eYSIP 2017

# Control and Firmware Development for Quadcopters

Pluto Drone by Drona Aviation

Heethesh Vhavle
Sanam Shakya
Pushkar Raj

Duration of Internship: 22/05/2017 – 07/07/2017

*2017, e-Yantra Publication*

# Contents

# List of Figures

# Introduction

## 1.1 Abstract

The project deals with the study of control algorithms and to develop a custom firmware for quadcopter (flight controller) on 32-bit micro-controllers such as the STM32F1xx (ARM Cortex-M3 core). The flight controller is designed to control parameters such as the throttle, yaw, pitch and roll and consists of algorithms considering various motion and dynamics. The next step is to analyze the control algorithm to identify effects of various parameters and to optimize it for stable motion. The final step is to develop a wireless joystick controller for simple maneuvering of the quadcopter.

## 1.2 Completion Status

1. Flight controller firmware development is complete. Various libraries such as UART, I2C, Timers, PWM, various sensors, MSP, motors, joystick and PID control libraries were developed for the firmware.
2. Sensor interfacing is complete. The MPU9250 IMU (Accelerometer and Gyroscope) and AK8963 Magnetometer were successfully interfaced, calibrated and the filtered pitch, roll and yaw angles were obtained. The MS5611 Barometer was successfully interfaced as well and an estimate of altitude, based on the pressure readings, was obtained.
3. The MultiWii Serial Protocol (MSP) was implemented for communication with the flight controller and for telemetry purposes. Real time data can be visualized using Python or MultiWii Conf.
4. An USB joystick controller was interfaced with the PC and using Python, wireless control of the quadcopter was developed.

5. A control architecture was designed for pitch, roll, yaw and throttle control, with a PID controller implemented for each axis. On-the-fly PID tuning was implemented along with a GUI developed for the same. The PID gains for the pitch, roll and yaw axes were tuned. However, fine tuning of these gains are not yet complete.

6. Quadcopter flight was achieved and parameters such as pitch, roll, yaw, throttle and trim were successfully controlled using the joystick controller. However, the flight is not stable yet and the quadcopter experiences drift in motion. Altitude hold feature is yet to be implemented.

Figure 1.1: Pluto Drone by Drona Aviation

# Hardware

One of the goals of the project is to develop a firmware for 32-bit micro-controllers. The micro-controller used in this project is the STM32F103 (ARM Cortex-M3 Core). Initially the firmware development was carried out on the Nucleo Development Board. The quadcopter, used in the later stage, is the Pluto Drone, provided by Drona Aviation. The following section lists all the hardware components used in this project along with its specifications.

## 2.1 List of Hardware Components

- NUCLEO-F103RB STM32 Nucleo Development Board
  Datasheet — Chip Datasheet — Vendor Link
- Drona Aviation - Pluto Drone
  Schematic — Vendor Link
  - STM32F103C8 Micro-controller
    Datasheet — Reference Manual — Guide Book — Vendor Link
  - MPU9250 Accelerometer and Gyroscope
    Datasheet — Register Map — Vendor Link
  - AK8963 Magnetometer
    Datasheet — Vendor Link
  - MS5611 Barometer
    Datasheet — Vendor Link
  - ESP8266 Wi-Fi Wireless Module
    Datasheet — Vendor Link

- Coreless Motors and Propellers
  Vendor Link
- 3.7V Li-Po Battery
  Vendor Link

## 2.2 STM32 Micro-controller

The STM32 family of micro-controllers, based upon the ARM Cortex-M3 core, provides a foundation for building a vast range of embedded systems from simple battery powered dongles to complex real-time systems such as helicopter autopilots. This component family provides wide-ranging choices in memory sizes, available peripherals, performance, and power. Unfortunately, power and flexibility are achieved at a cost  software development for the STM32 family can be extremely challenging for the uninitiated with a vast array of documentation and software libraries to wade through.[1]

The STM32F103xx medium-density performance line family incorporates the high-performance ARM Cortex-M3 32-bit RISC core operating at a 72 MHz frequency, high-speed embedded memories (Flash memory up to 128 Kbytes and SRAM up to 20 Kbytes), and an extensive range of enhanced I/Os and peripherals connected to two APB buses. All devices offer two 12-bit ADCs, three general purpose 16-bit timers plus one PWM timer, as well as standard and advanced communication interfaces: up to two I2Cs and SPIs, three USARTs, an USB and a CAN.[2]

## 2.3 Hardware Schematic

The Pluto Drone embedded board consists of the STM32F103C8 micro-controller. MPU9250 IMU and MS5611 Barometer are interfaced on I2C1 bus. The AK8963 Magnetometer is interfaced as a slave device to the MPU9250 master (refer to Section 3.1.1 for further details). Although the board supports upto six motor connections, only four are used (M1-M4), which are connected to the PWM channel pins. VL53L0X is a LASER ranging sensor, which is currently not interfaced. There are three on-board indicator LEDs (D1, D2 and D3). *Figure 2.2* shows the hardware schematic.

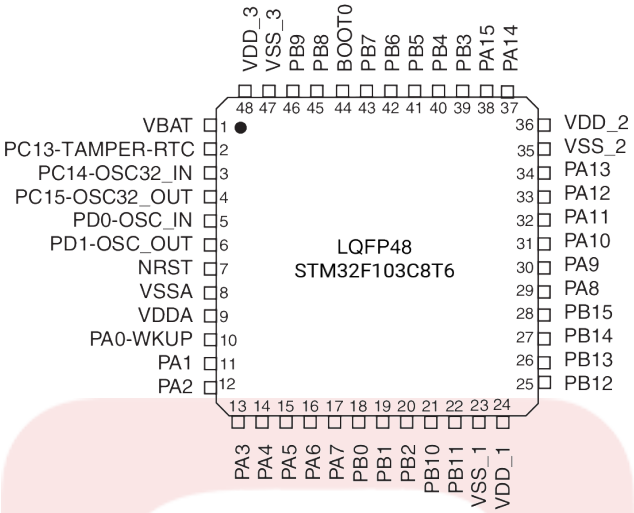


Figure 2.1: Pinout diagram of STM32F103C8T6 LQFP48 package

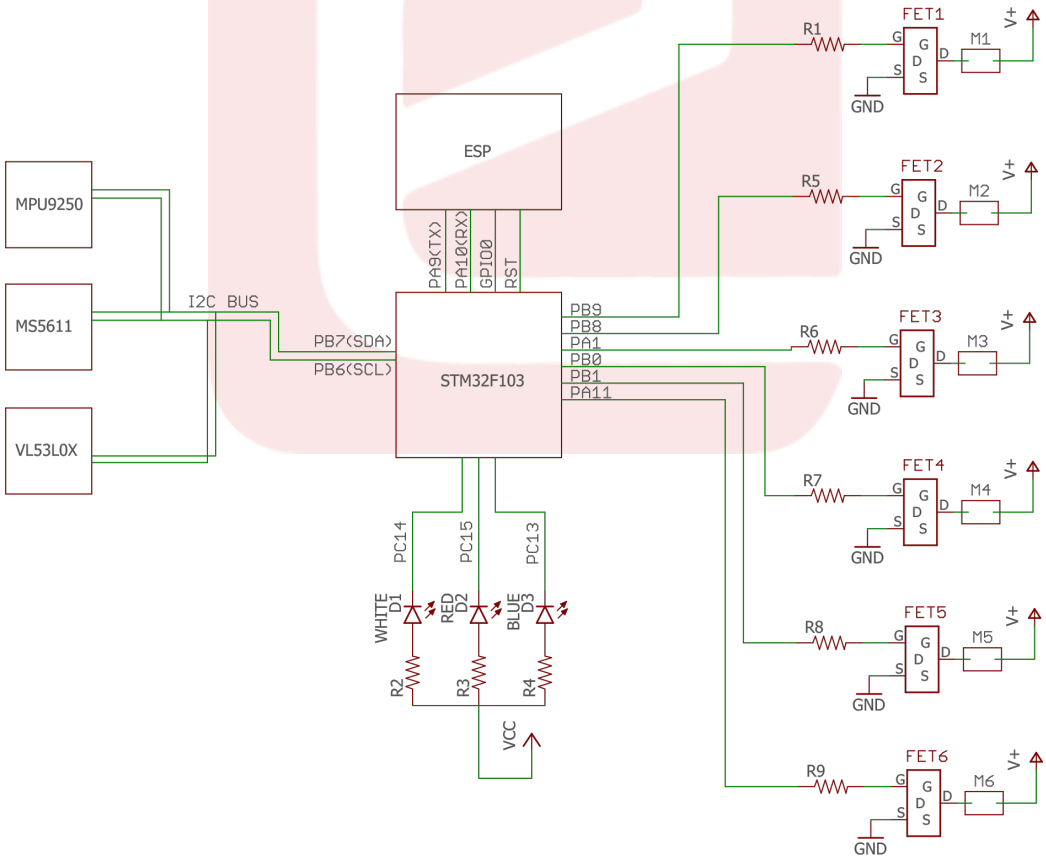


Figure 2.2: Hardware schematic of Pluto Drone

# Software

The primary software resource is the GNU software development tool-chain including gcc, objcopy, objdump, and the debugger gdb. The GNU ARM Embedded tool-chains are integrated and validated packages featuring the ARM Embedded GCC compiler, libraries and other GNU tools necessary for bare-metal software development on devices based on the ARM Cortex-M and Cortex-R processors.[1, 3] The entire firmware is written in C language. Coming to IDE, various different options are available such as ARM Keil MDK, EWARM, TrueSTUDIO and open source options such as the Eclipse IDE.

## 3.1 Installation of Softwares and Tools

This section gives a brief overview on setting up the IDE and the various methods of flashing the program on to the micro-controller.

### 3.1.1 Integrated Development Environment (IDE)

The Eclipse IDE requires the tool-chain, debug and build tools to be installed manually. The tool-chains are available for cross-compilation on Microsoft Windows, Linux and Mac OS X host operating systems. The tool-chain can be downloaded from here.[3]

The IDE used is in this project is Attolic TrueSTUDIO, which is based on the Eclipse IDE. One of the main advantages is that TrueSTUDIO IDE already comes with the GCC tool-chains for ARM and the required debugging tools.

### 3.1.2 Debugging Tools

One of the key feature of Nucleo board is that it already provides the on-chip programmer for STM32 micro-controllers. In order to use this, ST-LINK drivers need to be installed.

Although TrueSTUDIO has built-in debugging tools, some of the features are not available in the free version of the software. Instead, we can use the official ST-LINK Utility tool.

### 3.1.3 Flashing the Program

There are three ways to upload the firmware on to the micro-controller. One of the easiest way is to simply generate a *binary (*.bin)* file, connect your board via USB and it should be detected as an external storage device. Program can be flashed by just dragging and dropping the binary file on the device.

The ST-LINK Utility tool also supports flashing the program on to the board. One can alternatively use the ST Link V2 Programmer. The header for this programmer contains connections for 5V, 3.3V, SWCLK, SWDIO, SWIM, Reset (RST/NRST) and GND. The connector on the opposite side of this device is a USB connector and is intended to be plugged into the computer for programming.

Another method of flashing the program is by using the built-in UART Bootloader. Further details are explained here. An USART to USB driver chip like FT232 can be used along with STM32 Flash Loader Demonstrator to flash the program. The *Pluto Drone* board supports wireless programming (UART Bootloading) with the help of the ESP8266 which is connected to the boot, reset and USART pins on the board. CleanFlight Configurator software can be used to flash the program (**.hex*) over Wi-Fi.

## 3.2 Programming the STM32 Micro-controller

Programming the STM32 series micro-controllers is done in various layers namely, CMSIS, HAL, BSP and Application code. The software stack is shown in *Figure 3.1*.

The *Cortex Micro-controller Software Interface Standard (CMSIS)* supports developers in creating reusable software components for ARM Cortex-M based systems. It mostly contains definitions for the various registers.

The *Hardware Abstraction Layer (HAL)* drivers were designed to offer a rich set of APIs and to interact easily with the application upper layers. Each driver consists of a set of functions covering the most common peripheral features such as GPIO, UART, I2C, CAN and so on.[4]

When a STM32 micro-controller starts, it needs an hardware configuration to work correctly. Unfortunately, the development of one's own HAL requires a deep knowledge of the specific micro-controller. ST provides a dedicated tool that generates the initialization code and provides us the HAL peripheral drivers. This tool is called STM32CubeMX and it is very useful.[5]

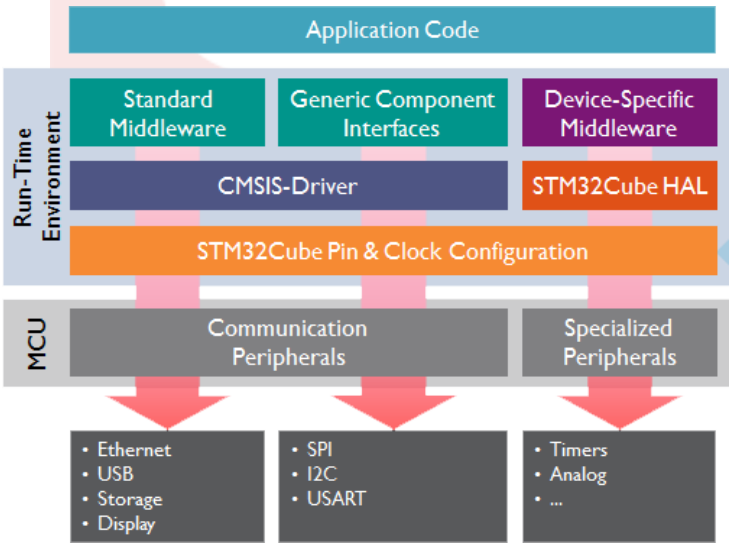


Figure 3.1: Software stack for STM32 programming

# Flight Controller Firmware

The motivation behind developing a custom flight controller firmware from scratch is that one can optimize the flight performance up to middleware layer. It may seem like a lot of effort to be put in to develop such a firmware from scratch initially, but once it is developed it is more advantageous to the user than using an already existing flight controller. Some advantages are that it allows for great customization and flexibility in terms of the control algorithm and in optimizing the flight dynamics, implementing our own communication and telemetry protocols for specific usage and allows for more advanced pilot control and features to be implemented easily.

Although open-source flight controllers like *CleanFlight* and *dRonin* exist, it is limited in terms of the customization aspect as these are designed to support a vast array of hardwares. Making a simple modification to a certain feature requires one to wade through the vast amount of libraries and requires good knowledge about the firmware implementation.

## 4.1 Structural Overview

A brief overview of the software stack was given in the previous chapter. The two most common peripheral drivers available for STM32 micro-controllers are *Standard Peripheral Library* and *STM32Cube HAL*. The later is used in the firmware as it is a more standardized method and is less error-prone as the driver initialization code is auto generated.

Firmware development in this project began from the *Board Specific Package (BSP)* layer with development of libraries such as GPIO, I2C, Serial, Timing

and PWM. Once these libraries were ready, the application layer development began with libraries for the different sensor, filters, control algorithms, telemetry protocols (MSP) and joystick interfacing. *Figure 4.1* shows the basic directory structure of the firmware. *Note: Only header files are shown.*

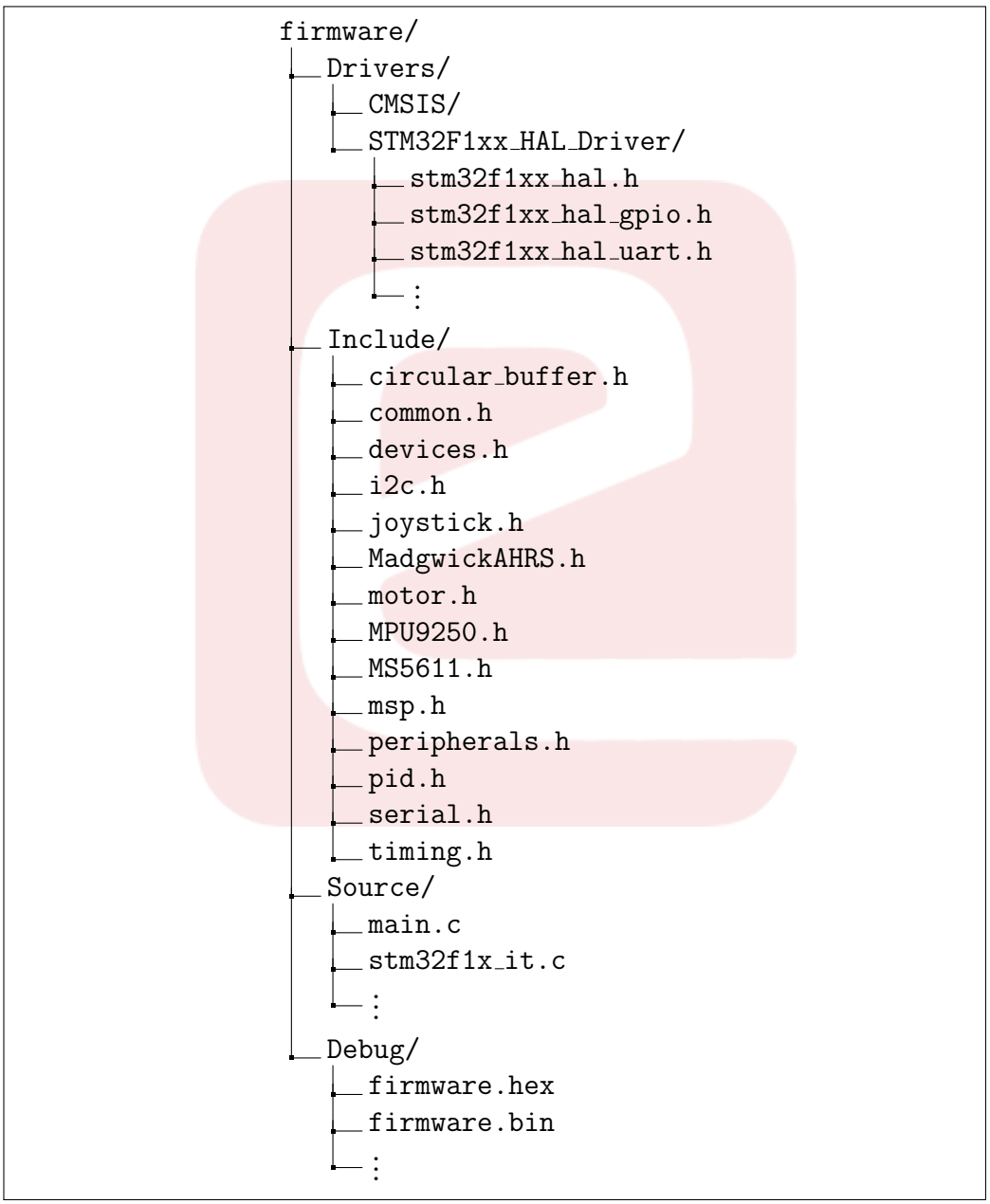


Figure 4.1: Directory structure of the flight controller firmware

## 4.2 Board Specific Package (BSP)

The BSP layer bridges the gap between the device drivers and the application layer. While the HAL only provides the peripheral initialization, the BSP implements the functionality of the peripherals which is used by the application layer. In the current firmware, the following libraries are a part of the BSP.

1. **peripherals.h** — This library consists of all the HAL peripheral initialization code and provides a master *Devices_Init()* function. It also handles the system clock configuration and error handling function.

2. **devices.h** — This library handles all of the I/O interfacing such as LED control and PWM write functions.

3. **timing.h** — The timing library provides the *delay_ms()* and *millis()* functions, built using *SysTick*.

4. **serial.h** — The serial library consists of serial read and write functions for a single byte from the circular buffer. The UART interrupt callback functions of the HAL are implemented in this library. It also provides functions for printing strings, floats and integers on a terminal.

5. **circular_buffer.h** — A circular buffer is required to store the received data when using interrupts for UART communication. Further details of UART communication is explained in [6].

6. **i2c.h** — This library provides function to write a single byte and read individual as well as multiple bytes over I2C protocol.

7. **common.h** — This library consists of all the miscellaneous functions such as *map()*, *map_float()*, *constrain()* and *lowPassFilter()* and is used by higher level libraries.

## 4.3 Attitude and Heading Reference System

One of the most trivial and important tasks of a flight controller firmware is control and stabilization of the orientation of the quadcopter. The three axes of orientation involved are namely pitch, roll and yaw as shown in *Figure 4.2*

An *Attitude and Heading Reference System (AHRS)* consists of sensors on three axes that provide attitude information for aircraft, including roll, pitch and yaw. They are designed to replace traditional mechanical gyroscopic flight instruments and provide superior reliability and accuracy.[7]

AHRS consist of either solid-state or microelectromechanical systems (MEMS) gyroscopes, accelerometers and magnetometers on all three axes. A form of non-linear estimation such as an Extended Kalman filter is typically used to compute the solution from these multiple sources. One abbreviation used in technology for sensor arrays used in AHRS is MARG (Magnetic, Angular Rate, and Gravity).[7]

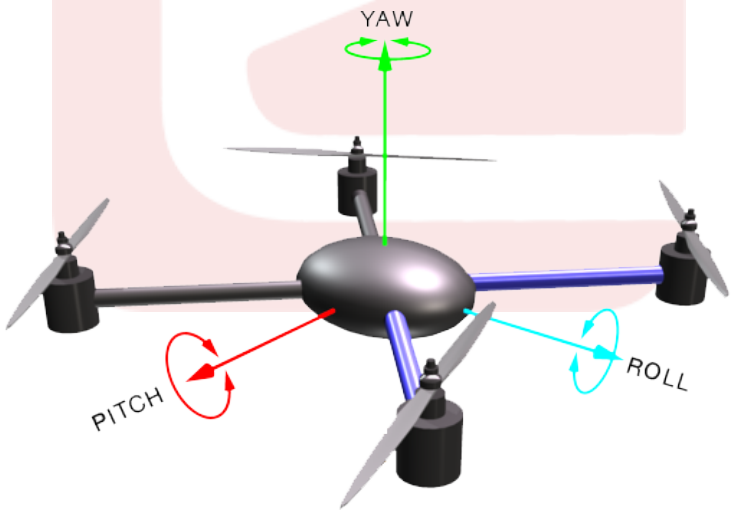


Figure 4.2: The pitch, roll and yaw angles (AHRS) of a quadcopter[8]

### 4.3.1 MPU9250 Accelerometer and Gyroscope

The MPU-9250 is an inertial measurement unit (IMU). MPU-9250 features three 16-bit analog-to-digital converters (ADCs) for digitizing the gyroscope outputs, three 16-bit ADCs for digitizing the accelerometer outputs, and three 16-bit ADCs for digitizing the magnetometer outputs. For precision tracking of both fast and slow motions, the parts feature a user-programmable gyroscope full-scale range of ±250, ±500, ±1000, and ±42 $^\circ\,s^{-1}$ (dps), a user-programmable accelerometer full-scale range of ±2g, ±4g, ±8g, and ±16g, and a magnetometer full-scale range of ±4800$\mu$T. The MPU-9250 can be interfaced via SPI or I2C. In our case, I2C at 400 kHz is used to interface the sensor.

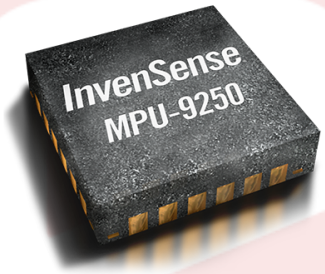


Figure 4.3: InvenSense MPU-9250 QFN package[9]

#### 4.3.1.1 Accelerometer

Accelerometers are devices that measure static acceleration in all the three axes. They measure in G-forces (g) equivalent to 9.8 kg m $s^{-2}$. Most accelerometers will have a selectable range of forces they can measure. These ranges can vary from ±1g up to ±250g. Typically, the smaller the range, the more sensitive the readings will be from the accelerometer. In our case, the selected range is ±4g.[10]

Assuming no external forces act (except gravity) on the sensor, the pitch and roll angles can easily be computed by knowing the components of gravity in the different axis, which is the sensor output. By using trigonometric formulas, we can easily compute these angles. Practically, the accelerometer not only measures gravity but also all the other vibrations and the output is very noisy. This noise can be filtered out with the help of a gyroscope.

#### 4.3.1.2 Gyroscope

The gyroscope sensor within the MEMS is tiny (between 1 to 100 micrometers). When the gyro is rotated, a small resonating mass is shifted as the angular velocity changes. This movement is converted into very low-current electrical signals that can be amplified and read by a host micro-controller [11]. The range of the gyroscope selected is ±1000 dps.

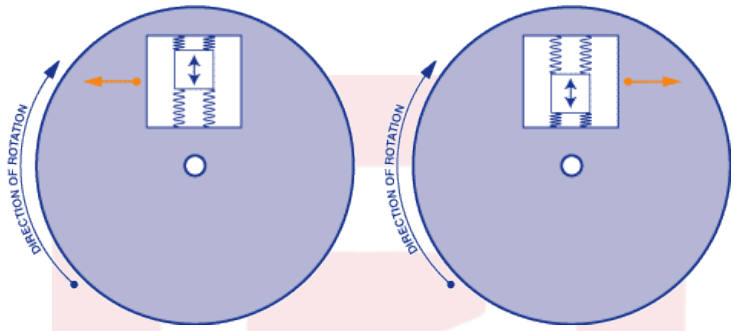


Figure 4.4: Internal operational view of a MEMS gyro sensor[11]

However, the gyroscope output doesn't give the angular orientation. It gives the angular rate of orientation around the three axes. This rate has to be integrated over time in order to obtain the orientation angles. But the main disadvantage of the gyroscope is that its value drifts over time due to rounding off errors and keeps accumulating these errors as we integrate it.

One of the solutions to combat this problem is to fuse the sensor data from both the accelerometer and the gyroscope. A simple filter like a complimentary filter can be used to achieve this. The result of this filter is pretty good. The accelerometer provides gravity as an absolute reference and helps to correct the drift from of the gyroscope.

However, this solution only provides us with only the pitch and roll angles. We also require to know the heading of the quadcopter or the yaw angle. Although, the gyroscope can measure the yaw rate of rotation, the accelerometer has no means to detect a change in yaw direction, as it only refers to gravity. Without a reference in the yaw direction, the gyroscope output will again drift. Now, that's where the Magnetometer comes in to provide this absolute reference.

#### 4.3.1.3 Sensor Initialization

Following is a snippet explaining the initialization of the IMU. It is always a safe practice to verify the device ID of the sensor, so that we know the I2C interface is successful and if not, handle the error. The register names are macros defined in the *MPU9250.h* header file. Address values and register bits are explained in detail in the MPU-9250 Register Map.

```
// Verify device by comparing the WHO_AM_I register value
uint8_t data = I2C_ReadByte(MPU9250_ADDRESS, WHO_AM_I, __FILE__, __LINE__);

// Device verification failed
if (data != WHO_AM_I_VALUE) _Error_Handler(__FILE__, __LINE__);
```

The next step is to first reset the device, enable the sensors and configure the internal clock source. Settings such as digital low pass filtering and full scale range can be configured as shown below.

```
// Reset
I2C_WriteByte(MPU9250_ADDRESS, MPU_PWR_MGMT_1, 0x80, 1);

// Set clock source to be PLL
I2C_WriteByte(MPU9250_ADDRESS, MPU_PWR_MGMT_1, 0x01, 1);

// Enable Accel and Gyro
I2C_WriteByte(MPU9250_ADDRESS, MPU_PWR_MGMT_2, 0x00, 1);

// Sample Rate Divider (Not set)
I2C_WriteByte(MPU9250_ADDRESS, SMPLRT_DIV, 0x00, 1);

// DLPF 184Hz
I2C_WriteByte(MPU9250_ADDRESS, ACCEL_CONFIG2, 0x03, 1);
I2C_WriteByte(MPU9250_ADDRESS, MPU9250_CONFIG, 0x03, 1);

// Full scale settings
I2C_WriteByte(MPU9250_ADDRESS, GYRO_CONFIG, GYRO_FS_1000_DPS, 1);
I2C_WriteByte(MPU9250_ADDRESS, ACCEL_CONFIG, ACC_FS_4_G, 1);
```

#### 4.3.1.4 Reading the Sensor Data

The initialization function will have to be called once and after that the raw data from the sensors can be read, converted and compensated at fixed refresh rate. The chosen refresh rate must be less than the update rate of the sensor. MPU-9250 has an update rate of 1 kHz. The raw data is available in the output registers in series, starting with the X axis. The output of each axis is a 16-bit value and is available as set of two bytes (high and low). Therefore, six bytes of data will have to be read in series (two for each axis) and they must be packed back to a 16-bit value.

```
// Data buffer to store the raw data
uint8_t raw_data[] = {0, 0, 0, 0, 0, 0};

// Read raw data
I2C_ReadByteArray(MPU9250_ADDRESS, ACCEL_XOUT_H, raw_data, 6);

// Pack it to 16-bits and store it
accelRaw.x = (int16_t) ((raw_data[0]<<8) | raw_data[1]);
accelRaw.y = (int16_t) ((raw_data[2]<<8) | raw_data[3]);
accelRaw.z = (int16_t) ((raw_data[4]<<8) | raw_data[5]);
```

The raw data is just a numerical value ranging from -32768 to +32767. This data will have to be converted to G-units. The conversion depends on the full scale range settings chosen for the sensor. Here the setting chosen was ±4 and the conversion is performed as follows.

```
accelData.x = (float) accelRaw.x * 4.0f/32768.0f;
accelData.y = (float) accelRaw.y * 4.0f/32768.0f;
accelData.z = (float) accelRaw.z * 4.0f/32768.0f;
```

Similarly, the gyroscope data can be obtained, the only difference being the conversion factor. When the sensor is kept in static position, the output of the gyroscope will still be non-zero. This gyroscope bias can calculated by taking multiple samples when kept still, and then averaging it.

```
gyroData.x = ((float) gyroRaw.x * 1000.0f/32768.0f) - gyroBias.x;
gyroData.y = ((float) gyroRaw.y * 1000.0f/32768.0f) - gyroBias.y;
gyroData.z = ((float) gyroRaw.z * 1000.0f/32768.0f) - gyroBias.z;
```

### 4.3.2 AK8963 Magnetometer

The AK8963 Magnetometer is 3-axis electronic compass IC with high sensitive Hall sensor technology. This sensor is interfaced as an auxiliary sensor to the MPU-9250. The MPU-9250 acts as a master to any external sensors connected to the auxiliary I2C bus. To access the magnetometer via MPU-9250, the *BYPAS_EN* bit in the *INT Pin / Bypass Enable Configuration* register of the MPU-9250 has to be set. Further details about this can be found in Page 23 - Datasheet. The initialization snippet is shown below.

```
/* Enable access to Magnetometer via MPU-9250 */
// Set bypass mode for external I2C master connection
I2C_WriteByte(MPU9250_ADDRESS, INT_PIN_CFG, 0x22, 1);

// Verify magnetometer device ID
uint8_t data = I2C_ReadByte(MAG_ADDRESS, MAG_WIA);
if (data != MAG_WIA_VALUE) _Error_Handler(__FILE__, __LINE__);

// Reset magnetometer
I2C_WriteByte(MAG_ADDRESS, MAG_CNTL2, 0x01, 1);
```

#### 4.3.2.1 Magnetometer Biases

Magnetometers are wonderful devices and absolutely essential for correcting gyro drift in applications that require absolute orientation through sensor fusion. The bane of magnetometer usage is, however, their non-ideal response surfaces. The ideal response surface for a three-axis magnetometer is a sphere centered at the 3D origin. This means the response to an external magnetic field of, let's say 400 milliGauss (mG) in the z-direction would be exactly Mz = 400 mG when the magnetometers z-axis was normal to the floor, My = 400 mG when the magnetometer's y-axis was normal to the floor, and Mx = 400 mG when the magnetometer's x-axis was normal to the floor. More simply, the ideal response surface no matter the orientation of the magnetometer is a sphere with radius 400 mG centered on the origin. In practice, MEMS magnetometers are rarely so well calibrated when you receive them. There are good reasons for this. The MEMS sensors are typically characterized at the factory but mounting on a PC board can add stresses that can easily result in a shift of the calibration.[12]

#### 4.3.2.2 Sensor Initialization

The first set of calibration process is to compensate the raw data with the help of factory calibrated data. The factory sensitivity adjustment data for each axis is stored in fuse ROM on shipment. These scaling factors can be read from the ROM as follows.

```
// Power down magnetometer
I2C_WriteByte(MAG_ADDRESS, MAG_CNTL1, 0x00, 1);

// Enter Fuse ROM access mode
I2C_WriteByte(MAG_ADDRESS, MAG_CNTL1, 0x0F, 1);

// Read calibration registers for sensitivity adjustment values
uint8_t rawData[3];
I2C_ReadByteArray(MAG_ADDRESS, MAG_ASAX, rawData, 3);

// Calculate and store the adjusted measurement data
magCalib.x =  (float)(rawData[0] - 128)/256.0f + 1.0f;
magCalib.y =  (float)(rawData[1] - 128)/256.0f + 1.0f;
magCalib.z =  (float)(rawData[2] - 128)/256.0f + 1.0f;
```

AK8963 has seven operation modes. Continuous measurement mode 2 is used in our case. Sensor is measured periodically in 100Hz in this mode and the configuration snippet is shown below.

```
// Power down magnetometer
I2C_WriteByte(MAG_ADDRESS, MAG_CNTL1, 0x00, 1);

// Res: 16 Bit, Mode: Continuous Mode 2 (100Hz)
I2C_WriteByte(MAG_ADDRESS, MAG_CNTL1, 0x16, 1);
```

In spite of the factory calibration, the problem of non-ideal response of the sensor in various axes, which was mentioned earlier, still persists. The data plotted in *Figure 4.5* was taken by slowly turning the sensor board through a variety of figure-eight patterns. The data is properly-scaled (mG) Mx, My, and Mz values plotted using *Matplotlib* in Python. Following are the observations that can be inferred from the uncalibrated sensor data.

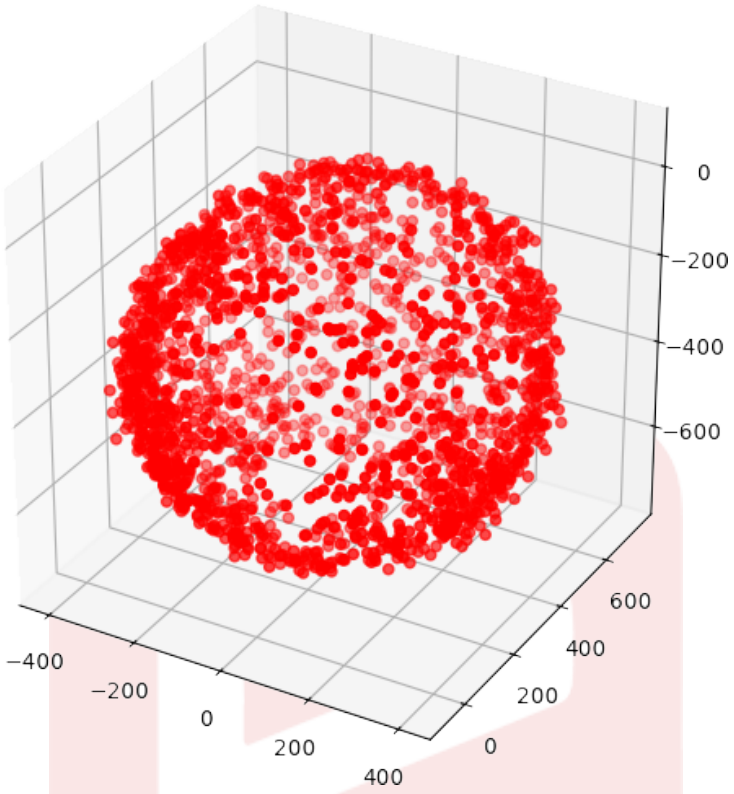


Figure 4.5: Magnetic field in mG along X, Y and Z axes measured for an uncalibrated sensor

1. The response between axes is not centered at the origin.
2. The response sensitivity is different along each axis.

These are often referred to as hard iron and soft iron biases, respectively. Calculating these biases is explained in *Section 4.3.2.4*.

**Hard iron biases** are typically the largest and the easiest errors to correct for. The simplest way to correct for them is to record a bunch of magnetometer data as the sensor is moved slowly in a figure eight pattern and keep track of the minimum and maximum field measured in each of the six principal directions; +/- Mx, +/- My, +/- Mz. Once the min/max values along the three axes are known, the average can be subtracted from the subsequent data which amounts to re-centering the response surface on the origin [12]. *Figure 4.6* shows the hard iron bias calibrated sensor data.

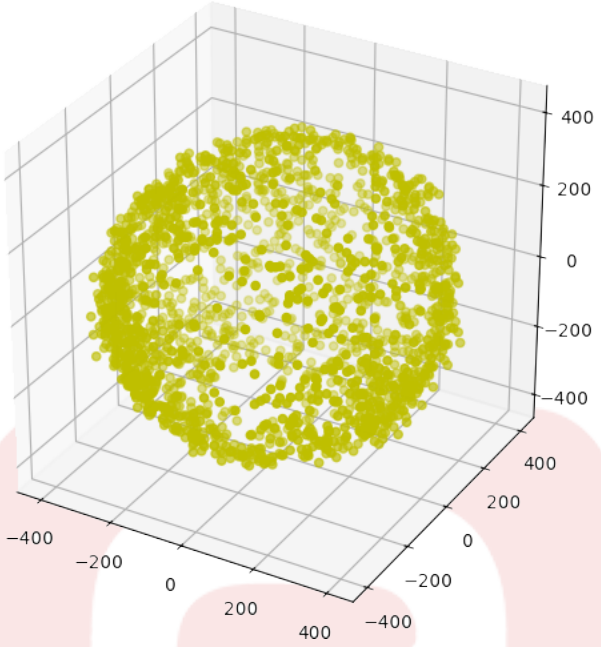


Figure 4.6: The AK8963 magnetometer data after subtraction of the three axial offset biases

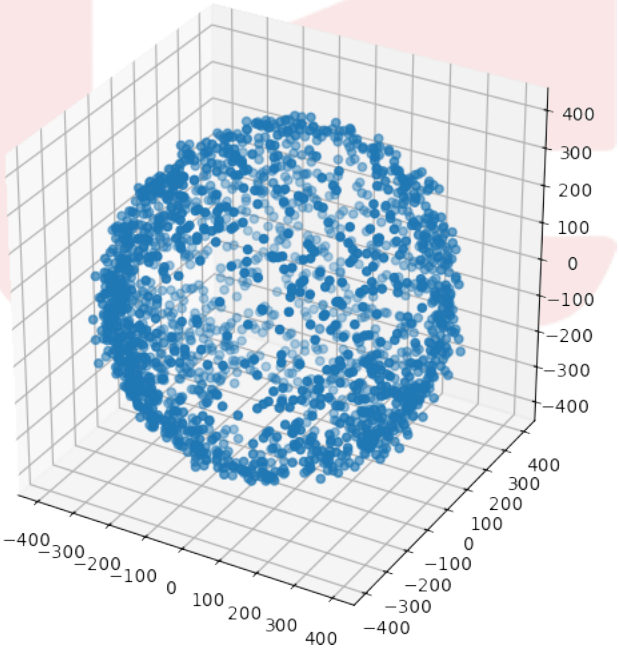


Figure 4.7: The MPU9250 magnetometer response after offset and scaling bias corrections

**Soft iron bias** correction is re-scaling the axial response to make is even more spherical. It means deconstructing the response surface into its elliptical principle axes and devising a 3 x 3 correction matrix to transform the general ellipsoidal response surface into a spherical one. The min/max values which are already computed are used to rescale the magnetometer data and compute a scale factor for all three axes. Finally, the calibrated sensor data is shown in *Figure 4.7*.[12]

#### 4.3.2.3 Reading the Sensor Data

Once all of the initialization and calibration is performed, the data can be read from the sensor now. There are certain points to note before we can read the data. When measurement data is stored and ready to be read, DRDY (Data Ready) bit in ST1 register turns to *1* and this condition has to be checked first *(line 2)*.

AK8963 has the limitation for measurement range that the sum of absolute values of each axis should be smaller than 4912$\mu$T.

$$| X | + | Y | + | Z | < 4912\mu T$$

When the magnetic field exceeds this limitation, data stored at measurement data is not correct. This is called Magnetic Sensor Overflow. When magnetic sensor overflow occurs, HOFL bit in ST2 register turns to *1*. When the next measurement starts, it returns to *0 (line 9)*. Also it is necessary that a read operation on the ST2 register has to be performed in order to initiate the next conversion cycle *(line 5)*.

Another thing to note is, unlike the MPU-9250, the raw data here is available as low byte first, then the high byte. Once the raw data has packed into 16-bit the factory calibration is applied. Now, the raw data has to be converted to milliGauss and *mRes* is the conversion factor.

$$mRes = 10.0 \times 4912.0/32760.0$$

The final step is to compensate the converted data with the hard and soft iron biases. Calculating these biases is explained in *Section 4.3.2.4*.

```
// Check if data is ready
if (I2C_ReadByte(MAG_ADDRESS, MAG_ST1, __FILE__, __LINE__) & 0x01)
{
  // Read data registers and ST2 register to check overflow
  I2C_ReadByteArray(MAG_ADDRESS, MAG_HXL, raw_data, 7, __FILE__, __LINE__);
  uint8_t OVF = raw_data[6];

  // Store data if no overflow occurred
  if (!(OVF & 0x08))
  {
    // Pack into 16-bit data
    magRaw.x = (int16_t) ((raw_data[1]<<8) | raw_data[0]);
    magRaw.y = (int16_t) ((raw_data[3]<<8) | raw_data[2]);
    magRaw.z = (int16_t) ((raw_data[5]<<8) | raw_data[4]);

    // Apply the factory calibration and conversion factors
    magData.x = (float) magRaw.x * mRes * magCalib.x;
    magData.y = (float) magRaw.y * mRes * magCalib.y;
    magData.z = (float) magRaw.z * mRes * magCalib.z;

    // Apply the hard iron and soft iron bias corrections
    magData.x = (magData.x - magBias.x) * magScale.x;
    magData.y = (magData.y - magBias.y) * magScale.y;
    magData.z = (magData.z - magBias.z) * magScale.z;
}
```

#### 4.3.2.4 Magnetometer Calibration

Currently, the hard and soft iron bias calibration is not implemented in the firmware and has to performed externally. To do this, the converted magnetometer readings are just printed on the serial terminal and then stored as **.csv* file. The calibration is done using a Python script. The **.csv* file of uncalibrated readings are read and the following steps are carried out in order to get the bias values.

1. Find out the maximum and the minimum values for each of the axes from the data samples.

2. Find the average of the maximum and the minimum values.

3. This average value is nothing but the hard iron bias value.

4. To find out the soft iron bias, calculate the range of the data samples. This is just the difference between the maximum and minimum value.

5. Find out the average of the range of all the three axes.

6. The average value divided by the range of each axis gives the scaling factor for that respective axis. This is the soft iron bias.

Following is a snippet from the *mag-calib.py* Python script, implementating the calibration process.

```
# max_ and min_ are lists which store the maximum and minimum
# values for the three axes, from the data samples
for i in range(0,3):
    # Find the hard iron bias
    bias[i] = (max_[i] + min_[i])/2

    # Find the range to calculate soft iron bias
    range_[i] = max_[i] - min_[i]

# Average of the range of three axes
avg = (range_[0] + range_[1] + range_[2])/3

# Calculate the scaling factor or the soft iron bias for each axis
for i in range(0,3):
    scale[i] = avg/range_[i]
```

There is an additional common type of magnetometer bias often encountered which is due to the presence of man-made sources of magnetic field like big steel buildings and current carrying wires, etc. Encounters with these environmental sources of magnetic field can be interpreted as changes in orientation if a magnetic anomaly algorithm is not employed to detect and prevent it. This is beyond the scope of the project right now and can be dealt with in future developement of the firmware. [12]

### 4.3.3 Sensor Fusion using Madgwick's AHRS Filter

A MARG (Magnetic, Angular Rate, and Gravity) sensor is a hybrid IMU which incorporates a tri-axis magnetometer. An IMU alone can only measure an attitude (pitch and roll angles) relative to the direction of gravity which is sufficient for many applications. MARG systems, also known as AHRS (Attitude and Heading Reference Systems) are able to provide a complete measurement of orientation relative to the direction of gravity and the earth's magnetic field.[13]

The task of an orientation filter is to compute a single estimate of orientation through the optimal fusion of gyroscope, accelerometer and magnetometer measurements. The Kalman filter has become the accepted basis for the majority of orientation filter algorithms. However, implementation of these filters demand a large computational load and not very efficient for embedded systems.[13]

Madgwick's algorithm uses a quaternion representation, allowing accelerometer and magnetometer data to be used in an analytically derived and optimised gradient descent algorithm to compute the direction of the gyroscope measurement error as a quaternion derivative. The library for Madgwick's AHRS filter is available here and further details of the working of the filter can be found here.[13]

Following is a snippet which shows how to use the AHRS filter. Notice the change in sign and axis while passing the IMU data to the filter. *AHRS_Angle* is a float array to store the filtered angles.

```
1  // Set integration time by time elapsed since last filter update
2  AHRS_timeNow = millis();
3  float delta = (float)((AHRS_timeNow - AHRS_lastUpdate)/1000.0f) ;
4  MadgwickSetDelta(delta);
5  AHRS_lastUpdate = AHRS_timeNow;
6
7  // Filter data and obtain the angles
8  MadgwickQuaternionUpdate(-accelData.y, -accelData.x, accelData.z,
9          gyroData.y, gyroData.x, -gyroData.z, magData.x, magData.y,
10         magData.z, AHRS_Angle);
```

## 4.4 Altitude Measurement

Altitude can be determined based on the measurement of atmospheric pressure. The greater the altitude, the lower the pressure. When a barometer is supplied with a nonlinear calibration so as to indicate altitude, the instrument is called a pressure altimeter or barometric altimeter. It is very important to know the altitude of a flight and altitude stabilization is required for better control of the quadcopter.

### 4.4.1 MS5611 Barometer

The MS5611 is a barometric pressure sensor is optimized for altimeters and variometers with an altitude resolution of 10 cm. This sensor is also interfaced on the same I2C bus as that of the IMU.

In order to estimate the altitude, first we have to get the pressure and temperature data from the sensor. The pressure and temperature data is obtained as 24-bit word (3 bytes) after internal ADC conversion. However, this data may not be very useful and must be calibrated and compensated before using them. The MS5611 has a 128 bit PROM where six calibration co-efficients are stored permanently. Using these co-efficients (can be read and stored on the STM32 ROM once initially), the temperature and temperature compensated pressure can be calculated. The compensation and data conversion procedure is explained in the datasheet.

Now that we have the current pressure (in millibar), by knowing the pressure at sea level (1013.25 millibar), the altitude can be estimated by using the following formula.

$$Altitude = 44330 \times \left[1 - \left(\frac{P}{P_0}\right)^{\frac{1}{5.255}}\right]$$

## 4.5 Communication and Telemetry

Fine tuning PID values of a quadcopter flight controller can be very devious and time consuming process if a programmer needs to adjust parameters in source code, compile it and upload the new program to copter every time they want to make a slight adjustment.

In addition to transferring new PID values to quadcopter, we can also request other data from the quadcopter, such as current PID values, sensor readings, flight status, motor values and any other data useful in error handling, debugging or even visualization.

A trivial problem that follows this is a standard and a secure method of data transfer. There has to some method to differentiate the various data types that we require to transmit. The *MultiWii Serial Protocol (MSP)* has been devised to solve this exact problem.

### 4.5.1 MultiWii Serial Protocol

*MultiWii Serial Protocol* is a part of the MultiWii flight controller firmware and has gained popularity and is currently widely used as the communication protocol in many flight controllers. It is a standard, secure and light weight protocol. MSP operates over UART communication. The data to be sent or received is packed into a frame. Each frame has a header, data payload and a checksum. The frame format is explained in detail below.

**Header** — Every frame begins with the characters *'$'* followed *'M'*. The next element represents the direction of the data transfer. '<' indicates that the data is being sent to the quadcopter and '>' indicates that the data is being sent from the quadcopter to a PC for telemetry or other purposes. To differentiate different types of data frames being sent (like sensor data, PID values and so on), the MSP frame assigns a 8-bit code for every frame. These range from 100-199 for a frame that is sent to the quadcopter and 200-255 for a frame that is received from the quadcopter. The last element in the header frame is the length of the payload which is useful in parsing the frame.

**Data Payload** — The payload is of fixed length and varies from frame to frame, depending on the type of the data frame (distinguished by the MSP code). The payload is usually packed and stored in the form a structure in memory in C.

**Checksum** — This is a simple method to keep the communication secure. Checksum is calculated by XORing the code, data length and all the bytes of the payload. When a frame is received the checksum must be calculated locally (on the micro-controller) and must be compared with the checksum received in the frame. A mismatch would indicate an error in the communication and the frame can be discarded.

The entire list of MSP codes and frame/payload format can be found here. Figure 4.8 shows the possible frame formats.

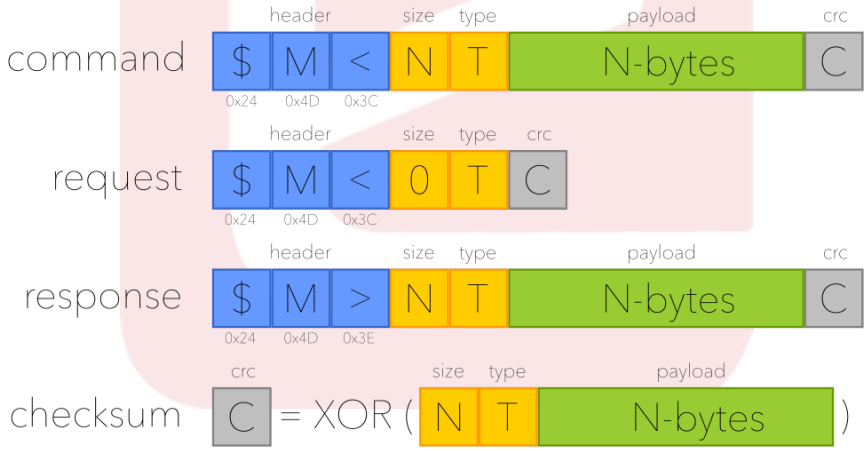


Figure 4.8: MultiWii Serial Protocol (MSP) frame format

In MSP, frames can always be sent to the flight controller (quadcopter). But, in order to receive frames from the flight controller, one has to send a request frame to the flight controller. The request frame is very similar to a command frame, except that its data length is zero and consequently, the checksum is just the MSP code.

## 4.6 Control Algorithm

The control algorithm plays a vital role in the stabilization quadcopter. The entire functionality implemented in the firmware until this point serves as the input to the control algorithm, to debug/visualize the data for designing the control architecture and finally to transfer the output of the control algorithm as PWM values to the different motors.

The stabilization control basically deals with how much power should be given to each of the motor based on the orientation of the quadcopter. To stabilize the quadcopter in mid-air, it is evident that we have to control the pitch, roll and yaw motion and hence a controller will have to be implemented in each of the axis. Another controller will have to designed to handle the throttle, given by the user to maintain a certain altitude. Another question that arises is how will the four different motors be controlled in order to stabilize. The later will be discussed towards the end of this section.

The control architecture chosen in this firmware is a parallel proportional integral derivative (PID) architecture. *Figure 4.9* shows the block diagram for the same. PID is a sum of three different terms, each of which and their contributions to the output is explained below.

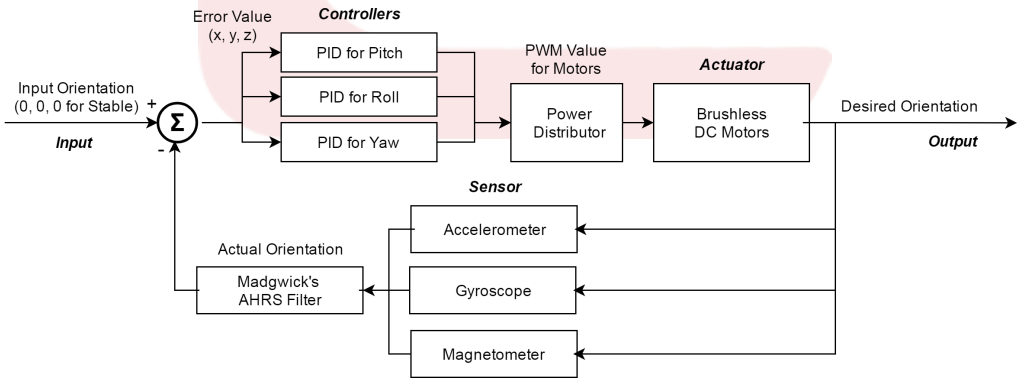


Figure 4.9: Parallel PID architecture for orientation stabilization

**Proportional Term** — The response of the proportional term is simply a constant (Kp) times the current error. Error is the difference between the desired position (Set Point) and the current position (orientation angle).

Higher the magnitude of the error, higher will be its response in the overall PID output in order to minimize the error in the current position. A very low gain (Kp) will make the quadcopter very unresponsive. Whereas, a high gain will make the quadcopter over responsive. The controller will not only correct the present error but will also introduce its own error, and in turn tries the correct this self-induced error again. Hence, it induces oscillations about the desired position. As oscillations become larger and larger, the quadcopter will became unstable and finally crash. It is evident that a steady state error always persists. Response of the proportional term is given below.

```
proportional_term = kp * error
```

**Integral Term** — The integral term sums up the amount of error over time. This term depends on the present magnitude of error as well as the past errors. The integral term helps in accelerating the current position towards the desired set point. This term will continually contribute to the overall response until the error becomes zero. It tries to eliminate any residual error (steady state) that should have been previously corrected. Higher the magnitude of the error, faster will be the response to minimize it. However, since it keeps accumulating errors over time and accelerates towards the set point, an overshoot will occur before it reaches the desired set point. Another problem is the wind-up phenomenon, which occurs when the accumulated error is beyond the maximum actuator output and can be solved by limiting its response in the output. The response of the integral term is given below.

```
integral_sum += (error * sampleTime)
integral_term = ki * constrain(integral_sum, -LIMIT, +LIMIT)
```

**Derivative Term** — The derivative term is proportional to rate of change of error. It is not dependent on the current magnitude of error and is incapable of minimizing the error on its own. The purpose of the derivative term is to anticipate the future behavior of the error. Thus, it helps in achieving the desired position much faster and reduces the unwanted overshoot caused by the integral term. The derivative term helps in damping the oscillations. However, a high magnitude of derivative gain (Kd) can induces vibrations in the quadcopter because it amplifies the noise in the system. Derivate response is given below.

```
derivative_term = kd * (current_error  previuos_error)/sampleTime
```

Finally, the desired output response can be achieved by tuning these PID gains properly to control their individual contributions in the output. The output of the PID control algorithm is given below.

```
PID_output = proportional_term + integral_term + derivative_term
```

## 4.7 Motor Operation

The *Pluto Drone* and the firmware supports upto six motors. Currently only four motors are used. The motors are coreless motors. Coreless brushed motors have very little inertia because only the coils are connected to the shaft. This is great if you want to abruptly change the speed or the direction of rotation. Coreless motors achieve high RPMs, but very little torque. Torque here is not a very serious issue as the payload of the drone is very less (60 g).

The motor speed control is achieved using *Pulse Width Modulation (PWM)*. As the duty cycle of the pulse width increases, more power is delivered to the motors in a single cycle and as a result they operate at higher speeds. The PWM values range from 0 (stopped) to 1000 (full speed).

Essentially when propellers rotate, applying the rotational analog of Newtons third law of motion, it generates a torque effect on the quadcopters body in the opposite direction. Hence, if all motors rotate in the same direction, then the quadcopter will keep rotating (or yaw) in that direction. The cause and effect is an important function to understand with quadcopter blade rotation. In order to counteract this torque effect, we need an equal amount of motors that spin in the opposite direction. *Figure 4.10* show the motor configurations. Here is a good article to understand the dynamics of a quadcopter.[14]

The next step is distribute the PID output to the different motors of the quadcopter. The power distribution is much simpler to implement than described in theory. Consider the quadcopter in *Figure 4.10* with four motor labeled M1 to M4. If the quadcopter experiences a tilt in the forward direction (the pitch axis), motors M2 and M4 will have to provide more thrust in order to bring the copter back to stable level.

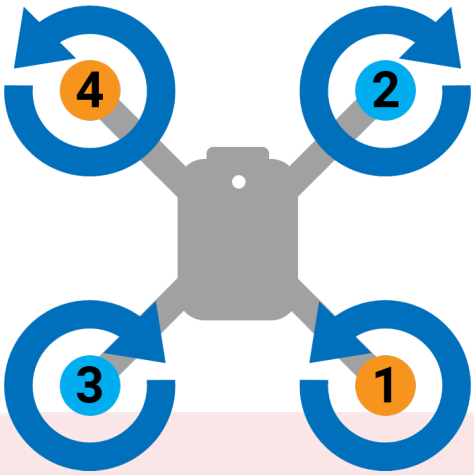


Figure 4.10: Clockwise and Anti-clockwise motor configuration for a quad-copter

Thus the pitch PID output value will be added to the front set of motors (M4 and M2) and be subtracted the back set of motors (M1 and M3). In case there a tilt in the backward direction, here the error will negative and the same distribution algorithm will work. *Figure 4.11* shows a representation of the same.

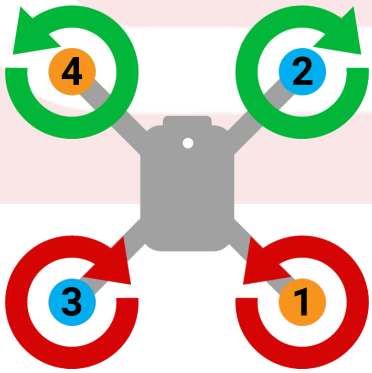


Figure 4.11: Pitch axis control (Red:0 - Green:1000)

Stabilization of the roll axis works in a similar way. But, here the tilt correction takes place in the left-right direction. Therefore, roll PID output will have to subtracted from the left set of motors (M3 and M4) and added to the right set of motors (M1 and M2) as shown in *Figure 4.12*.

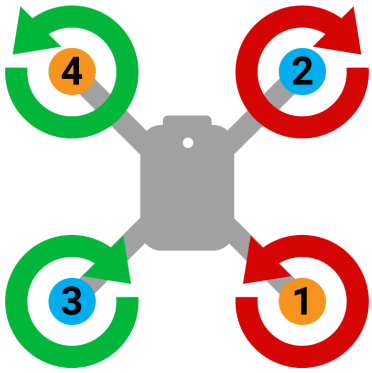


Figure 4.12: Roll axis control (Red:0 - Green:1000)

For yaw stabilization, diagonally opposite motors will have to provide more thrust to bring about a rotational torque and hence change the yaw as shown in *Figure 4.13*.

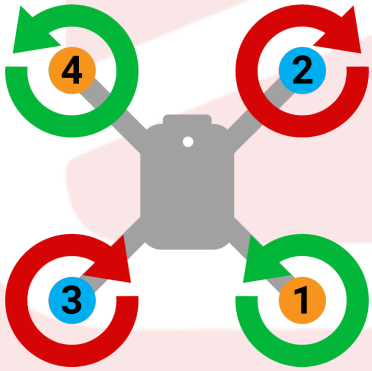


Figure 4.13: Yaw axis control (Red:0 - Green:1000)

Summing up all the outputs from the different axes, the following motor distribution algorithm can be obtained. An equal amount of thrust given to all motors varies the altitude of the drone.

```
M1 = throttle - pitch + roll + yaw  // Back Right
M2 = throttle + pitch + roll - yaw  // Front Right
M3 = throttle - pitch - roll - yaw  // Back Left
M4 = throttle + pitch - roll + yaw  // Front Left
```

## 4.8 Wireless Joystick Controller

A very common method of piloting drones is by using a RC transmitter with 6-8 channels. In this project an USB-HID gamepad controller is interfaced with Python, which in turn transmits commands (MSP frames) to the quadcopter over Wi-Fi. Other than throttle and direction control for drones, there are certain other controls that must be provided to a pilot for a stable and safe flight. Following is the list of controls implemented in the firmware.

- **Throttle** — *Range: 1000 - 2000* — This sets the base throttle value for all the motors in manual control mode and is used to control the altitude set point in auto flight mode.
- **Roll** — *Range: -60 - +60* — This channel varies the set point for the roll angle to maneuver the drone in the left-right direction.
- **Pitch** — *Range: -60 - +60* — This channel varies the set point for the pitch angle to maneuver the drone in the left-right direction.
- **Yaw** — *Range: -180 - +180* — This channel varies the set point for the yaw angle rotate the drone either in the clockwise or anti-clockwise direction.
- **Arm** — *Range: 0 or 2000 on AUX1* — Initially, when the drone in switched on, the motors must be disabled for safety. When the drone is ready for flight, the pilot must enable the motors and this is referred to as Arming the drone.
- **Altitude Hold** — *Range: 0 or 2000 on AUX2* — This control toggles between manual and auto flight modes. In manual flight mode, the user is responsible for maintaining the altitude by varying the throttle. In auto mode, the drone will hold its current altitude.
- **Pitch Trim** — *Range: Increments of $\pm 0.4$ on AUX3* — Adds an offset angle to the pitch axis so that drone is perfectly balanced about this axis.
- **Roll Trim** — *Range: Increments of $\pm 0.4$ on AUX4* — Adds an offset angle to the roll axis so that drone is perfectly balanced about this axis.

Further details on the usage of the joystick controller in explained in *section 5.2*.

# Usage and Demonstration

This chapter covers the usage of the flight controller firmware for a user as well as a developer to continue the development of this firmware.

## 5.1 Firmware Usage

The current version of the flight controller firmware is *Firmware Version 1.0.1 targeted for STM32F103C8T6* and is under development at this GitHub Repository.

### 5.1.1 Flashing the Firmware

Let us begin with building the firmware. The entire firmware is written in C language in the TrueSTUDIO IDE. To build the project, go to *Project — Build Project*. To change the output format, navigate to *Project — Build Settings — Tool Settings — Other — Output Format* and select *Intel Hex*.

The Pluto Drone has a ESP8266 and supports wireless programming. Clean-Flight Configurator can be used to flash the program. Under *Firmware Flasher* tab, load the **.hex* file and click *Flash Firmware*. Make sure that the quadcopter is powered on and the computer is connected to the Pluto Drone Wi-Fi. Once it starts to program, all the LED indicators will turn off and will flash in a sequence once the programming is completed.

### 5.1.2 Serial Communication Setup

For communication with the drone and debugging purposes, it will be easier to do so with the help of serial communication port. But the PC is connected to the drone over Wi-Fi. A virtual COM port can be created with the help of softwares such as Eltima Serial to Ethernet Connector or HW Virtual Serial Port which redirects the data from TCP/IP sockets to the COM ports.

In case any error occurs on the micro-controller, the LEDs will blink continuously in alternate pattern. The error can debugged by opening the serial terminal which will display the file name and the line number where the error had occurred. Another method which supports real-time debugging is using the ST-LINK which is covered in *subsection 3.1.2*.

### 5.1.3 Sensor Calibration

Once the basic setup is ready, the next step is to first calibrate the sensor. To do so, open the *MPU9250.c* file and **define the macro *IMU_DEBUG*** in *MPU9250.c* file. In the *main.c* file, make sure you first call the *IMU_Init()* function and then just call the required sensor read function. This will print the converted data on the serial terminal.

```
1  void setup(void)
2  {
3    Devices_Init();
4    serialBegin();
5    IMU_Init();
6  }
7
8  void main()
9  {
10   while(1)
11   {
12     // To read and print the gyro data in dps
13     MPU9250_ReadGyroData()
14
15     // To read and print the magnetometer data in milliGauss
16     //AK8963_ReadData();
```

```
17
18      // Delay to avoid hanging of the serial terminal
19      delay_ms(100);
20    }
21  }
```

To calibrate the gyroscope, *gyroBias* should be set to {*0, 0, 0*} in *MPU9250.c* file. Now keep the gyroscope sensor still and open the serial terminal. Once the data is printed on the serial terminal (around 500 samples), copy it into a spreadsheet and find out the average. Replace these average values to the *gyroBias* struct initialization.

To calibrate the magnetometer, *magBias* and *magScale* should be set to {*0, 0, 0*} in *MPU9250.c*. Build, flash the firmware and open the serial terminal. Move and rotate the sensor through a variety of figure-8 patterns and do it for around 1000 samples. Copy this data into a spreadsheet and save it as a **.csv* file. Use the *mag-calib.py* Python script and set the file name to the **.csv* with the data samples. Run the script and the hard iron and soft iron biases will be displayed. A plot similar to the one shown in *Figure 4.7* will pop-up as well. The response should be almost spherical in all the axes and must be centered at the origin. Replace the *magBias* struct initialization values with hard iron bias values and *magScale* with the soft iron biases.

### 5.1.4 Using the MultiWii Serial Protocol Library

MSP can be used for communication and telemetry by calling the *MSP_Update()* function in *main.c* file. Ideally, this should be sufficient for both, sending and receiving MSP frame. However, there are certain issues with it currently and the UART communication process hangs (*Refer section 7.1*). As of now, only receiving MSP frames work with *MSP_Update()* and can is sufficient to interface the joystick controller. To send MSP frames for telemetry, you have to add the following snippet in the *taskScheduler()* in *main.c* file.

```
1  if ((millis() - last_tick) > 100)
2  {
3    /** Save Time */
4    last_tick = millis();
```

```
5
6     /* Telemetry */
7     MSP_SendIdent();
8     MSP_SendStatus();
9     MSP_SendMotor();
10    MSP_SendAttitude();
11    MSP_SendAltitude();
12    MSP_SendRawIMU();
13    MSP_SendPID();
14  }
```

The following snippet is an example to update a particular MSP frame and can be done in any other file by just including the *msp.h* header file.

```
1   #include "msp.h"
2
3   // MSP frame structure to hold the PID data to be sent
4   extern msp_pid msp_txf_pid;
5
6   void PID_UpdateMSP()
7   {
8     // Update pitch PID gains
9     msp_txf_pid.pitch.p = pid_pitch.con_KP * 255;
10    msp_txf_pid.pitch.i = pid_pitch.con_KI * 255;
11    msp_txf_pid.roll.d = pid_pitch.con_KD * 255;
12
13    // Update roll PID gains
14    msp_txf_pid.roll.p = pid_roll.con_KP * 255;
15    msp_txf_pid.roll.i = pid_roll.con_KI * 255;
16    msp_txf_pid.roll.d = pid_roll.con_KD * 255;
17
18    // Update yaw PID gains
19    msp_txf_pid.yaw.p = pid_yaw.con_KP * 255;
20    msp_txf_pid.yaw.i = pid_yaw.con_KI * 255;
21    msp_txf_pid.yaw.d = pid_yaw.con_KD * 255;
22  }
```

When using the MSP_Update() function, once a MSP frame is received, the frame is parsed and a callback function is called based on the MSP code. These callback function are defined with the *__weak* symbol and are not implemented in the *msp.c* file.

```
__weak void MSP_SetMotor_Callback()
{
  /* Prevent unused argument(s) compilation warning */
  UNUSED(MSP_SET_MOTOR);

  /* NOTE: This function can be modified, when the callback is needed,
  or MSP_SetMotor_Callback() can be implemented in the user file*/
}
```

The user must implement this function in any other file, but without the *__weak* symbol. For example, the following snippet implements the callback for receiving the motor PWM values sent from a PC.

```
#include "msp.c"

extern msp_set_motor msp_rxf_motor;

void MSP_SetMotor_Callback()
{
  float m1, m2, m3, m4;

  m1 = constrain(msp_rxf_motor.motor[0] - 1000, 0, 1000);
  m2 = constrain(msp_rxf_motor.motor[1] - 1000, 0, 1000);
  m3 = constrain(msp_rxf_motor.motor[2] - 1000, 0, 1000);
  m4 = constrain(msp_rxf_motor.motor[3] - 1000, 0, 1000);

  Motor_SetRawSpeed(m1, m2, m3, m4);
}
```

## 5.2 Wireless Joystick Controller

To pilot the quadcopter and to control the various flight parameters, a joystick controller is required. In this project, the Xbox® One Controller[15] is used. this controller is a USB-HID Gamepad compliant device and is interfaced with Python with the help *pygame* module. The inputs from the controller are read, parsed, converted and then packed into MSP frames and sent to the quadcopter. The communication process is same as explained in *subsection 5.1.2*.


Figure 5.1: Labeled diagram showing the mapping of the different controls on the Xbox® One Controller[15]

To use the joystick controller, just plug in the controller to any USB port and run the *joystick-xbox.py* script. This script depends on the *MultiWii.py* module. A CLI will open as shown in Figure 5.4. First Arm the drone, then gradually increase the throttle. The various controls are explained in *section 4.8*.

## 5.3 Results and Demonstration

The IMU and magnetometer were successfully interfaced and fused using AHRS filter to get the attitude and heading. The barometer was also interfaced and altitude was estimated.

MultiWii Serial Protocol (MSP) was successfully implemented in the firmware and *Figure 5.2* shows the data obtained from the flight controller. Flight parameters and data like attitude, heading, altitude, motor PWM values, raw IMU data, PID gains and several other data was obtained and visualized in MultiWii Conf, which is open source software written in *Processing*. This is the *link* for the video demonstration of the same.

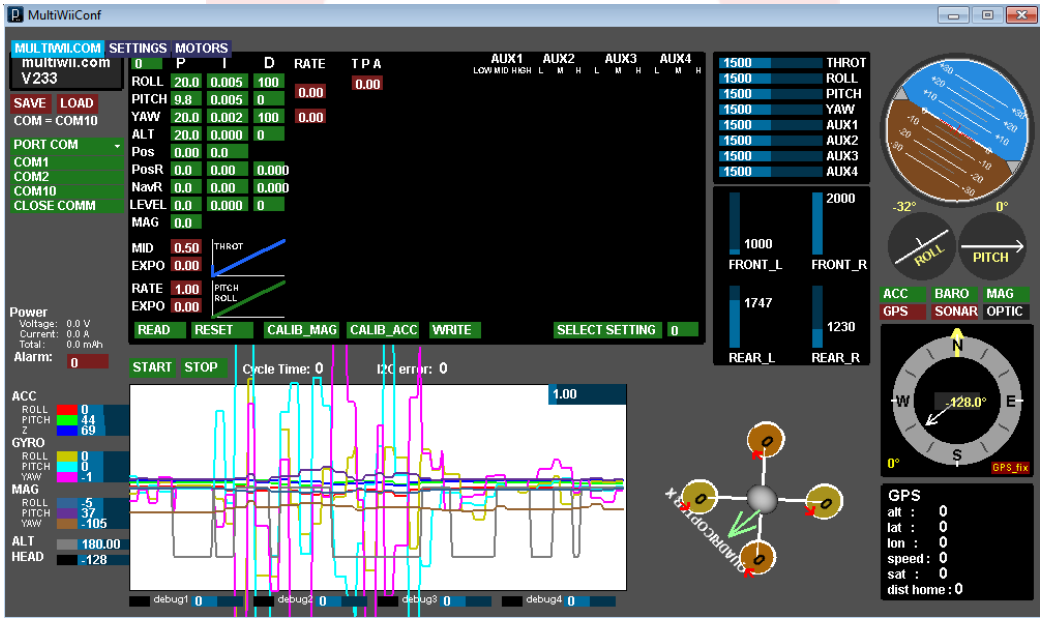


Figure 5.2: Flight parameters and data visualized in MultiWii Conf software using MSP

Fine tuning PID values of a quadcopter flight controller can be very devious and time consuming process if a programmer needs to adjust parameters in source code, compile it and upload the new program to copter every time they want to make a slight adjustment. To make the process of PID tuning easier, a GUI in Python was developed.

This GUI helps in tuning the pitch, roll, yaw and throttle axes. MSP is used for communication with the quadcopter. Apart from setting PID gains, one can set the trim for any axis or change the set point. Throttle value can also be set. Various MSP frames such as status, identifier and raw IMU data can also be requested from the drone through this GUI. This is the *link* for the video demonstration of the PID tuning process using the GUI.

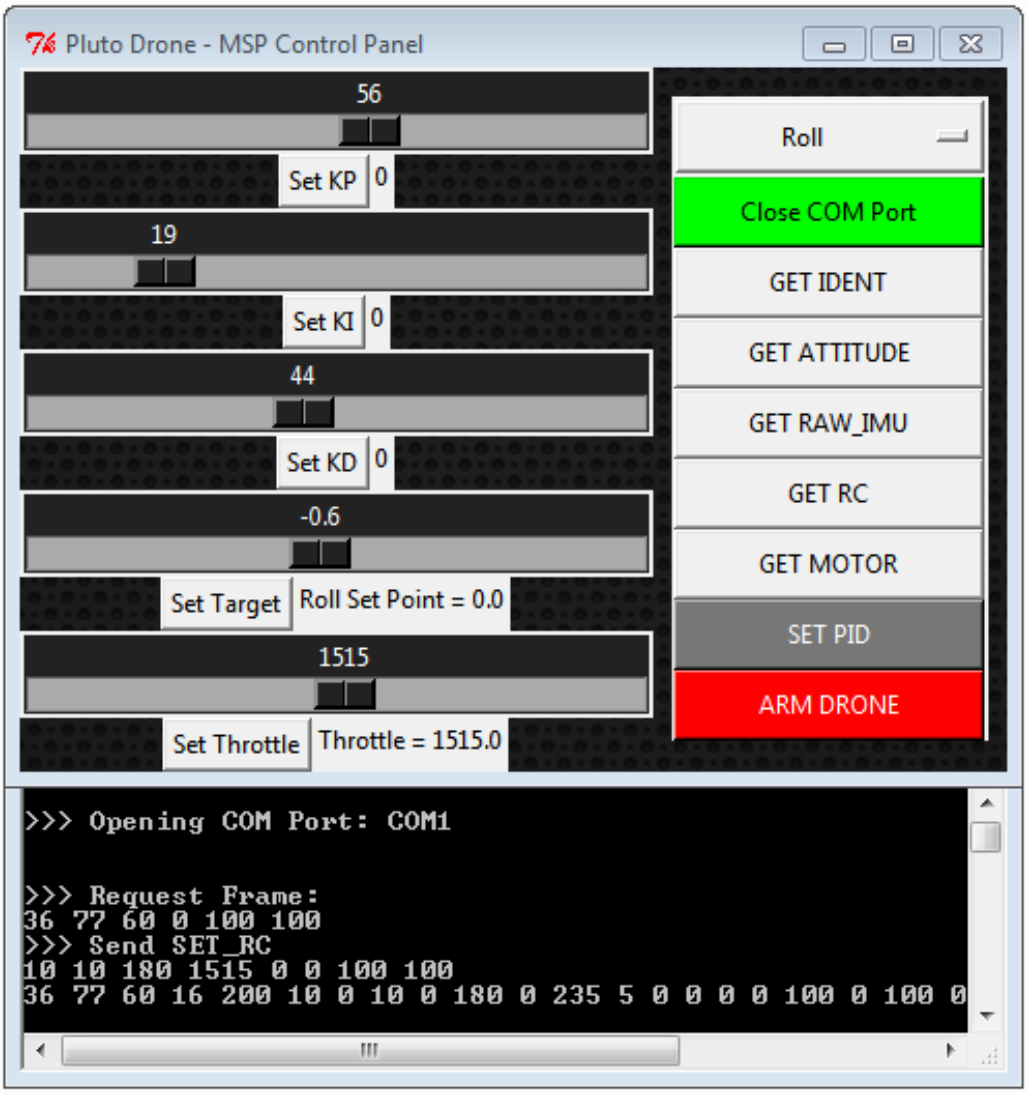


Figure 5.3: A GUI developed in Python for tuning of PID parameters

The joystick controller was successfully interfaced with Python using the *pygame* module. A CLI in Python was developed to visualize the joystick controls as shown in *Figure 5.4*. The various controls supported are shown in *Figure 5.1*. Vibration feedback from the joystick to the user is also implemented. This is the *link* for the video demonstration of the joystick controller interfaced.

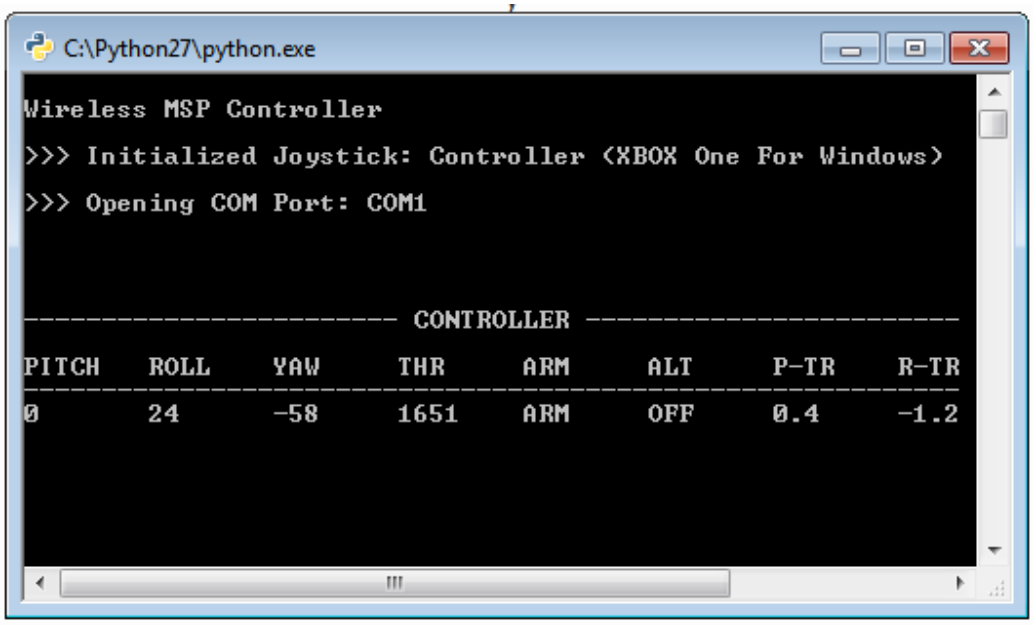


Figure 5.4: A CLI developed in Python for the wireless joystick controller

The PID gains were tuned for pitch, roll and yaw axes and the drone response was quick and stable for each of the axes individually. However, when PID for all three axes was enabled, there were slight offsets due to shift in CG. Successful flight was achieved upto a height of 5 m. However, due to certain offsets, the drone drifts quite a lot in the pitch and roll axis and the flight is not very stable. This is the *link* for the video demonstration of the quadcopter flight.

# Future Work

A flight controller firmware is a very complex one as it has to support various hardware devices and should be designed in a way such that the addition of new hardware or expansion in functionality must be an easy process. For example, if a new module such as GPS or XBee communication is required, such a change must easily be able to be incorporated into the the current firmware.

Another aspect of a flight controller firmware is that it should be somewhat more generic and must be able to support constant hardware modifications, especially in this case as flight hardware has a greater probability of incurring damage. From this perspective, following is a list of features that can be integrated into the current firmware in the future.

- **Altitude Stability** — In the current firmware, the MS5611 barometer is interfaced and estimated altitude is obtained as well. There is also provision made for altitude control in the PID algorithm. However, altitude hold and stability is still not complete. Ranging sensor such as a LASER sensor or Ultrasonic sensor can be fused with the barometer data to obtain a better estimate of the altitude.

- **Control Algorithm** — Currently, the motors need to be well calibrated so that each of them provide roughly an equal amount of thrust. So either by fine tuning the PID gains or re-implementing the control algorithm, achieving stable flight for any get set of motors is a task.

- **Auto Sensor Calibration** — Auto calibration of sensors, the gyroscope and magnetometer in particular, is another feature that can in built within the firmware itself. Currently, sensor calibration is performed externally by sending data samples to a Python script.

- **Localization** — The current firmware has no support for localization. Adding features such as GPS for outdoor way-point navigation and return-to-home feature would be very helpful. The MSP currently implemented already supports the addition of GPS module.

- **Safe Landing Feature** — The flight dynamics vary a lot when the battery level is low. In fact, the micro-controller begins to reboot several times due to insufficient power when all the motors are on. Interfacing the battery monitoring sensor (already present on-board) and implementing a safe landing feature once the voltage level drops below a certain voltage should be integrated with the firmware.

- **Error Handling** — There are several sources of error that may occur in the firmware such as I2C errors, UART errors and so on. Currently if an error occurs, the program control transfers into a while loop and prints the file name and line number on the serial terminal, at where the error had occurred. To make the firmware more robust and safe, better error handling can be implemented.

- **Custom Joystick Controller** — In this project, a Xbox® One [15] USB-HID gamepad controller is used. In the future a custom wireless controller can be developed from scratch using the ESP8266 Wi-Fi module.

# Bug Report and Challenges

## 7.1 Bug Report

Following is a list of several bugs with fixes as well as some bugs that are not fixed yet. Currently, the *_Error_Handler()* function in *peripherals.c* only prints the file name and the line at which the error occurs. The user has to manually debug the error.

1. **Bug** — Peripheral driver function or some macros not defined.
   **Fix** — Recheck the peripheral driver library used. The two libraries available are Standard Peripheral Library and STM32Cube HAL library. Refer *Section 3.2*.

2. **Bug** — I2C read or write error.
   **Fix** — Recheck SCL/SDA connections. Check the initialization of I2C bus in *peripherals.c*. I2C error also occurs when the sensor is not powered properly or the battery is low.

3. **Bug** — Incorrect readings of IMU data.
   **Fix** — Check if scaling factor has been chosen properly and the full scale range settings are correct. Check if the typecasting is done correctly when packing the high and low byte into a 16-bit signed value.

4. **Bug** — I2C error while interfacing the Magnetometer.
   **Fix** — Check if the magnetometer has been enable via the MPU-9250 as the auxiliary sensor. Refer *Section 4.3.2*.

5. **Bug** — AHRS filtered angles are drifting.
   **Fix** — Check if the AHRS integration time is set properly. Check if the beta value has been set (0.6). Gyroscope rate should be in radians per second for the AHRS filter, however, the rate is senr in dps and is internally converted to rps in *MadgwickAHRS.c* file.

6. **Bug** — Yaw angle is drifting or always converging to a particular angle when rotated.
   **Fix** — Magnetometer requires hard and soft iron calibration to be performed as the bias value will change quite often.

7. **Bug** — Used I2C_ReadByteArray for reading the 3 bytes from the MS5611 ADC register.
   **Fix** — Perform multiple read using from the same register using I2C_ReadBytes (Note the 's') and specify the number of bytes to be read in the function call (Here it is 3). I2C_ReadByteArray is used to read only one byte from a series of continuosly incrementing registers.

8. **Bug** — UART interrupt communication not working.
   **Fix** — UART interrupts have callback functions defined with *__weak* attribute and must be implemented in the user file.

9. **Bug** — MSP two-way communication not working *or* MSP not working with MultiWii Conf.
   **Possible Fix** — Currently there is a problem with UART communication when multiple frames or large amount of serial data is sent and received together. One of the possible solutions would be to use UART interrupts for sending data as well. Another solution would be to use the UART DMA peripheral.

10. **Bug** — *micros()* function does not work properly.
    **Possible Fix** — Currently the implementation of *micros()* and *millis()* is the same. *micros()* does not work properly, when the SysTick is configured to interrupt every 1 µs.

# References


[1] Geoffrey Brown, *Discovering the STM32 Microcontrollers*, 2016.

[2] STMicroelectronics, *Mainstream Performance line, ARM Cortex-M3 MCU*.

[3] ARM Developer, *GNU ARM Embedded Toolchain*.

[4] STMicroelectronics, *STM32F1xx HAL Manual*

[5] Carmine Noviello, *Setting up a GCC/Eclipse toolchain for STM32Nucleo - Part 1*

[6] Simply Embedded, *UART Receive Buffering*

[7] Wikipedia Article, *Attitude and heading reference system*

[8] Arducopter, *Arducopter Flight Modes*, 2013

[9] InvenSense, *MPU-9250 Nine-Axis (Gyro + Accelerometer + Compass) MEMS MotionTracking Device*

[10] Toni Corinne, *Accelerometer Basics* at SparkFun

[11] A1ronzo, *Gyroscope* at SparkFun

[12] Kris Winer, *Simple and Effective Magnetometer Calibration*

[13] Sebastian O.H. Madgwick, *An efficient orientation filter for inertial and inertial/magnetic sensor arrays*

[14] V. Kadamatt, *Quadcopter blade rotation and lift : How and why*

[15] Xbox® One is a trademark of Microsoft Corporation. The Xbox® Logos are trademarks of Microsoft® Corporation.