizione giunti all'interno della scheda tramite il ricevitore RC saranno letti da tutto il sistema facendo riferimento al campo **hal.rcin**.

Nel caso in cui arrivi un RC_OVERRIDE con switch posto in posizione *0* bisogna sovrascrivere i dati ricevuti. Per fare ciò utilizziamo un vettore di otto interi a 16bit denominato **v**. In esso andremo a salvare il contenuto del messaggio RC_OVERRIDE organizzato per canali. Precedentemente tramite la funzione **mavlink_msg_rc_channels_override_decode(msg, &packet)** abbiamo decodificato il messaggio MAVLINK suddividendolo negli 8 canali e l'abbiamo salvato nella struttura **packet**. Infine la sovrascrittura viene effettuata tramite il comando **set_overrides(v,8)**. Inoltre viene abilitata la funzione di *failsafe* per l'operazione di *rc_override* con relativa registrazione dell'ultima modifica utile nel caso in cui non venisse ricevuto nessun altro messaggio di RC_OVERRIDE per un certo periodo di tempo in modo da ridare comunque il controllo all'RC transmitter.

Nel caso in cui arrivi un RC_OVERRIDE con switch posto in posizione *1* non bisogna fare niente poicè i dati devono essere prelevati così come sono stati trasmessi dall'RC transmitter. Il comando **clear_overrides()** serve ad accertarsi che non sia stato sovrascritto alcun dato. In questo caso la funzione di *failsafe* per l'operazione di *rc_override* viene disabilitata poichè lo switch in posizione *1* ci da un grado di sicurezza più alto.

In caso si intendesse implementare questo meccanismo per un Rover ad esempio, può essere fatto molto agilmente modificando il file `ardupilot/APMrover2/GCS_Mavlink.pde` con lo stesso codice usato per il copter.

3.2.6 Deployment

Il passo finale è quello della **compilazione** del codice inserito all'interno del software APM Ardupilot e dell'**upload** sulla scheda Pixhawk.

Innanzi tutto bisogna creare una directory nella quale vengono collocati:

- **Software Ardupilot** (con le relative modiche apportate);
- **PX4 Firmware**;
- **PX4 NuttX**.

Tramite i seguenti comandi da terminale entriamo nella directory dell'ArduCopter e generiamo il **configuration file** responsabile della compilazione:

```
1 cd ardupilot/ArduCopter
2 make configure
```

Listing 3.4: Configuration File.

Modifichiamo il configuration file **config.mk** nella directory `ardupilot/` in modo da dargli la possibilità di accedere al PX4 Firmware e al PX4 NuttX in fase di compilazione:

```
1 PX4_ROOT=../PX4Firmware // puntatore alla directory del PX4
      Firmware
2 NUTTX_SRC=../PX4NuttX/nuttx // puntatore alla directory principale
      del PX4 NuttX
```

Listing 3.5: Modifica al Configuration File.

Eseguiamo i seguenti comandi per pulire e compilare l'intero sistema (Ardupilot + PX4 Firmware + PX4 NuttX) specificatamente per la Pixhawk:

```
1 make px4-clean // pulizia
2 make px4-v2 // compilazione
```

Listing 3.6: Compilazione.

Il processo di compilazione può durare abbastanza tempo (30/40 minuti).

Una volta terminata la compilazione, se andata a buon fine, può essere caricata sulla Pixhawk tramite un cavo USB tramite il seguente comando:

```
1 make px4-v2-upload // caricamento sulla Pixhawk
```

Listing 3.7: Upload.

Se tutti i precedenti passaggi sono stati eseguiti correttamente, avremo a disposizione un UAV con tutto il necessario per supportare il nuovo sistema di safety switch.

Capitolo 4

Legge di Controllo ad Alto Livello

Siamo giunti quindi alla parte in cui affrontiamo una tematica molto interessante e soprattutto importante per il corretto funzionamento del sistema.

Riprendiamo brevemente ciò che abbiamo detto in precedenza. Nel nostro sistema possiamo comandare l'UAV in tre modi. Il primo riguarda l'uso dell'RC transmitter. In questo caso l'UAV riceverà un segnale PPM che verrà interpretato come segnale di riferimento per l'algoritmo di controllo specifico della modalità di volo selezionata. Il secondo riguarda invece l'uso del software APM Planner in esecuzione sulla GCS. Questo metodo è il meno usato. Solitamente si delega alla GCS la funzionalità di pannello di controllo nel quale avremo tutti i dati tecnici relativi all'UAV. A livello diagnostico è fondamentale poichè ci permette istante per istante di monitorare le condizioni del velivolo. Il terzo modo è mediante una base remota, che in alcuni casi potrebbe coincidere con la GCS. Basandoci sulle affermazioni appena fatte potremmo ridurre le funzionalità di comando al solo RC transmitter e al calcolatore remoto.

Dal momento che stiamo progettando UAV in grado di supportare situazioni critiche, avremo però bisogno di sistemi resistenti, creati appositamente per soddisfare determinati requisiti, più propriamente detti "obiettivi di controllo". Ovviamente questo implica la progettazione di un sistema di controllo che vada a completare quello implementato nell'APM Ardupilot.

L'architettura di cui abbiamo parlato finora è perfetta per supportare leggi di controllo codificate e messe in esecuzione sul modulo di alto livello. Da ora in poi, nel resto del capitolo, faremo riferimento ad uno scenario nel quale il modulo di alto livello riceve dei comandi in termini di posizione dal calcolatore remoto. Lo stesso modulo ha come obiettivo quello di rendere

automatico lo spostamento dell'UAV attraverso la mediazione con il modulo di basso livello.

4.1 Architettura di Controllo

L'architettura di controllo utilizzata nell'obiettivo O1 del progetto SHERPA è la seguente:

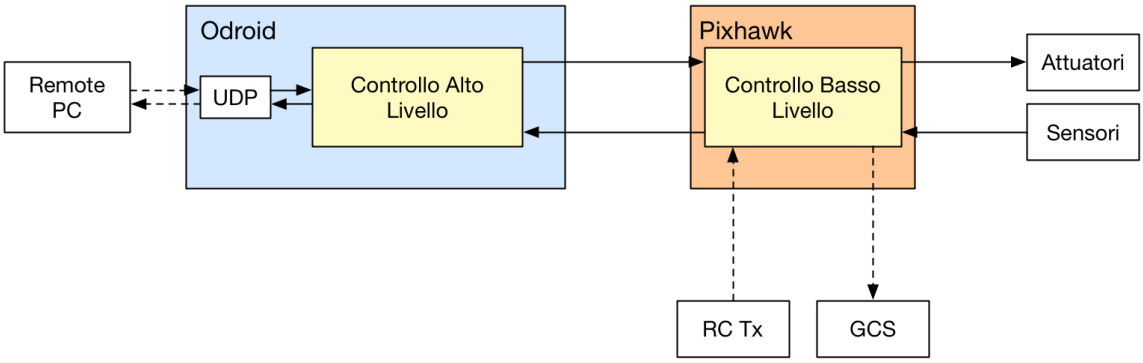

Figura 4.1: Suddivisione del controllo all'interno del sistema.

Nella Figura 4.1 viene rappresentato a livello logico come viene suddiviso il controllo fra i vari blocchi. Analizziamola partendo da sinistra:

- un calcolatore remoto invia dei comandi in termini di posizione al blocco di alto livello utilizzando il protocollo UDP (poichè molto leggero, veloce e flessibile);
- il modulo di alto livello riceve i comandi e li utilizza come riferimenti all'interno del proprio algoritmo di controllo;
- i nuovi comandi calcolati nel loop di controllo di alto livello vengono passati al modulo di basso livello tramite protocollo MAVLINK con messaggio di RC_OVERRIDE;
- il modulo di basso livello riceve i comandi e li utilizza come riferimento all'interno del proprio algoritmo di controllo;
- i valori finali dell'algoritmo vengono passati agli ESC che regolano i motori;

- comincia la retroazione del segnale: i sensori rilevano la posizione dell'UAV e, tramite la mediazione del modulo di basso livello, la passano al modulo di alto livello;

- il modulo di alto livello infine confronta il valore ricevuto con quello di riferimento e chiude il loop di controllo. Volendo può comunicare la posizione attuale al calcolatore remoto.

Il meccanismo di "safety switch", che è stato implementato nel capitolo precedente, ci permette di riprendere il controllo dell'UAV in qualsiasi momento. Tutto ciò torna veramente utile nelle fasi di test degli algoritmi di controllo poichè i valori simulati al computer possono essere molto diversi da quelli registrati nella realtà con conseguenti comportamenti inaspettati da parte dell'UAV.

Il bello di questa architettura è che un domani l'algoritmo di controllo di alto livello potrebbe essere spostato su un calcolatore lasciando il risultato invariato.

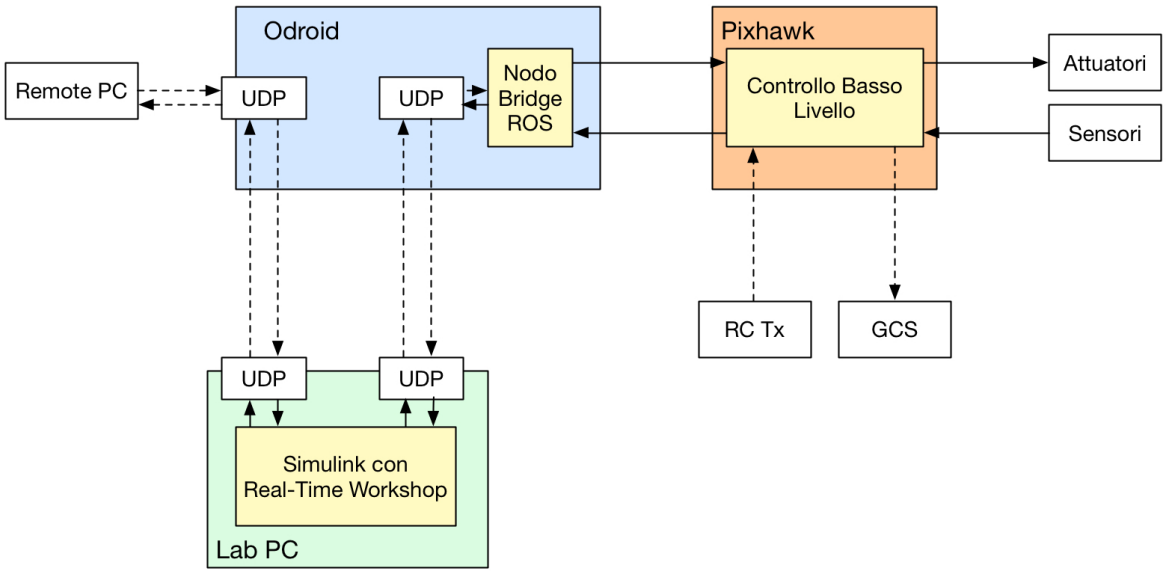

Figura 4.2: Suddivisione del controllo all'interno del sistema distribuito con controllo real-time.

In Figura 4.2 abbiamo spostato la parte di controllo ad alto livello su un calcolatore denominato "Lab PC". Questa struttura risulta più comoda della precedente dal momento che possiamo implementare la legge di controllo direttamente su Simulink, strumento che ci permette di lavorare facilmente

tramite blocchi di funzioni. Tale legge viene tradotta in codice, mandato poi in esecuzione in real time mediante il framework Real-Time Workshop.

4.2 Legge di Controllo

In questo momento ci concentreremo sullo studio di una legge di controllo. Prendiamo come riferimento il primo tipo di architettura (Figura 4.1) in cui il codice viene caricato fisicamente sul modulo di alto livello.

Ci occuperemo di trovare una legge di controllo che tenga conto dei limiti fisici del corpo rigido che stiamo considerando. Questa legge dovrà calcolare delle velocità e delle accelerazioni specifiche per l'UAV, data una traiettoria di riferimento passataci dal calcolatore remoto. La nostra uscita verrà passata al software APM Ardupilot nel modulo di basso livello. Ci affideremo ad una modalità di volo specifica, con relativo algoritmo di controllo implementato, denominata "Loiter", che prende in ingresso una velocità di riferimento.

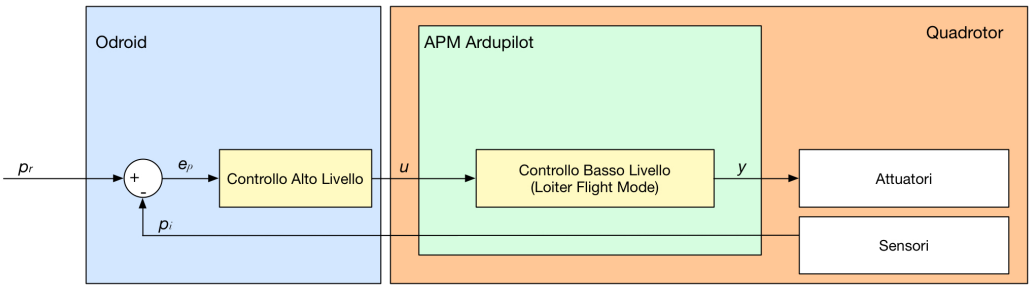

Figura 4.3: Loop di controllo a livello di sistema.

Per iniziare la progettazione del blocco di controllo di alto livello, bisogna studiare il problema dinamico che è alla base del sistema:

$$\begin{cases} m\dot{v}_I = \sum_{i=1}^{N} F_i^{ext} \\ \dot{p}_I = v_I \end{cases} \Rightarrow \begin{cases} \dot{v}_I = \frac{1}{m} \sum_{i=1}^{N} F_i^{ext} \\ \dot{p}_I = v_I \end{cases} \tag{4.1}$$

Data v_R come velocità di riferimento, definiamo l'errore di velocità come:

$$e_V = v_R - v_I$$

e quindi:

$$\dot{e}_V = \dot{v}_R - \dot{v}_I$$

usando la prima equazione della 4.1 otteniamo:

$$\dot{e}_V = \dot{v}_R - \frac{1}{m}\sum_{i=1}^{N} F_i^{ext}$$

Se assumiamo che il software APM Ardupilot riesca a produrre un errore di velocità $e_V = 0$ e relativo $\dot{e}_V = 0$, possiamo scrivere che $v_I = v_R$. Abbandoniamo quindi lo studio della dinamica (dato che è il software APM Ardupilot ad occuparsene) e ci concentriamo sulla seconda equazione della 4.1. Il nostro problema si è ridotto alla progettazione di un blocco che controlli un Plant (oggetto da controllare) che ingloba il software APM e i suoi controlli. Questo blocco lo tratteremo come un integratore che, passatagli una velocità controllata, restituisce la velocità inerziale dell'UAV che ci permette di chiudere il loop di controllo.

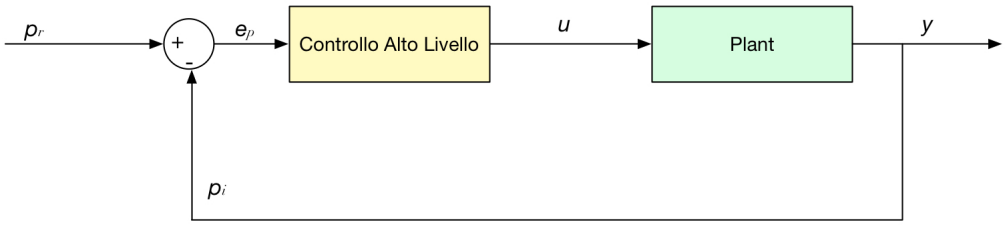

Figura 4.4: Loop di controllo a livello di sistema basandoci sulle considerazioni fatte.

Il sistema viene descritto mediante rappresentazione dello spazio degli stati:

$$\begin{cases} \dot{\boldsymbol{p}}_{\boldsymbol{I}} = \boldsymbol{v}_{\boldsymbol{I}} = \boldsymbol{v}_{\boldsymbol{R}} = \boldsymbol{u} \\ \boldsymbol{y} = \boldsymbol{p}_{\boldsymbol{I}} \end{cases} \tag{4.2}$$

Il blocco che andremo a progettare può essere scomposto in due parti:

- **Controllore**: implementa la legge di controllo;
- **Filtro**: modella il segnale in uscita adattandolo al limite fisico dell'UAV.

4.2.1 Controllore

Iniziamo con lo studio del controllore. Il nostro obiettivo è quello di **avere a regime un errore di posizione nullo**:

$$\lim_{t\to\infty} e_P(t) = 0$$

L'obiettivo di controllo può essere tradotto in termini di trasformate di Laplace, utilizzando il *"Teorema del valore finale"*:

$$\lim_{s\to 0} sE_P(s) = 0$$

dove s è la variabile complessa e $E_P(s)$ è il segnale $e_P(t)$ trasformato con Laplace. Grazie alle precedenti equazioni possiamo studiare il segnale errore sia in funzione del tempo che in funzione delle variabili complesse.

Riprendendo lo state space (Equazione 4.2) possiamo suddividere la legge di controllo in due componenti:

$$u_P = u_{P_S} + u_{P_T}$$

dove $\boldsymbol{u_{P_S}}$ è la componente di **stabilità** e $\boldsymbol{u_{P_T}}$ è la componente di **tracking**. Quindi la 4.2 può essere riscritta:

$$\begin{cases} \dot{p}_I = u_{P_S} + u_{P_T} \\ y = p_I \end{cases} \tag{4.3}$$

Possiamo scrivere a questo punto l'**errore di posizione** in funzione della prima della 4.3:

$$\dot{e}_P = \dot{p}_R - (u_{P_S} + u_{P_T}) \tag{4.4}$$

e trasformandola avremo:

$$sE_P = sP_R - (U_{P_S} + U_{P_T}) \tag{4.5}$$

Possiamo calcolarci molto facilmente $\boldsymbol{u_{P_T}}$ lavorando sulla 4.4 e ponendo $\dot{e}_P = 0$ e $u_{P_S} = 0$:

$$\boldsymbol{u_{P_T} = \dot{p}_R} \tag{4.6}$$

Per trovare il giusto valore di $\boldsymbol{u_{P_S}}$ mettiamo a sistema la trasformata della 4.6 e la 4.5:

$$\begin{cases} sE_P = sP_R - (U_{P_S} + U_{P_T}) \\ U_{P_T} = sP_R \end{cases} \quad \Rightarrow sE_P = -U_{P_S}$$

Possiamo riscrivere la precedente equazione con un disturbo $d(t)$ costante sull'ingresso:

$$sE_P = -U_{P_S} + \frac{C}{s} \tag{4.7}$$

dove $\frac{C}{s}$ è proprio la trasformata del disturbo costante $d(t)$. Arrivati a questo punto possiamo confrontare il valore ottenuto con dei tipi di controllo come ad

esempio il **P, PI e PID (P: proporzionale, I: integrale, D: derivativo)**. Nell'ipotesi di usare un controllore di tipo **P**, verifichiamo se l'errore a regime si annulla. Partiamo mettendo a sistema la 4.7 e l'equazione tipica di un controllore P:

$$\begin{cases} sE_P = -U_{P_S} + \frac{C}{s} \\ U_{P_S} = K_P E_P + e_P(0^-) \end{cases}$$

dove $e_P(0^-)$ è l'errore all'istante 0^- e K_P è il parametro di progetto. Sostituendo:

$$sE_P = -K_P E_P + \frac{C}{s} - e_P(0^-) \Rightarrow (s + K_P)E_P = \frac{C}{s} - e_P(0^-)$$
$$\Rightarrow E_P = \frac{C}{s(s + K_P)} - \frac{e_P(0^-)}{s + K_P}$$

e applicando il *"Teorema del valore finale"* otteniamo:

$$\lim_{s\to 0} sE_P = \lim_{s\to 0} \frac{C}{s + K_P} - \frac{s}{s + K_P} e_P(0^-) = \frac{C}{K_P} \tag{4.8}$$

Dal risultato precedente capiamo che questo tipo di controllo, in presenza di disturbi costanti in ingresso, ci porterà ad avere un errore a regime pari a $\frac{C}{K_P}$. Questo vuol dire che, pur mantenendo il sistema stabile, non si arriverà a regime al valore di riferimento. Per questo lo scartiamo.

Analizziamo ora il controllore di tipo **PI** e verifichiamo se in questo caso l'errore a regime si annulla. Partiamo mettendo di nuovo a sistema la 4.7 e l'equazione tipica di un controllore PI:

$$\begin{cases} sE_P = -U_{P_S} + \frac{C}{s} \\ U_{P_S} = (K_P + \frac{K_I}{s})E_P + e_P(0^-) \end{cases}$$

dove K_P e K_I sono i parametri di progetto. Sostituendo:

$$sE_P = -\left(K_P + \frac{K_I}{s}\right) E_P + \frac{C}{s} - e_P(0^-) \Rightarrow \left(s + K_P + \frac{K_I}{s}\right) E_P = \frac{C - e_P(0^-)s}{s}$$
$$\Rightarrow E_P = \frac{C - e_P(0^-)s}{s^2 + K_P s + K_I}$$

e applicando il *"Teorema del valore finale"* otteniamo:

$$\lim_{s\to 0} sE_P = \frac{-e_P(0^-)s^2 + Cs}{s^2 + K_P s + K_I} = 0 \tag{4.9}$$

In questo caso l'errore a regime si annulla. Abbiamo così verificato che possiamo utilizzare un controllore PI per ottenere un sistema stabile e con errore a regime nullo.

Riassemblando le due componenti di u_P otteniamo:

$$\boldsymbol{u_P = u_{P_S} + u_{P_T} = K_P e_P + K_I \int_{\tau} e_P d\tau + \dot{p_R}} \tag{4.10}$$

4.2.2 Filtro

Il filtro che progetteremo serve per modellare la velocità comandata, in uscita dal controllore, in maniera tale che possa adattarsi ai limiti fisici dell'UAV che stiamo considerando. Questo **"Filtro Formatore"** ha le seguenti proprietà:

- limita il comando d'ingresso e la sua derivata (nel nostro caso limita la velocità u_P e la sua derivata);
- limita la banda passante comportandosi come un filtro passa-basso;
- fa sì che non ci siano discontinuità nel segnale.

La sua espressione è la seguente:

$$\begin{cases} \begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & -2\delta\omega_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ \omega_n^2 \end{bmatrix} sat\left(e_u, L_u^{min}, L_u^{MAX}\right) \\ \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \end{cases} \tag{4.11}$$

con e_U espresso nel seguente modo:

$$e_u = sat\left(u, u^{min}, u^{MAX}\right) - x_1 \tag{4.12}$$

ed i due valori di saturazione nella prima della 4.11 sono espressi come:

$$\begin{cases} L_u^{min} = \frac{2\delta}{\omega_n} L^{min} \\ L_u^{MAX} = \frac{2\delta}{\omega_n} L^{MAX} \end{cases}$$

Analizzando l'equazioni che sono state scritte in precedenza, possiamo intuire il comportamento del filtro:

- inizialmente viene saturato il valore in ingresso (nel nostro caso u_P) tra due valori limite;
- il valore saturato viene sommato con la velocità che uscirà dal filtro cambiata di segno;

- viene saturato nuovamente il segnale tra due valori limite;
- questo segnale entrerà nel sistema rappresentato dall'equazione 4.11;
- in uscita avremo a disposizione due segnali: la velocità e l'accelerazione entrambe filtrate.

4.2.3 Progetto dei Parametri di Funzionamento

Il risultato di quanto abbiamo appena descritto in termini matematici può essere rappresentato con uno schema a blocchi:

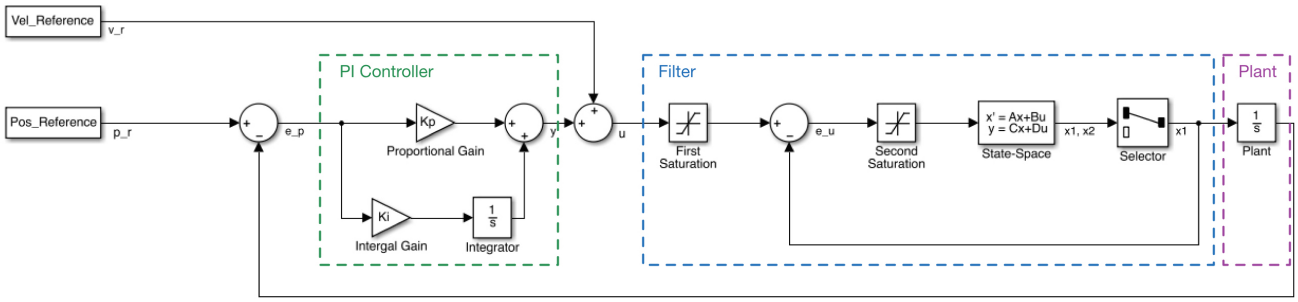

Figura 4.5: Schema a blocchi del controllo di alto livello.

Lo step successivo è quello di definire dei valori appropriati per i parametri che compaiono nelle formule precedenti per poter raggiungere il nostro goal. Diversi valori verranno stimati in base ai limiti fisici dell'UAV che abbiamo a disposizione. È comunque possibile riadattarli al sistema da controllare.

Immaginiamo di avere in ingresso, come posizione di riferimento, una sinusoide di ampiezza A e di pulsazione ω. Se ad esempio questa sinusoide rappresenta la coordinata x del sistema di riferimento dell'UAV e la sua pulsazione ω è molto elevata, in termini pratici stiamo comandando quest'ultimo a muoversi molto velocemente avanti e indietro sull'asse x. Dal momento che fisicamente l'UAV non riesce a seguirlo, abbiamo bisogno di determinare il limite al quale bisogna attenersi. Il ragionamento più logico da seguire è quello di imporre una pulsazione massima tale che:

$$\omega \leq \omega_{max}$$

Nel nostro caso potremmo imporre sperimentalmente una $\omega_{max} = 6rad/s$ dato che corrispondono a $0.95 \simeq 1Hz$. Il filtro dovrà agire almeno ad una pulsazione dieci volte più grande della ω_{max} per fare in modo di tagliare via le componenti ad alte frequenze lasciando inalterate le ampiezze e le fasi della risposta del sistema relativa all' input alla pulsazione ω_{max}.

I valori che dobbiamo ancora fissare riguardano:

- la δ all'interno dell'equazione del filtro;
- le costanti del controllore PI (K_P e K_I);
- i limiti di saturazione (u^{min},u^{MAX},L_u^{min} e L_u^{MAX}).

Definiamo δ con il valore ottimale per evitare picchi di risonanza. La teoria dei controlli automatici ci insegna che per ottenere queste condizioni dobbiamo imporre:

$$\delta = 0.707 \tag{4.13}$$

Per progettare i valori che le costanti del controllore PI devono avere, ricaviamoci le tre **funzioni di trasferimento** dei relativi blocchi.

Iniziamo con il blocco di controllo lavorando esclusivamente sul controllo PI senza la componente di feed forward (ovvero la $\dot{p}_R$). In questo caso abbiamo già a disposizione la FdT:

$$G_C = \frac{(K_P s + K_I)}{s} = K_I \frac{(1 + \tau s)}{s} \tag{4.14}$$

con

$$\tau = \frac{K_P}{K_I} \tag{4.15}$$

Passiamo al blocco di filtraggio trascurando la saturazione:

$$\begin{cases} \dot{x}_1 = x_2 \\ \dot{x}_2 = -2\delta\omega_n x_2 + \omega_n^2 u - \omega_n^2 x1 \end{cases} \Rightarrow \begin{cases} sX_1 = X_2 \\ s^2 X_1 = -2\delta\omega_n s X_1 + \omega_n^2 U - \omega_n^2 X1 \end{cases}$$

$$\Rightarrow \left(s^2 + 2\delta\omega_n s + \omega_n^2\right) X_1 = \omega_n^2 U$$

Se definiamo $G_F = \frac{X_1}{U}$ allora otteniamo:

$$G_F = \frac{\omega_n^2}{s^2 + 2\delta\omega_n s + \omega_n^2} \tag{4.16}$$

Il blocco del plant viene descritto come un integratore, che può essere espresso semplicemente nel seguente modo:

$$G_P = \frac{1}{s} \tag{4.17}$$

Se facciamo la serie dei tre blocchi otteniamo il blocco risultante G_{CFP} in catena diretta che assume la seguente forma:

$$G_{CFP} = \frac{K_I \omega_n^2 (1 + \tau s)}{s^2 (s^2 + 2\delta\omega_n s + \omega_n^2)} \tag{4.18}$$

Se poniamo $K = K_I \omega_n^2$, tramite il luogo delle radici possiamo determinare i valori di K per i quali il sistema risulta stabile.

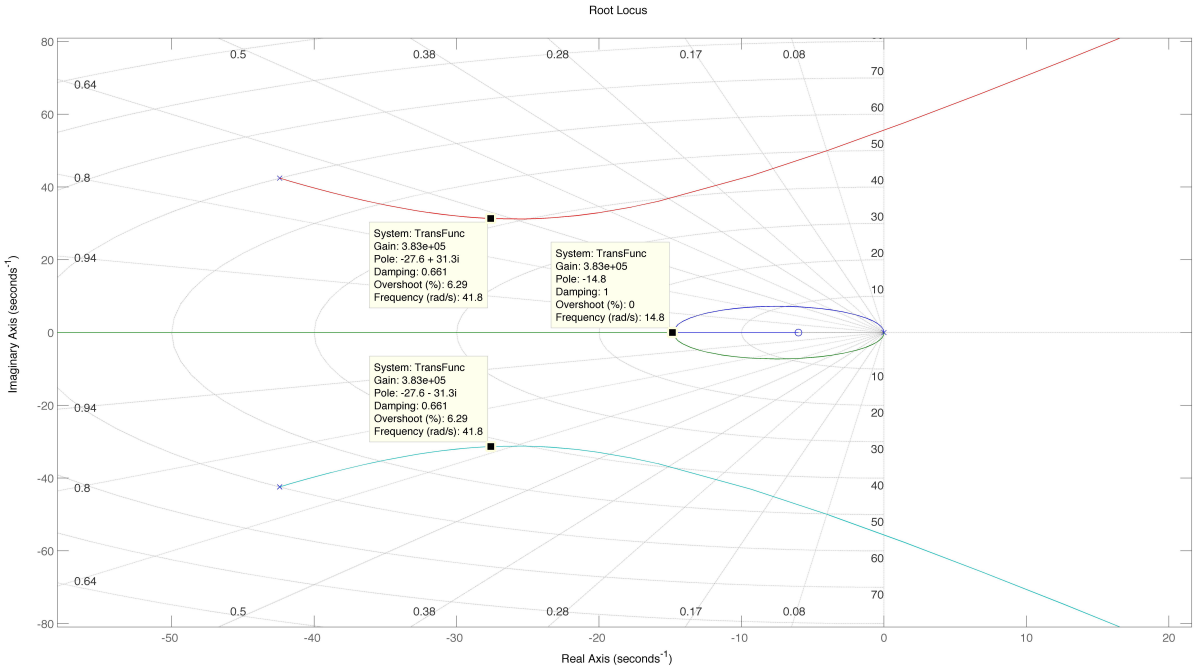

Figura 4.6: Luogo delle Radici del sistema in catena aperta.

Analizzando il diagramma 4.6, notiamo che il valore di K ottimale per mantenere stabile il sistema equivale a 3.83×10^5. Ricordandoci le relazioni precedenti otteniamo per sostituzione:

$$\begin{cases} K_I = \frac{K}{\omega_n^2} \\ K_P = \tau K_I \\ \tau = \frac{10}{\omega_n} \\ \omega_{max} = \frac{1}{\tau} = 6 rad/s \end{cases} \Rightarrow \begin{cases} K_I = \frac{3.83 \times 10^5}{3600} = 106.3 \\ K_P = \frac{K_I}{6} = \frac{106.3}{6} = 17.71 \end{cases} \tag{4.19}$$

Il **diagramma di Bode** associato a questi tre blocchi e alla loro serie è il seguente:

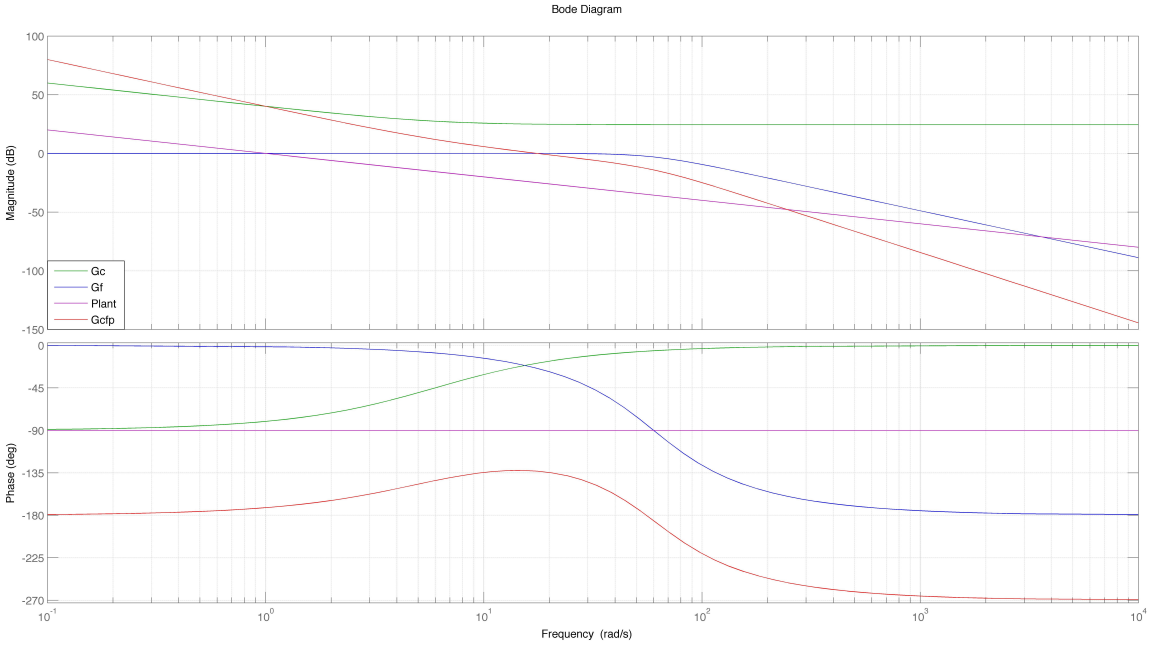

Figura 4.7: Diagramma di Bode della funzione di guadagno d'anello

Il **diagramma di Nyquist** associato alla serie dei tre blocchi è invece il seguente:

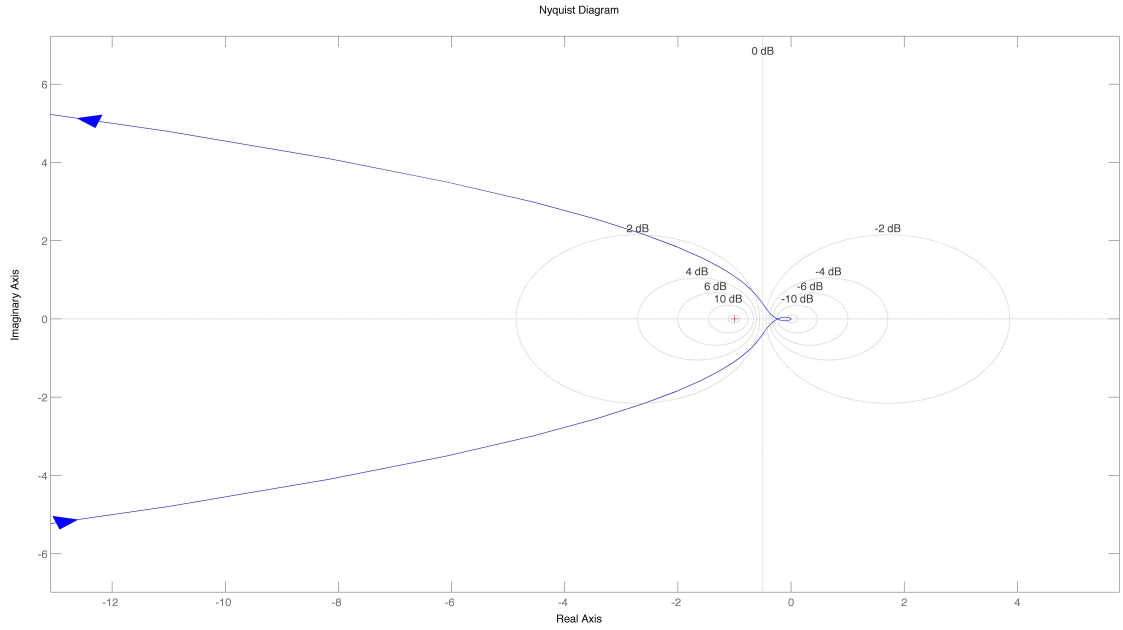

Figura 4.8: Diagramma di Nyquist della funzione di guadagno d'anello

Da quest'ultimo verifichiamo che il sistema è stabile tramite l'utilizzo del ***"Criterio di Nyquist"***:
condizione necessaria e sufficiente perchè questo sistema di controllo sia asintoticamente stabile, ossia affinchè la funzione di trasferimento del sistema in retroazione abbia tutti poli a parte reale negativa, è che il diagramma polare completo della funzione di risposta armonica $G_{CFP}(j\omega)$ circondi il punto critico $(-1+j0)$, ossia il punto -1 sull'asse reale, tante volte in senso antiorario quanti sono i poli di $G_{CFP}(s)$ con parte reale positiva (cioè quanti sono i poli instabili).

Infatti, prendendo come riferimento il diagramma polare completo (ovvero lo stesso diagramma in figura 4.8 chiuso in senso orario dall'alto verso il basso all'infinito) non otteniamo nessun giro intorno al punto critico $(-1+j0)$ e allo stesso tempo non abbiamo nessun polo a parte reale positiva.

Per i valori da assegnare a u^{min},u^{MAX},L_u^{min} e L_u^{MAX} ci siamo affidati a dati sperimentali sull'UAV. In particolare se imponiamo come limite del primo saturatore $u^{min} = -3m/s$ e $u^{MAX} = 3m/s$, stiamo imponendo all'UAV di non superare la velocità in modulo di $3m/s$. Da qui possiamo determinare il secondo valore limite. Dovremmo limitare l'errore di velocità (definito come differenza fra la velocità comandata saturata e quella in uscita dal filtro) a non superare il range imposto dal saturatore precedente. Quindi possiamo scrivere:

$$\begin{cases} |L_u| = 2\,|u_{sat}| \\ |L_u| = \frac{2\delta}{\omega_n}\,|L| \end{cases} \Rightarrow |L| = \frac{\omega_n\,|u_{sat}|}{\delta} = \frac{60 * 3}{0.707} = 254.59$$

Il valore di nostro interesse deriva direttamente dalla prima equazione del precedente sistema e quindi $L_u^{min} = -6$ e $L_u^{MAX} = 6$. Dal momento che vi è una dipendenza da ω_n e δ è bene trovarsi per sostituzione i valori di L^{min} e L^{MAX} che sono $L^{min} = -254.59$ e $L_u^{MAX} = 254.59$ ed hanno le dimensioni di un' accelerazione.

4.2.4 Anti-Windup

Nonostante il nostro sistema sia stato progettato correttamente, vi è un problema legato alla coesistenza all'interno del sistema tra l'integratore del PI ed i due saturatori. In presenza di saturazioni accade che l'integratore continui ad accrescere il suo contributo dal momento che integra un errore che non può smaltire a causa delle saturazioni. Nell'ipotesi in cui il saturatore si disattivi, questa "carica" accumulata dall'integratore potrebbe essere troppo grande e portare ad un'azione di controllo errata poichè non più legata al sistema sotto controllo. A questo punto si rischia l'instabilità. questo

fenomeno è chiamato **windup**. Per ovviare a questo problema, si scarica l'integratore evitando così, in presenza di lunghi periodi di saturazione, di portare il sistema alla deriva.

Una tecnica di **anti-windup** molto utilizzata è la seguente: il valore in ingresso al saturatore viene sottratto alla relativa uscita, amplificato attraverso una costante K positiva e sommato al segnale di ingresso nell'integratore del PI. In questo modo se vi sono delle saturazioni, la differenza fra il valore di uscita e di ingresso è un valore negativo che aumenta tramite la costante di guadagno positiva. In seguito viene sommato con un valore di errore precedente, amplificato tramite la costante K_I, e dato in pasto all'integratore. Questo fa sì che più il valore di uscita dal controllore viene saturato, più l'integratore viene scaricato in fretta. L'azione dell'anti-windup si può tradurre come una forzante che porta a zero l'errore integrale, che a sua volta rappresenta la condizione iniziale (dell'integratore) nell'istante in cui la saturazione viene disattivata.

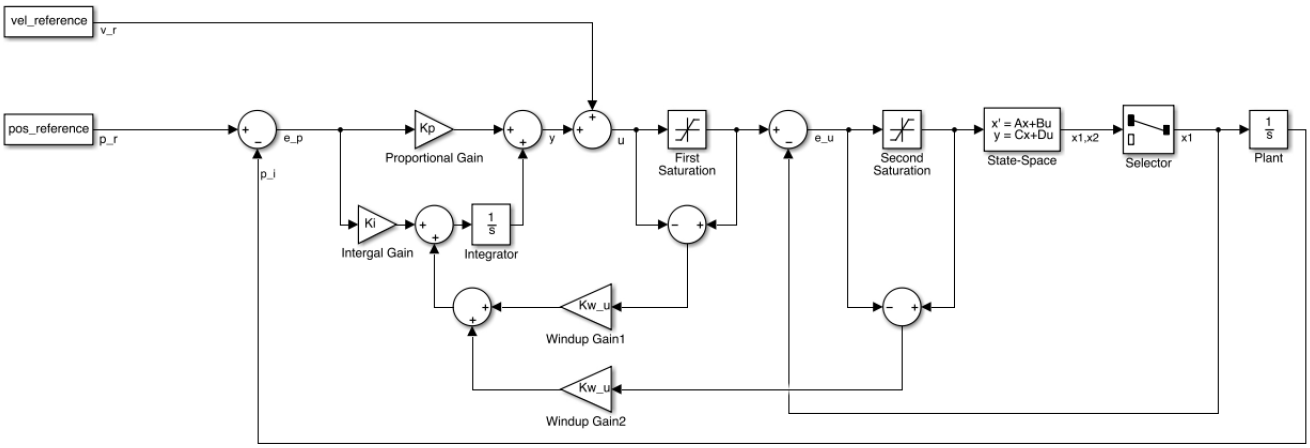

Figura 4.9: Schema a blocchi del controllo di alto livello con tecnica di Anti-Windup.

L'unica cosa che ci resta da fare è determinare un valore di K per la costante dell'anti-windup, adeguato al nostro sistema. Prendiamo come riferimento il sistema in figura 4.10 con ingresso nullo ed errore iniziale $e(0^-)$.

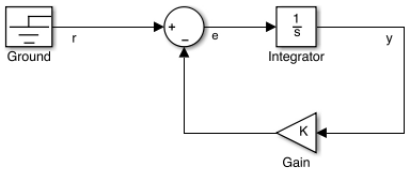

Figura 4.10: Controllo con Integratore.

Studiando il sistema otteniamo:

$$\begin{cases} Y = \frac{E}{s} \\ E = e(0^-) - KY \end{cases} \Rightarrow \left(1 + \frac{K}{s}\right) Y = \frac{e(0^-)}{s} \Rightarrow Y = \frac{e(0^-)}{s+K}$$

che può essere riscritta come:

$$\begin{cases} Y = \frac{e(0^-)}{1+\tau s} \\ \tau = \frac{1}{K} \end{cases}$$

Dallo studio delle risposte canoniche sappiamo che il tempo di assestamento al 5% può essere scritto come:

$$t_a = 3\tau$$

ed andando a sostituire la nostra τ otteniamo:

$$t_a = \frac{3}{K} \Rightarrow K = \frac{3}{t_a}$$

Perciò scopriamo che K è inversamente proporzionale al tempo di assestamento ovvero aumentando K diminuiamo il tempo di assestamento della risposta. Dal momento che il ciclo può essere eseguito sul modulo di alto livello con una frequenza di 1Hz, provando a fissare un tempo di assestamento $t_a = 1s$ otterremo $K = 3$. Questo progetto può essere utilizzato per entrambi gli anti-windup dei saturatori.

4.2.5 Risultati Finali

Una volta progettato tutto il blocco di controllo di alto livello possiamo studiare la funzione di trasferimento di tutto il blocco in retroazione.

Riprendendo il grafico 4.7 abbiamo:

$$\begin{cases} P_I = \frac{1}{s} X_1 \\ X_1 = G_F(s) U \\ U = s P_R + G_C(s) E_P \\ E_P = P_R - P_I \end{cases} \Rightarrow$$

$$P_I = \frac{1}{s} G_F(s) \left[s P_R + G_C(s) \left(P_R - P_I \right) \right] = G_F(s) P_R + \frac{1}{s} G_F(s) G_C(s) \left(P_R - P_I \right)$$

$$\Rightarrow \left[1 + \frac{1}{s} G_F(s) G_C(s) \right] P_I = G_F(s) \left[1 + \frac{1}{s} G_C(s) \right] P_R$$

che infine portano alla funzione di trasferimento di tutto il blocco $G_e(s)$:

$$G_e(s) = \frac{P_I}{P_R} = \frac{G_F(s) \left(1 + \frac{1}{s} G_C(s) \right)}{1 + \frac{1}{s} G_C(s) G_F(s)} \tag{4.20}$$

Sostituendo la 4.14 e la 4.16 otteniamo:

$$G_e(s) = \frac{\omega_n^2 \left(s^2 + K_P s + K_I \right)}{s^2 \left(s^2 + 2\delta\omega_n s + \omega_n^2 \right) + \omega_n^2 \left(K_P s + K_I \right)} =$$

$$= \frac{\omega_n^2 s^2 + K_P \omega_n^2 s + K_I \omega_n^2}{s^4 + 2\delta\omega_n s^3 + \omega_n^2 s^2 + \omega_n^2 K_P s + \omega_n^2 K_I}$$

Proviamo ora a graficare il diagramma di Bode della funzione di trasferimento precedente.

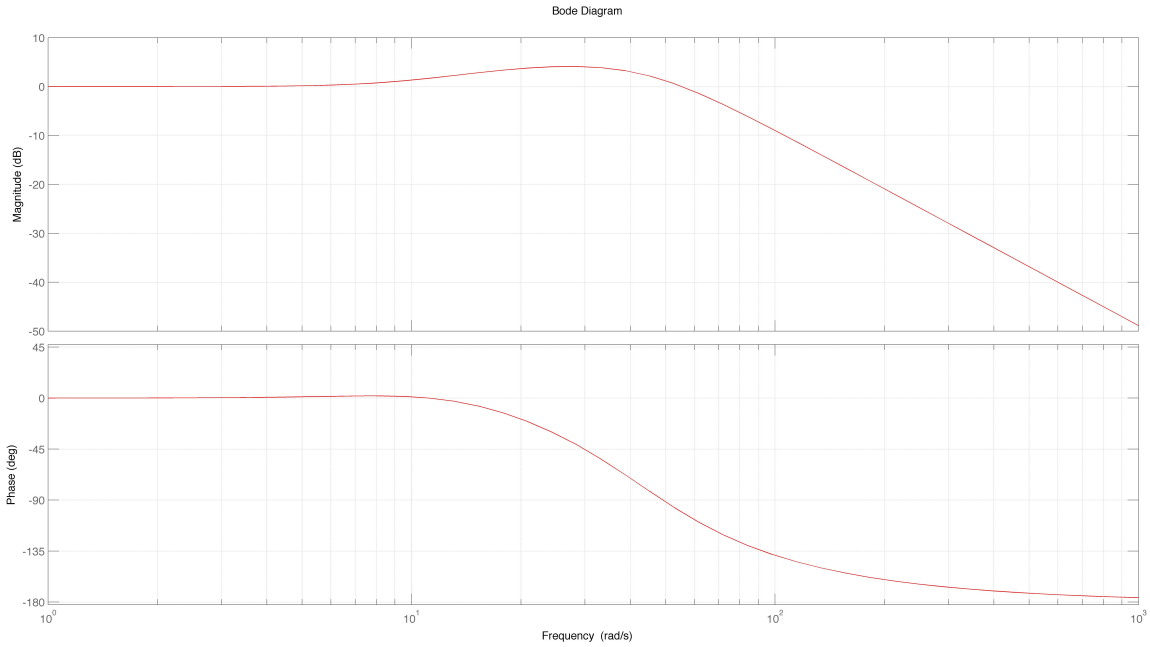

Figura 4.11: Diagramma di Bode del sistema in retroazione.

Analizzando il diagramma di Bode, notiamo che:

- fino alla pulsazione $\omega \simeq 6 rad/s$ il sistema porta in uscita esattamente ciò che ha in ingresso, senza ritardi;
- dalla pulsazione $\omega \simeq 6 rad/s$ alla pulsazione $\omega \simeq 10 rad/s$ il sistema amplifica leggermente l'ingresso mantenendo l'uscita ancora in fase;
- dalla pulsazione $\omega \simeq 10 rad/s$ il sistema comincia ad amplificare l'ingresso ed a sfasare in ritardo fino ad arrivare intorno ad un guadagno massimo di $4.38 dB$ ed una fase di $-39.1°$ nella pulsazione $\omega \simeq 28 rad/s$;
- dalla pulsazione $\omega \simeq 28 rad/s$ in poi il sistema comincia ad attenuare con pendenza $-20 dB/dec$ per poi aumentarla a $-40 dB/dec$ in prossimità della pulsazione $\omega \simeq 60 rad/s$ continuando a sfasare fino a raggiungere un minimo di $-180°$.

Questo ci porta a dire che il sistema non ha problemi ad inseguire il riferimento fin quando riceve segnali con pulsazioni non troppo elevate (fino ad un massimo $\omega \simeq 10 rad/s$). Oltre questo valore comincia ad accumulare ritardo e a non seguire precisamente il riferimento, attenuando via via l'ingresso.

4.3 Simulazione

Una volta finito il progetto, è stato realizzato su MATLAB/Simulink. Per simulare il sistema, abbiamo utilizzato come posizione di riferimento un segnale sinusoidale che derivato ci fornisce la velocità di riferimento. Nello scenario reale i due valori ci verranno passati separatamente poichè la presenza di rumori ad alta frequenza sul segnale di posizione renderebbe troppo piccolo il rapporto segnale-rumore della derivata.

In seguito riporteremo lo schema completo, attrezzato per la simulazione con la sinusoide:

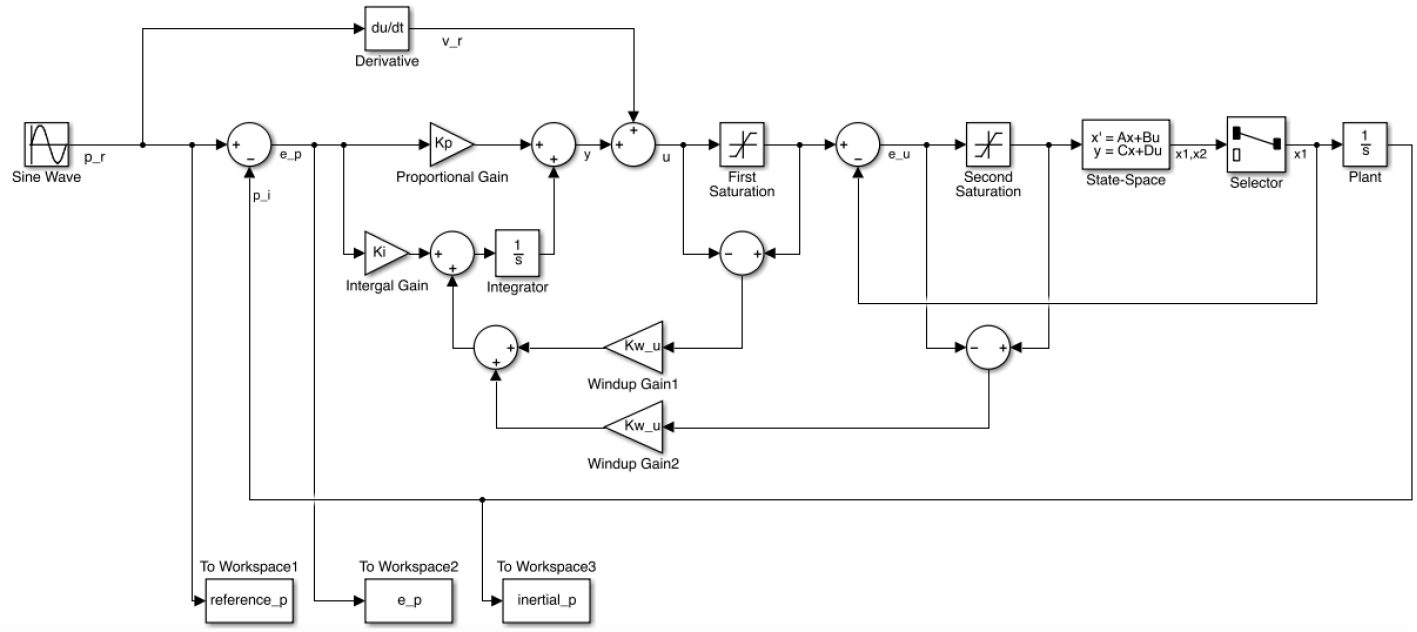

Figura 4.12: Schema della simulazione del sistema.

I tre blocchi che compaiono nella parte bassa della Figura 4.21 ci servono per tracciare le simulazioni di tre valori: la posizione di riferimento, la posizione inerziale in uscita dal plant e l'errore di posizione.

Riportiamo per semplicità l'equazioni della posizione e della sua derivata che ci torneranno utili in qualche simulazione successiva:

$$\begin{cases} p_R(t) = A\sin(\omega t) \\ v_R(t) = \omega A\cos(\omega t) \end{cases} \tag{4.21}$$

Seguiranno una serie di test basati sulla variazione dell'ampiezza A e della pulsazione ω eliminando momentaneamente i saturatori per vedere come reagisce normalmente il sistema progettato:

1. Poniamo un valore di ampiezza unitario $A = 1$ ed una pulsazione $\omega = 1rad/s$ alla sinusoide. Siccome ω è minore dell' $\omega_{max} = 6rad/s$, l'uscita del blocco sarà il più possibile uguale all'ingresso.

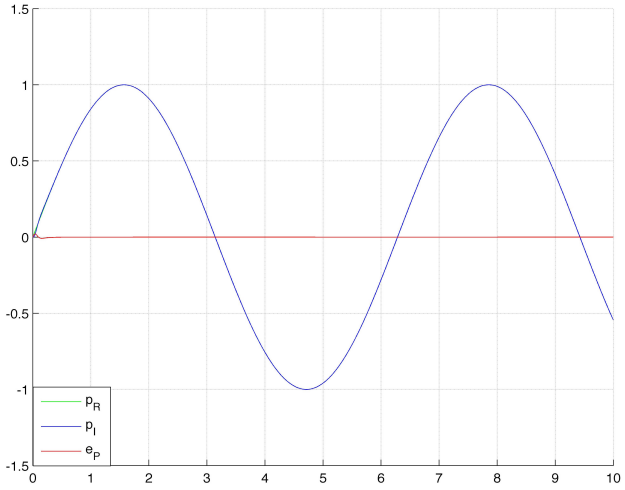

Figura 4.13: Simulazione con $p_R(t) = \sin(t)$.

2. Poniamo un valore di ampiezza unitario $A = 1$ ed una pulsazione $\omega = 7rad/s$ alla sinusoide. In questo caso verrà generata in uscita un'onda con un'ampiezza leggermente maggiore rispetto a quella d'ingresso senza ritardi.

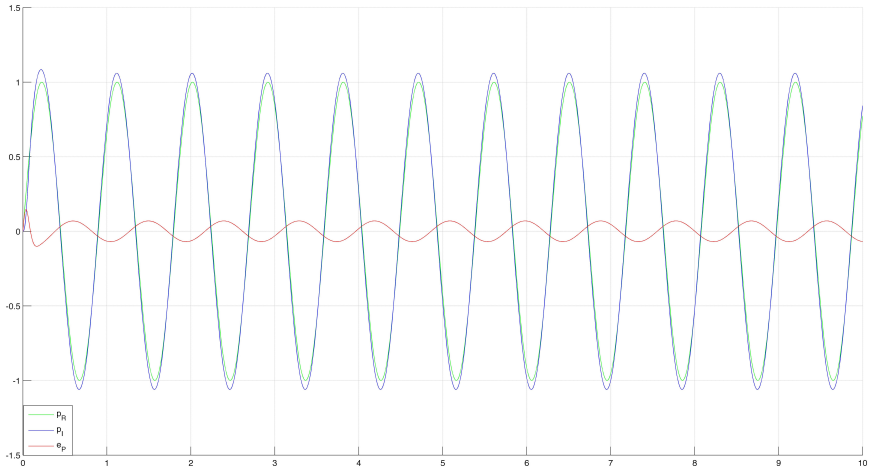

Figura 4.14: Simulazione con $p_R(t) = \sin(7t)$.

3. Poniamo un valore di ampiezza unitario $A = 1$ ed una pulsazione $\omega = 30 rad/s$ alla sinusoide. In questo caso in uscita avremo un'onda amplificata di circa $4dB$ e sfasata di $-45°$.

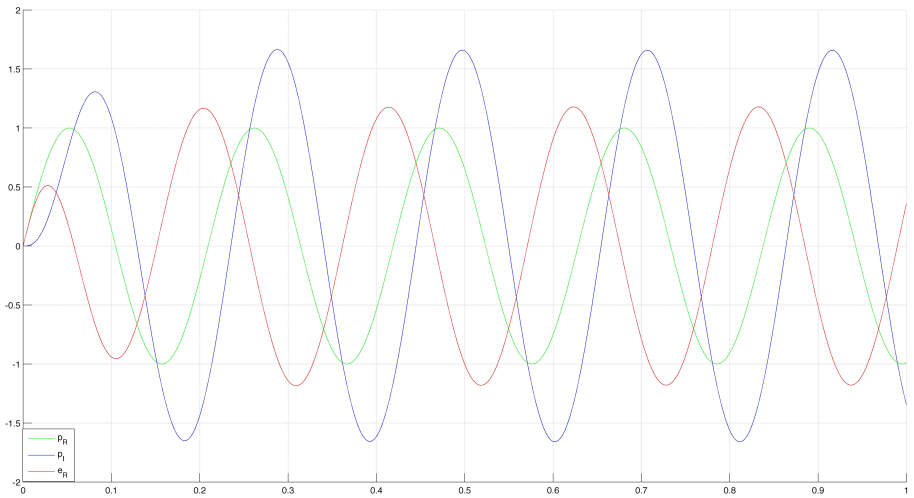

Figura 4.15: Simulazione con $p_R(t) = \sin(30t)$.

4. Poniamo un valore di ampiezza unitario $A = 1$ ed una pulsazione $\omega = 100 rad/s$ alla sinusoide. In questo caso in uscita avremo un'onda attenuata di circa $-10dB$ e sfasata di circa $-135°$.

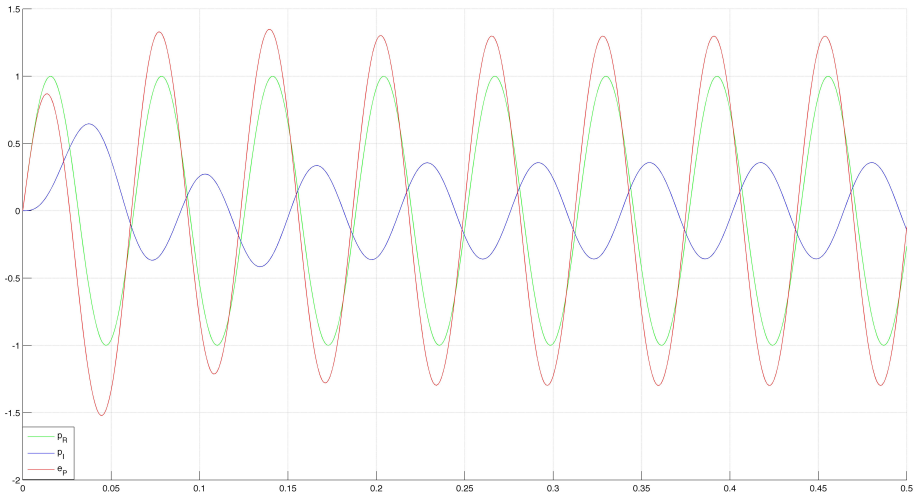

Figura 4.16: Simulazione con $p_R(t) = \sin(100t)$.

Introduciamo ora i saturatori per vedere come variano le uscite con gli stessi ingressi dei test precedenti:

1. Poniamo un valore di ampiezza unitario $A = 1$ ed una pulsazione $\omega = 1rad/s$ alla sinusoide. La velocità di riferimento iniziale sarà perciò $v_R = \omega A = 1m/s$. Dal momento che il nostro limite di velocità è stato impostato sui $3m/s$ tutto rimane nella norma, quindi avremo lo stesso grafico della Figura 4.13.

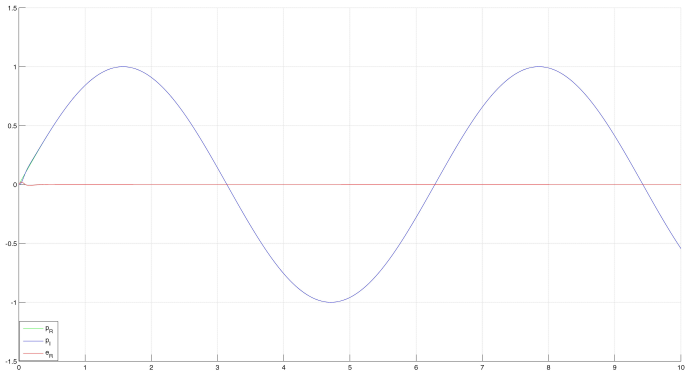

Figura 4.17: Simulazione con $p_R(t) = \sin(t)$ e saturatori attivi.

2. Poniamo un valore di ampiezza unitario $A = 1$ ed una pulsazione $\omega = 7rad/s$ alla sinusoide. La velocità di riferimento iniziale (e massima) sarà perciò $v_R = \omega A = 7m/s$. Dal momento che il nostro limite di velocità è stato impostato sui $3m/s$ il risultato varierà rispetto alla Figura 4.14 a causa dei saturatori che entrando in azione ritardano l'uscita attenuandola allo stesso tempo.

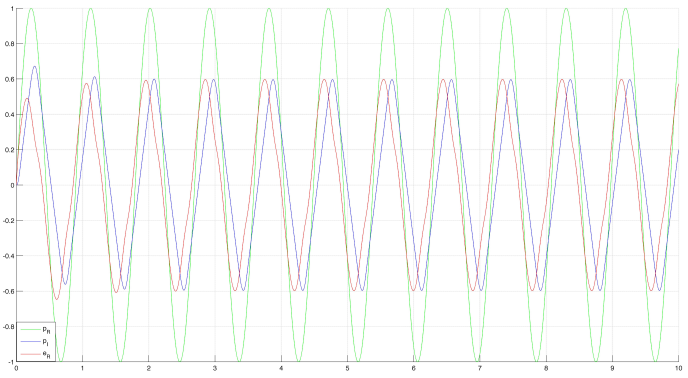

Figura 4.18: Simulazione con $p_R(t) = \sin(7t)$ e saturatori attivi.

3. Poniamo un valore di ampiezza unitario $A = 1$ ed una pulsazione $\omega = 30rad/s$ alla sinusoide. La velocità di riferimento iniziale (e massima) sarà perciò $v_R = \omega A = 30m/s$. Dal momento che il nostro limite di velocità è stato impostato sui $3m/s$ il risultato varierà rispetto alla Figura 4.15 a causa dei saturatori che entrando in azione attenuano di molto l'uscita mantenendo invariato lo sfasamento.

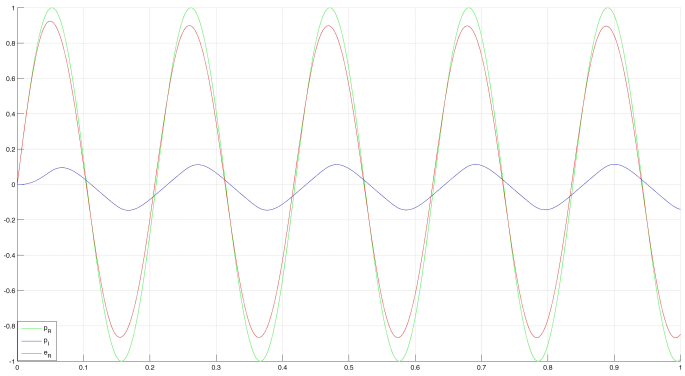

Figura 4.19: Simulazione con $p_R(t) = \sin(30t)$ e saturatori attivi.

4. Poniamo un valore di ampiezza unitario $A = 1$ ed una pulsazione $\omega = 100rad/s$ alla sinusoide. La velocità di riferimento iniziale (e massima) sarà perciò $v_R = \omega A = 100m/s$. Dal momento che il nostro limite di velocità è stato impostato sui $3m/s$ il risultato varierà rispetto alla Figura 4.16 a causa dei saturatori che entrando in azione attenuano moltissimo l'uscita mantenendo l'UAV quasi del tutto fermo e lasciando invariato lo sfasamento.

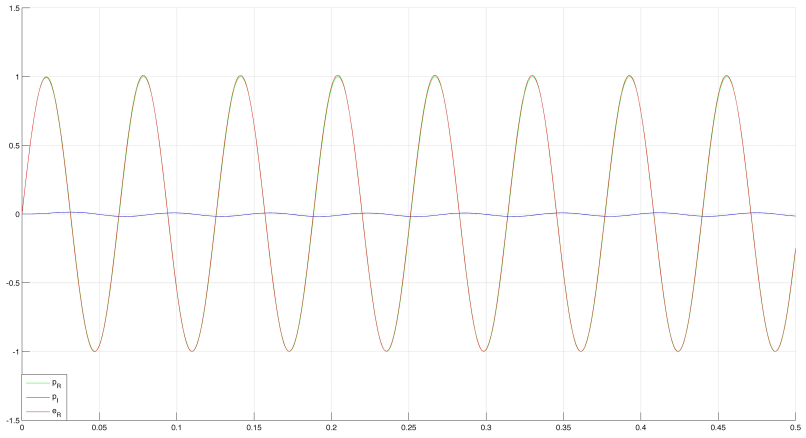

Figura 4.20: Simulazione con $p_R(t) = \sin(100t)$ e saturatori attivi.

Tramite questi test abbiamo simulato il comportamento dell' UAV con il nostro blocco di controllo ad alto livello. Ovviamente nella realtà i dati non saranno così precisi come nelle precedenti simulazioni. Questo modello inoltre semplifica molto la realtà poichè il Plant, che per noi è un semplice integratore, all'interno è formato da tutto ciò che si trova nel blocco "Quadrotor" in figura 4.3. Perciò dobbiamo tener conto di tutti i ritardi che possono essere accumulati all'interno del software APM Ardupilot e di tutti i ritardi di trasmissione che possono influire sul risultato finale.

Conclusioni

Il progetto portato avanti in questo elaborato mi ha dato l'opportunità di approfondire e mettere in pratica le conoscenze acquisite in ambito informatico, elettronico e delle telecomunicazioni confrontandole con quelle dell'ingegneria industriale. In particolare ho avuto modo di mettermi in gioco su aspetti che finora avevo studiato solo in maniera teorica.

È stato molto interessante e stimolante aver partecipato a questo progetto europeo e saper di aver contribuito, anche se in minima parte, nella realizzazione di un sistema di sicurezza e di una legge di controllo ad alto livello per gli UAV.

Inoltre spero che lo studio dell'architettura del software, utilizzato nel modulo di basso livello, possa essere un "mattoncino" importante per tutti coloro che intendono approcciarsi a questo tipo di tecnologia.

Ringraziamenti

Essere arrivato a concludere questa laurea triennale per me vuol dire tanto. Se tornassi indietro rifarei la stessa scelta poichè ho imparato davvero tante cose interessanti e coinvolgenti.

Il primo grande ringraziamento va ad Alice che mi ha supportato/sopportato in ogni istante. Mi è stata sempre vicino anche nei momenti più difficili, facendomi sentire sempre tanto/troppo importante. Questi anni di convivenza insieme a lei, partiti proprio grazie al percorso universitario appena concluso, mi hanno insegnato tante cose facendomi crescere ed ampliare i miei orizzonti.

Non posso non ringraziare anche i miei genitori che hanno sempre creduto in me e ce l'hanno messa tutta per farmi realizzare i miei sogni.

Un grande ringraziamento va anche a tutti i miei amici, vicini e lontani, che mi hanno tenuto compagnia in questi tre anni e mezzo tramite chiacchierate, birrette, jam session, etc. Ringrazio anche gli amici con cui ho studiato e portato avanti progetti d'esame.

Grazie anche a Roberto per avermi dato l'opportunità di lavorare in questo progetto a contatto con gente competente e motivata come i ragazzi del CASY di Bologna.

Un ringraziamento speciale va a Nicola che mi ha seguito passo-passo nel mio progetto di tesi, essendo sempre presente per chiarimenti, spiegazioni e confronti.

Un grazie di cuore a tutti!

Bibliografia

[1] SHERPA. *Sherpa Project*, http://www.sherpa-project.eu/sherpa/.

[2] Wikipedia. *UAV - wikipedia, the free encyclpedia*, https://en.wikipedia.org/wiki/Unmanned_aerial_vehicle.

[3] Michele Persiani. *Sistema di controllo WEB per droni UAV*, Marzo 2013.

[4] APM Multiplatform Autopilot. *Mission Planner Overview*, http://planner.ardupilot.com/wiki/mission-planner-overview/.

[5] PX4 Autopilot. *Pixhawk Autopilot*, https://pixhawk.org/modules/pixhawk.

[6] NuttX. *NuttX Real-Time Operating System*, http://www.nuttx.org.

[7] APM Multiplatform Autopilot. *Developer Ardupilot APM*, http://dev.ardupilot.com.

[8] Hardkernel. *Odroid - Products*, http://www.hardkernel.com/main/products/prdt_info.php?g_code=G138745696275.

[9] Lubuntu. *Lubuntu - lightweight, fast, easier*, http://lubuntu.net.

[10] ROS. *ROS - What is ROS?*, http://www.ros.org.

[11] Simon Puligny. *ROS interface and URDF parser for Webots*, 14 February 2014.

[12] 3DRobotics. *Products - IRIS+*, https://store.3drobotics.com/products/iris.

[13] Qgroundcontrol, *MAVLINK Overview*, http://qgroundcontrol.org/mavlink/start.

[14] ROS. *MAVROS - package documentation*, http://wiki.ros.org/mavros.

[15] Ente Nazionale per l'Aviazione Civile. ENAC, `http://www.enac.gov.it/Home/`.

[16] APM Multiplatform Autopilot. *Arducopter*, `http://copter.ardupilot.com`.