The Scout Flight Controller

By Tim Hanewich

Table of Contents

My Greatest Engineering Accomplishment: The Scout Flight Controller

The Scout Flight Controller running on a quadcopter

Over the course of the past six years, I have dedicated my professional focus to various software development disciplines — back-end development, database design, software architecture, front-end frameworks, DevOps, API integration, and more. These domains have become my passion and I consider myself incredibly fortunate to have a career that allows me to focus on these in my daily work.

In April 2023, I embarked on an ambitious project that demanded the extension of my skills into the hardware realm: **the development of a custom flight controller for a quadcopter drone**. Quadcopters have always fascinated me as I've marveled at the simultaneous elegance and sophistication of these machines that is often overlooked. I recognized that creating my own flight controller would be a significant undertaking, likely my most challenging project yet. It presented an interesting opportunity to translate my software expertise into a tangible, real-world application.

What is a Flight Controller?

In contrast to airplanes, which possess various active aerodynamic elements such as wings, rudders, stabilizers, and ailerons that allow them to control their attitude while in flight, quadcopters have a distinct lack of any aerodynamic elements. Given this lack of active aerodynamics, quadcopters are *inherently unstable*. To stay airborne and make attitude adjustments (e.g., pitch forward, roll right, yaw left), a quadcopter must meticulously make minute adjustments to the thrust applied to each of its four motors. This intricate process of adjusting the thrust of each motor occurs rapidly, often several thousand times per second!

This is the role of the **flight controller** — a vital component of a quadcopter drone that acts as its "brain". The flight controller is

responsible for controlling and stabilizing the flight of the quadcopter while it is airborne. More specifically, it is responsible for ensuring that the quadcopter manages stable flight at all times and complies with the attitude "requests" of it's pilot.

Introducing "Scout"

Quadcopter flight controllers are very complex; consumer drones like those made by DJI run proprietary flight controller software that engineers spend years developing and perfecting. Hobbyists that assemble drones for activities like FPV flying often choose an off-the-shelf flight controller from large companies with heavy R&D funding like the Speedybee F405, Speedybee F7, or KK2, among others.

Rather than relying on an off-the-shelf solution, I took on the challenge of developing my own flight controller from scratch. While commercial alternatives can easily cost well over $100, my flight controller runs on nothing more than a $4 Raspberry Pi Pico.

I named my flight controller *Scout*. Just as a scout ventures into uncharted territories to gather knowledge and pave the way for future success and innovation, this flight controller represents my entry into a new domain of expertise, one of which I knew very little about when beginning this project.

Scout Demo Flight

Scout's Development

Scout was not easy to develop. I started the project on April 29, 2023. 70 days and 1,194 code commits later, on July 7, 2023, Scout achieved its first successful flight:

Scout's First Flight

This was a very tedious and challenging task — one that I contemplated giving up numerous times during its lengthy development. Despite the countless failed tests, injuries, crashes, fires, and more, I remained steadfast in my efforts. Now with a fully functioning and stable-flying flight controller, I'm thankful I persevered. When Scout made its first successful flight, I was filled with mixture of relief, excitement, and partial embarrassment that is took me so long!

How I made Scout

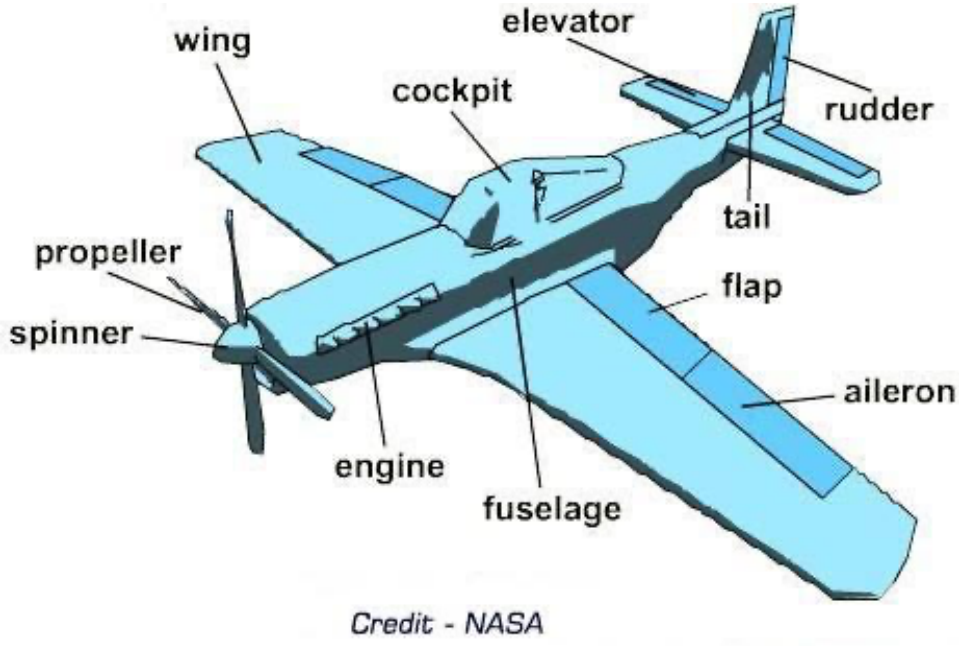
In the following weeks, I will release a series of chapters detailing how I created the Scout Flight Controller. These chapters will delve into the technical concepts required for a quadcopter flight controller and will cover the basics of quadcopter flight dynamics.

Quadcopter Flight Dynamics

As mentioned in the introduction to the Scout Flight Controller, a flight controller plays a vital role in the operation of any multicopter drone. Serving as the "brain" of a quadcopter, the flight controller performs rapid mid-flight adjustments to the thrust of each motor, ensuring stable flight while responding to the pilot's inputs.

Fixed-Wing Aviation

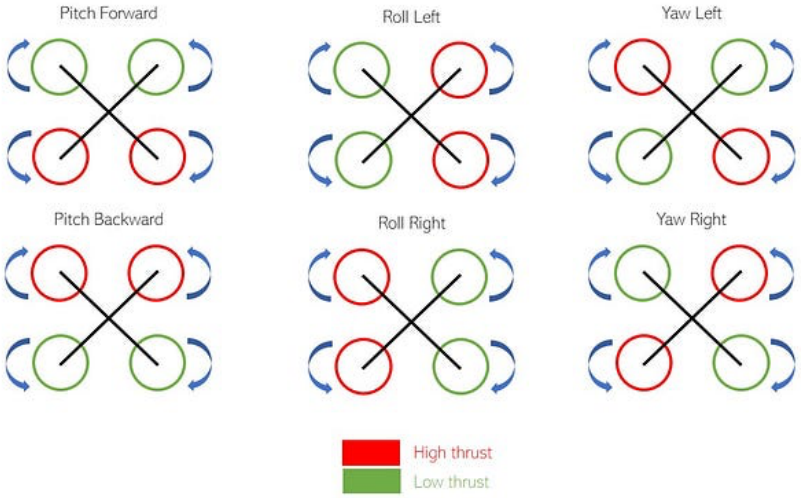
In traditional fixed-wing aviation, airplanes benefit from passive aerodynamics. The presence of aerodynamic elements such as wings, ailerons, flaps, elevators, stabilizers, and rudders allows airplanes to maintain stable flight. Pilots can alter the flight trajectory or attitude of the aircraft by adjusting the angles of these elements. The airflow over these elements generates lift, enabling the aircraft to maintain stable flight.



Airplanes use aerodynamic elements to control airflow

Quadcopter Aviation

However, the principles of fixed-wing aviation do not apply to quadcopters, as they lack any inherent aerodynamic elements. Unlike airplanes, quadcopters solely rely on the thrust generated by multiple motors to stay airborne. Specifically, a quadcopter controls its attitude by adjusting the thrust levels of its four motors, as depicted below:



Quadcopters control attitude by varying the thrust applied to each motor

By using the flight dynamics illustrated above, the quadcopter can control its pitch, roll, and yaw.

At first glance, it might seem intuitive to program a flight controller based on these basic principles. For example, if the pilot wants to pitch forward, the thrust on the front two motors should be decreased while increasing the thrust on the back two motors. Similarly, rolling to the right would involve increasing the thrust on the left motors and decreasing it on the right motors. If you're anything like me when I embarked on this project, this approach may appear to be a sufficient solution. However, this would be a mistake.

Although it may seem that applying equal thrust to all four motors would result in stable, level flight, this assumption is *far* from accurate. Even the slightest misalignment in weight distribution, wind conditions, or motor speeds can introduce chaotic instability. Moreover, despite using identical make and model motors, minute differences in each unit's build will result in slightly varied thrust levels. These slight discrepancies would lead to **highly** unstable flight conditions.

Unfortunately, quadcopters are inherently unstable for these reasons. They constantly need to balance the thrust among all four motors, finding the perfect mixture to achieve the desired attitude set by the pilot. In this context, "constantly" means numerous adjustments per second. Many off-the-shelf flight controllers perform these mid-flight adjustments at an astonishing rate of several thousand times per second! In my case, **I have programmed the Scout Flight Controller to execute this adjustment loop at a frequency of 250 hertz**, corresponding to 250 adjustments per second.

In other words, a quadcopter flight controller constantly compares the *desired attitude of the pilot* against the *actual attitude* of the vehicle. In uses methods we will get into later in this series to determine what level of thrust to apply to each motor to meet the desires of the pilot, and it does so at rapid speeds (250 times per second in Scout's case).

This system — the flight controller — is what I've developed from scratch.

Flight Control Modes

Quadcopter flight controllers primarily operate in one of two distinct "flight modes":

The first mode is known as **angle mode**. In this mode, when the pilot releases the attitude stick, the quadcopter automatically

maintains a parallel orientation to the ground, with a 0 degree pitch and roll angle. Pushing the stick all the way forward will cause the quadcopter to pitch forward until it reaches the maximum allowable attitude angle. The quadcopter will retain this position until the pilot adjusts the input again.

The second mode is **rate mode**, also referred to as **acro mode** or **gyro mode**. This mode represents a more advanced form of flight. In rate mode, the quadcopter does *not* measure its precise angle in relation to the ground. Instead, it focuses solely on tracking the rate of rotation in each of the three axes and compares it to the pilot's desired rate of rotation in those same axes. For example, pushing the stick forward prompts the quadcopter to pitch forward continuously until the pilot releases the stick, at which point the quadcopter seeks to maintain its existing attitude. Unless there is a deliberate change in attitude by the pilot, the quadcopter will retain the attitude previously set by the pilot. This means that if the pilot desires a level orientation, they need to manually readjust the quadcopter — it will not automatically level itself. While rate mode presents a more challenging control scheme, it enables higher performance, greater maneuverability, and is exclusively used in FPV drone racing.

The Scout Flight Controller implements the **rate mode** functionality.

Next Chapter

We have now gained an understanding of how a quadcopter flight controller achieves the desired attitude by continually comparing the quadcopter's *actual* attitude with the *desired* attitude, making rapid adjustments to the thrust of each motor to attain the desired attitude or rate of attitude change.

Now, let's explore how the quadcopter monitors its attitude and rate of attitude change, which brings us to the essential role of the gyroscope. Continue reading in the next chapter here!

Capturing Telemetry with the Gyroscope

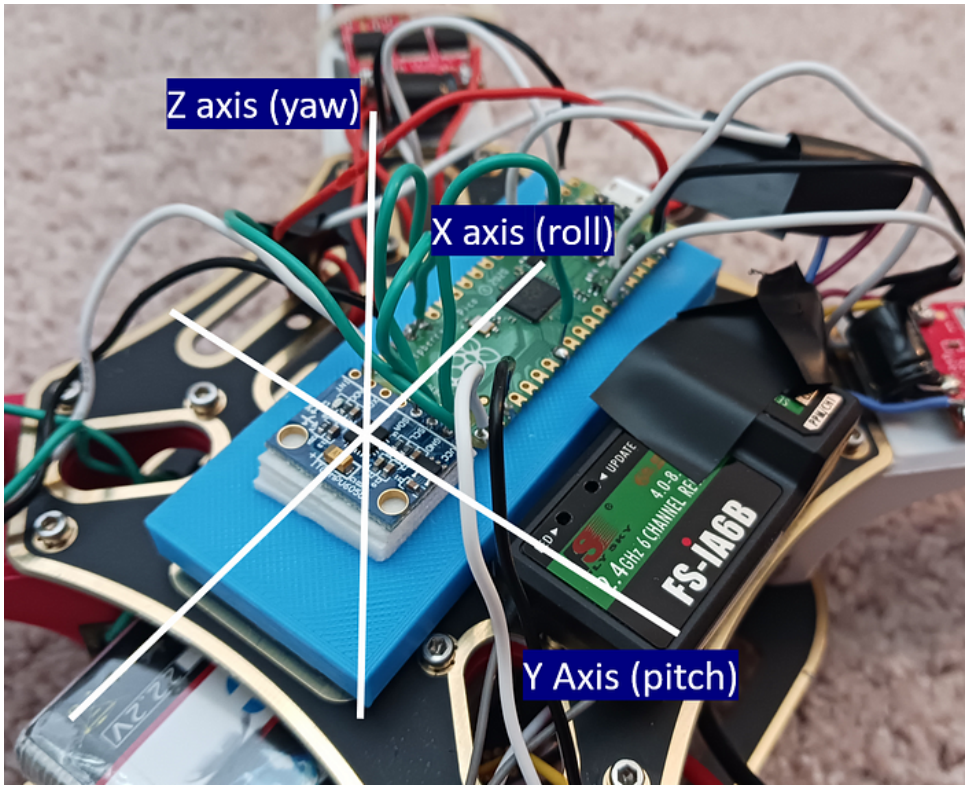
In the previous chapter of this series, where I discussed the development of the custom flight controller named *Scout*, we explored how a quadcopter flight controller achieves stable flight. By constantly comparing the actual attitude (or rate of change in attitude) of the quadcopter to the desired attitude (or rate of change in attitude) of the pilot, the flight controller determines the appropriate level of thrust to apply to each motor, thus reaching the desired state. **But how does it monitor its own attitude?**

A **gyroscope** is a device that is used to measure angular velocity. In other words, a gyroscope is a module that measures the rate of rotation around all three axes. Gyroscopes are extremely common and can be found in almost every cell phone made today. Gyroscopes also play a *critical* roll in avionics, and in particular, multicopters.

Rate Mode with the Gyroscope

As mentioned in a previous article, the Scout flight controller implements a **Rate Mode** — meaning that the quadcopter's flight controller is exclusively focused on achieving a specific *rate of rotation*, not orientation angle to the ground. A gyroscope is perfect for this scenario. A flight controller determines motor thrust needed by comparing the actual attitude rate of change to the desired attitude rate of change — a gyroscope allows the flight controller to know one half of this equation.

I mounted an MPU-6050 (on a GY-521 breakout board) onto the quadcopter in the following orientation. The MPU-6050 is a common low-cost gyroscope that is used in many commercial and hobby applications.



With the MPU-6050 lined up along all three axes as seen above, we can visually see how a rotation of the MPU-6050 in any direction can be understood as a change in the roll, pitch, or yaw of the drone. The mounting position and orientation is **extremely important**. The MPU-6050 *must* be mounted in the following ways:

- In the center of the drone (where it turns on its axis). Mounting the gyroscope in an area that is *not* at its

center may cause instability, as a true *rotation* of the drone may be interpretted as an *acceleration* instead. For the readings to be as "clean" as possible, it must be mounted very near to the center of the drone.

- With the axes of the MPU-6050 lined up as closely as possible with the axes of the quadcopter. Line up the X axis of the gyroscope with the X axis (roll) of the quadcopter, the Y axis with the Y axis (pitch) of the quadcopter, and Z axis with the yaw angle of freedom.

I2C

I2C, or Inter-Integrated Circuit, is a serial communication format invented in the early 1980s that allows for the communication between two devices: the "master", which is reading from and writing to a peripheral device for its own purposes, and the "slave", the device itself that is providing a service to the master device.

I2C allows for bidirectional communication between the two devices via only two wires — one for data transmission, known as "SDA", and one for a clock signal, known as "SCL", which plays a crucial role in synchronizing the data exchange between the master and slave devices. The Master reads data from and writes data to the Slave device by reading and writing to specific *registers* of the Slave device.

Fortunately, I2C is a well-understood and implemented communication protocol; the MicroPython implementation for the Raspberry Pi Pico that we are developing in has a full module, `machine`, for interfacing with peripheral (slave) devices over I2C.

As described in the [MPU-6050 datasheet], the MPU-6050 uses the I2C protocol to communicate with its parent (master) device.

Interfacing with the MPU-6050

I've put together a MicroPython class that allows for communication with the MPU-6050, and thus, allows for the collection of gyroscope telemetry:

```
"""
A lightweight MicroPython implementation for
interfacing with an MPU-6050 via I2C.  Author: Tim
Hanewich - https://github.com/TimHanewich Version:
1.0License: MIT License
Copyright 2023 Tim Hanewich
Permission is hereby granted, free of charge, to
any person obtaining a copy of this software and
associated documentation files (the "Software"), to
deal in the Software without restriction, including
without limitation the rights to use, copy, modify,
merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to
whom the Software is furnished to do so, subject to
the following conditions:
The above copyright notice and this permission
notice shall be included in all copies or
substantial portions of the Software.
THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY
OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT
LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR
COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES
OR OTHER LIABILITY, WHETHER IN AN ACTION OF
CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF
OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
OTHER DEALINGS IN THE SOFTWARE.
"""import machineclass MPU6050:
```

```
    """Class for reading gyro rates and
acceleration data from an MPU-6050 module via
I2C."""

    def __init__(self, i2c:machine.I2C, address:int
= 0x68):
        """
        Creates a new MPU6050 class for reading
gyro rates and acceleration data.
        :param i2c: A setup I2C module of the
machine module.
        :param address: The I2C address of the MPU-
6050 you are using (0x68 is the default).
        """
        self.address = address
        self.i2c = i2c

    def wake(self) -> None:
        """Wake up the MPU-6050."""
        self.i2c.writeto_mem(self.address, 0x6B,
bytes([0x01]))    def sleep(self) -> None:
        """Places MPU-6050 in sleep mode (low power
consumption). Stops the internal reading of new
data. Any calls to get gyro or accel data while in
sleep mode will remain unchanged - the data is not
being updated internally within the MPU-6050!"""
        self.i2c.writeto_mem(self.address, 0x6B,
bytes([0x40]))

    def who_am_i(self) -> int:
        """Returns the address of the MPU-6050
(ensure it is working)."""
        return self.i2c.readfrom_mem(self.address,
0x75, 1)[0]

    def read_temperature(self) -> float:
        """Reads the temperature, in celsius, of
the onboard temperature sensor of the MPU-6050."""
        data = self.i2c.readfrom_mem(self.address,
0x41, 2)
        raw_temp:float =
self._translate_pair(data[0], data[1])
        temp:float = (raw_temp / 340.0) + 36.53
        return temp    def read_gyro_range(self) ->
int:
        """Reads the gyroscope range setting."""
        return
self._hex_to_index(self.i2c.readfrom_mem(self.addre
ss, 0x1B, 1)[0])

    def write_gyro_range(self, range:int) -> None:
        """Sets the gyroscope range setting."""
        self.i2c.writeto_mem(self.address, 0x1B,
bytes([self._index_to_hex(range)]))

    def read_gyro_data(self) -> tuple[float, float,
float]:
        """Read the gyroscope data, in a (x, y, z)
tuple."""

        # set the modified based on the gyro range
(need to divide to calculate)
        gr:int = self.read_gyro_range()
        modifier:float = None
        if gr == 0:
            modifier = 131.0
        elif gr == 1:
            modifier = 65.5
        elif gr == 2:
            modifier = 32.8
        elif gr == 3:
            modifier = 16.4

        # read data
        data = self.i2c.readfrom_mem(self.address,
0x43, 6) # read 6 bytes (gyro data)
```

```
        x:float = (self._translate_pair(data[0],
data[1])) / modifier
        y:float = (self._translate_pair(data[2],
data[3])) / modifier
        z:float = (self._translate_pair(data[4],
data[5])) / modifier

        return (x, y, z)

    def read_accel_range(self) -> int:
        """Reads the accelerometer range
setting."""
        return
self._hex_to_index(self.i2c.readfrom_mem(self.addre
ss, 0x1C, 1)[0])

    def write_accel_range(self, range:int) -> None:
        """Sets the gyro accelerometer setting."""
        self.i2c.writeto_mem(self.address, 0x1C,
bytes([self._index_to_hex(range)]))

    def read_accel_data(self) -> tuple[float,
float, float]:
        """Read the accelerometer data, in a (x, y,
z) tuple."""

        # set the modified based on the gyro range
(need to divide to calculate)
        ar:int = self.read_accel_range()
        modifier:float = None
        if ar == 0:
            modifier = 16384.0
        elif ar == 1:
            modifier = 8192.0
        elif ar == 2:
            modifier = 4096.0
        elif ar == 3:
            modifier = 2048.0

        # read data
        data = self.i2c.readfrom_mem(self.address,
0x3B, 6) # read 6 bytes (accel data)
        x:float = (self._translate_pair(data[0],
data[1])) / modifier
        y:float = (self._translate_pair(data[2],
data[3])) / modifier
        z:float = (self._translate_pair(data[4],
data[5])) / modifier

        return (x, y, z)

    def read_lpf_range(self) -> int:
        return self.i2c.readfrom_mem(self.address,
0x1A, 1)[0]

    def write_lpf_range(self, range:int) -> None:
        """
        Sets low pass filter range.
        :param range: Low pass range setting, 0-6.
0 = minimum filter, 6 = maximum filter.
        """        # check range
        if range < 0 or range > 6:
            raise Exception("Range '" + str(range)
+ "' is not a valid low pass filter setting.")

        self.i2c.writeto_mem(self.address, 0x1A,
bytes([range]))
    #### UTILITY FUNCTIONS BELOW ####

    def _translate_pair(self, high:int, low:int) ->
int:
        """Converts a byte pair to a usable value.
Borrowed from https://github.com/m-
rtijn/mpu6050/blob/0626053a5e1182f4951b78b8326691a9
223a5f7d/mpu6050/mpu6050.py#L76C39-L76C39."""
        value = (high << 8) + low
```

```
        if value >= 0x8000:
            value = -((65535 - value) + 1)
        return value       def _hex_to_index(self,
range:int) -> int:
        """Converts a hexadecimal range setting to
an integer (index), 0-3. This is used for both the
gyroscope and accelerometer ranges."""
        if range== 0x00:
            return 0
        elif range == 0x08:
            return 1
        elif range == 0x10:
            return 2
        elif range == 0x18:
            return 3
        else:
            raise Exception("Found unknown gyro
range setting '" + str(range) + "'")

    def _index_to_hex(self, index:int) -> int:
        """Converts an index integer (0-3) to a
hexadecimal range setting. This is used for both
the gyroscope and accelerometer ranges."""
        if index == 0:
            return 0x00
        elif index == 1:
            return 0x08
        elif index == 2:
            return 0x10
        elif index == 3:
            return 0x18
        else:
            raise Exception("Range index '" + index
+ "' invalid. Must be 0-3.")
```

Don't be too concerned with the minute details of using I2C within the code above! Understanding how to use the class above is plenty sufficient for this project.

An example of using the `MPU6050` class seen above:

```
import machinei2c = machine.I2C(0,
sda=machine.Pin(12), scl=machine.Pin(13)) #
creating the object that allows for I2C
communication in MicroPythonimu = MPU6050(i2c) #
passing the i2c object to the MPU6050 class above.
This class will handle all communicationsimu.wake()
# wakes up the MPU-6050 (it may have been in sleep
mode)gyro_data =
imu.read_gyro_data()print(gyro_data) # (0.346823, -
0.198345, 0.023958)
```

Low Pass Filtering

The MPU-6050 is very sensitive. Naturally, when the motors begin rotating, there are minor vibrations from the four motors and propellers. While we cannot see them with our eyes, the MPU-6050's gyroscope picks them up. This can be interpreted as erratic changes in attitude rate of change and thus can be hugely inhibiting to stable flight.

Fortunatlely, the MPU-6050 has a built-in low pass filter for this type of scenario. The onboard low pass filter filters out low-end frequencies, like those created by the motors/propellers. By turning this low pass filter setting on, we can get a much cleaner stream of incoming telemetry that better represents the attitude rate of change of the vehicle (it is not perfect, but vastly improved).

To activate the MPU-6050's onboard digital low pass filter (extending from the chunk of code above):

```
imu.write_lpf_range(5) # Turning the low pass
filter setting to level 5 (out of 6)
```

Accounting for Gyroscope Bias

Every gyroscope exhibits a phenomenon known as "bias". When leaving the gyroscope perfectly still and running several consecutive readings, you will notice that no gyroscope will ever read exactly 0.0, 0.0, 0.0, indicating a rotational velocity of zero in each axis. Instead, consistent readings of seemingly random values are obtained, averaging around readings like 0.346823, -0.198345, 0.023958, or even varying further in certain scenarios.

This inherent deviation is referred to as "bias". Despite the gyroscope remaining perfectly still, there will still be at least *some level* of indicated rotation. No gyroscope is perfect, and bias exists for several reasons — imperfections in manufacturing, temperature variations, residual stress, electromagnetic interference, aging, etc.

Obviously we cannot pass these raw values to the quadcopter flight controller as this would lead to the flight controller to believe that the vehicle is constantly undergoing a slight rotation, despite it remaining motionless. To rectify this issue, we must account for gyroscope bias with a **calibration**.

The calibration process is conceptually simple: upon system boot-up, we dedicate a few seconds to monitor the raw gyroscope readings **while the quadcopter remains perfectly stationary**. We then calculate the average of these observed values across all three axes. In all subsequent gyroscope readings, we will subtract each axis' respective calibration reading (the average we calculated) to get the "calibrated" reading; essentially, we are *subtracting* the bias *out of* the raw reading. This adjustment effectively eliminates the bias from the raw readings, providing us with a "clean" reading that can be utilized within our flight controller code. Details of this calibration process will be explained in a future chapter.

Using the Gyroscope Telemetry

By reading data from the MPU-6050 gyroscope via the I2C protocol, the Scout flight controller now possesses the ability to monitor its own change in attitude. **Now what do we do with this data?**

This data — the quadcopters *actual* rate of rotation must be compared against the pilots *desired* rate of rotation. Now that we have one half of the equation, we must acquire the other: the pilots *desired* rate of change in attitude.

In the next chapter you'll learn how an onboard FS-iA6B radio receiver allows Scout to receive attitude adjustment commands from the pilot via a radio transmitter. This is the final piece of data the flight controller requires before it can determine what levels of thrust to apply to each motor.

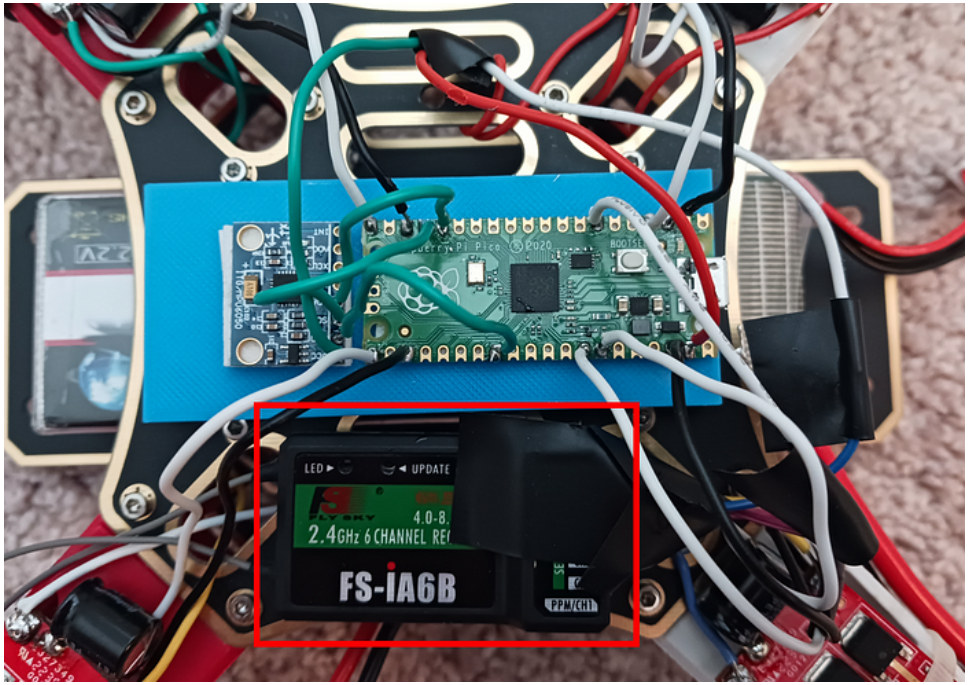
Receiving Control Inputs via an RC Receiver

In the previous chapter on gyroscopes we learned that a quadcopter flight controller achieves stable flight by comparing the *actual* rate of attitude change against the pilot's *desired* rate of atttitude change. In then determines the precise thrust to apply to each of its four motors to achieve (comply with) the aforementioned desired rate of attitude change.

In the last chapter, I explained how an inexpensive **MPU-6050** module is onboard my *Scout* quadcopter and how this serves as the gyroscope which allows the Scout Flight Controller to determine its roll, pitch, and yaw rate of change; this is the *actual* rate of change data. Next, we need to collect the *desired* rate of change data from the pilot. We will do this via an RC receiver.

Beside the Raspberry Pi Pico (microcontroller that runs the Scout Flight Controller) and the MPU-6050, a FlySky FS-iA6B Receiver is mounted directly to the frame. This module allows the Scout Flight Controller to receive control inputs from the pilot over a 2.4 GHz radio frequency.

The FS-iA6B receiver module can be seen below:



The pilot will be using a FlySky FS-i6 Transmitter to transmit control commands to the drone.

Reading from the FS-iA6B Receiver

I won't go into heavy detail about how the FS-iA6B works (PWM, PPM, etc.). Scout reads incoming data from the FS-iA6B receiver module via its **ibus** protocol functionality. **Ibus** is a protocol that uses a UART serial interface to transmit the value of each channel from the receiver. In other words, by connecting our FS-iA6B to one of the Raspberry Pi Pico's UART-capable pins, we can perform some code that will allow us to read each channel (inputs from the pilot).

Fortunately, the community has our back. I found this fantastic video by the YouTube channel **Penguin Tutor** that describes, in detail, the ibus protocol, how we can leverage it, and even provides sample code for doing so.

You can find the sample code that Penguin Tutor provided on his website here. I will provide the class definition that he wrote below as well. This is the class that we will use to read (and parse) channel values from the FS-iA6B receiver:

ibus.py

Interpreting the Data

The `read` method of the `IBus` class, for a 6-channel transmitter like the one I am using (FS-i6), will return a list of seven integer values. For example:

```
[1, 1499, 1498, 1002, 1500, 1000, 1000]
```

- Index 0 (`1` in this case) is the **status** of this returned list. `1` means this is a valid data packet (new values).
- Index 1 (`1499` in this case) is the **horizontal axis of the right stick**. We'll use this to control the **roll** rate. The value ranges from `1000` (full left) to `2000` (full right). At approximately 1500, the stick is in the middle position, untouched.
- Index 2 (`1498` in this case) is the **vertical axis of the right stick**. We'll use this to control the **pitch** rate. The value ranges from `1000` (full up) to `2000` (full down). At approximately 1500, the stick is in the middle position, untouched.

- Index 3 (`10002` in this case) is the **vertical axis of the left stick**. We'll use this to control **average thrust (throttle)** of all four motors, controlling if the quadcopter is ascending or descending. The value ranges from `1000` (full down) to `2000` (full up). At approximately 1000, the stick is at its lowest position.

- Index 4 (`1500` in this case) is the **horizontal axis on the left stick**. We'll use this to control **yaw** rate. The value ranges from `1000` (full left) to `2000` (full right). At approximately 1500, the stick is in the middle position, untouched.

- I customized index 5 (`1000` in this case) to read the values of the `SWA` switch on the transmitter. This is the small switch at the top left part of the controller (see picture above). 1,000 is the *off* position (up) while 2,000 is the *on* position (down). You must configure your FS-I6 RC controller to provide the value of this switch in the setting menu of the transmitter.

- I customized index 6 (`1000` in this case) to read the values of the `SWB` switch on the transmitter. This is the large switch at the top left part of the controller, directly to the right of SWA. 1,000 is the *off* position (up) while 2,000 is the *on* position (down). You must configure your FS-I6 RC controller to provide the value of this switch in the setting menu of the transmitter. We will not be using the value of this switch in this version of the Scout Flight Controller.

Normalizing Input Values

Instead of working with these values that range from 1,000 to 2,000, I preferred to *normalize* each within the range of -1.0 to 1.0 for the roll, pitch, and yaw, and between 0.0 and 1.0 for the throttle. This makes the values easier to understand and work with. I wrote the following simple function in Python to handle the normalization process:

```
def normalize(value:float, original_min:float, original_max:float,
new_min:float,          new_max:float)          ->          float:
    """Normalizes (scales) a value to within a specific range."""
    return new_min + ((new_max - new_min) * ((value - original_min)
/ (original_max - original_min)))
```

The method above takes the `value` you provide and scales it to a new value between the `new_min` and `new_max` values you provide, proportionally to the `original_min` and `original_max` values. We will use this in the Scout Flight Controller code to normalize the roll, pitch, yaw, and throttle inputs.

Using the Control Data

Now with both the *actual* attitude rate of change values from the gyroscope (see the last chapter) and the *desired* attitude rate of change values form the RC receiver (covered above), we now have the full set of telemetry required to calculate precisely how much thrust to apply to each of the four motors to *achieve* the desired attitude rates.

In the next chapter, we'll learn how a system called a *PID Controller* is implemented in quadcopter flight controllers that determine the optimal level of thrust to apply at any moment to achieve the desired attitude rates.

Stabilizing Flight with PID Controllers

In the previous chapters in this series, we explored the crucial steps of capturing telemetry data through the gyroscope and acquiring pilot input via the RC receiver. Through these processes, we discovered that a quadcopter flight controller determines the appropriate thrust to be applied to each of its four motors by comparing the actual rate of change of attitude (roll, pitch, and yaw) with the desired rate of change. In essence, the quadcopter adjusts the thrust delivered to each motor in order to execute the desired maneuvers as intended by the pilot.

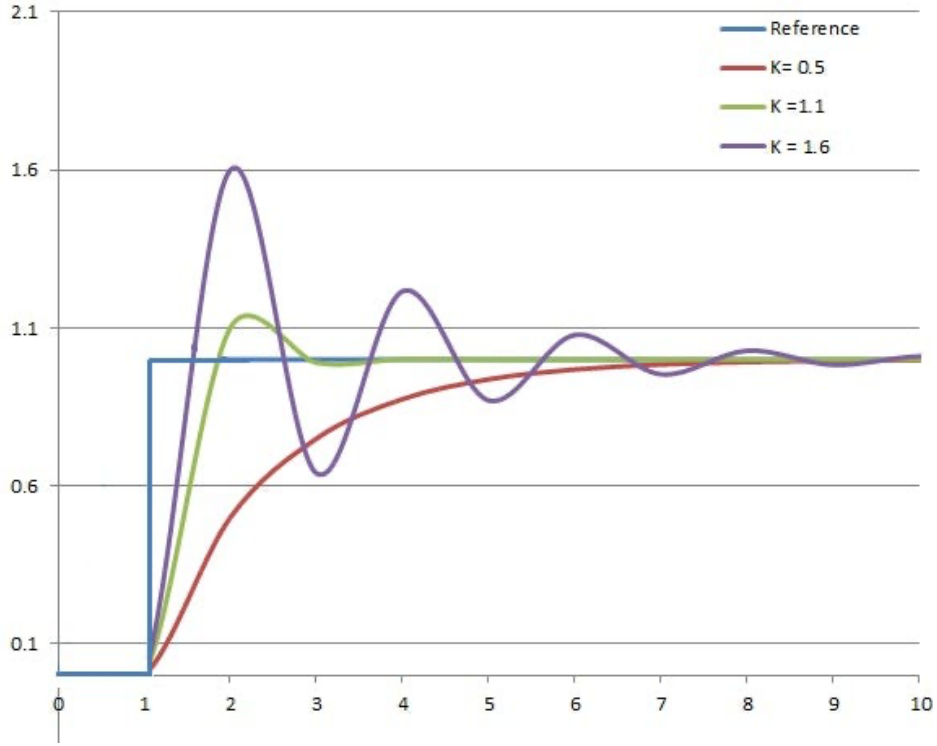
Why do we need a PID controller?

Given the availability of both **gyroscope telemetry** (actual rate of change) and **pilot input** (desired rate of change), how does our quadcopter flight controller precisely calculate the required throttle input for each motor?

One could employ a simple algorithm to tackle this problem. For instance, if the actual roll angle measures +20°/second (indicating a right bank) while the desired angle is 0°/second (representing a stable state), one might apply a proportionally higher thrust to the right motors and a correspondingly lower thrust to the left motors, effectively correcting the roll. This approach might show promise, and it does possess elements reminiscent of the PID controllers commonly employed in modern quadcopter flight controllers. However, it falls short of sufficiency and often results in instability during flight due to the occurrence of overshoot errors.

What does a PID controller do?

A PID (Proportional, Integral, and Derivative) controller represents a simple mathematical system that dictates how a quadcopter attains the desired state specified by the pilot, considering the current actual state of the vehicle.



The diagram above depicts the PID system, with the blue line representing the **goal** or "setpoint." The purple, green, and red lines illustrate distinct approaches through which a PID controller can operate to reach the desired goal. The user aims for the PID controller to approximate a goal value of 1.0, with the different lines representing various means of achieving this (such as by adjusting thrust levels).

In the case of the purple line example, the PID controller significantly **overshoots** the target. It applies excessive pressure, reaching the goal too quickly and overshooting it. Subsequently, it compensates by exerting less pressure, overshoots again, and repeats this process until it eventually achieves the desired goal.

Conversely, the red line example shows the PID controller **undershooting** the target. It applies insufficient pressure, causing a slow approach towards the goal. Although it avoids overshoot, this approach might prove too weak for a quadcopter flight controller.

The green line example demonstrates the most optimal behavior. The PID controller reaches the goal relatively swiftly, experiences a slight overshoot, but promptly corrects itself. Among the three examples, this approach strikes the best balance.

So, how exactly does a PID controller achieve this?

How does a PID controller work?

PID stands for *Proportional*, *Integral* and *Derivative*. These three "terms" collaborate to determine the throttle input for each motor, blending their outputs to produce an appropriate level of thrust at any given moment during flight.

The PID controller primarily focuses on the concept of error. Here, error denotes the discrepancy between the desired rate the pilot intends to achieve and the actual current rate along a specific axis.

- **Proportional**: The proportional component states that the thrust that should be applied should be *proportional* to the error. i.e. if the desired roll rate is +25°/second and our actual roll rate is -30°/second, greater thrust should be applied than in a scenario where our desired roll rate is +25°/second and our actual roll rate is 15°/second. This term asserts that *the level of thrust we should apply should be proportional to the error we are encountering*.

- **Integral**: The integral term bears resemblance to the proportional term, but it also takes cumulative error into account. Rather than solely considering a thrust correction proportionate to the error, the integral component continues to increase thrust (growing progressively) as long as the error persists. For example, if a +40°/second error remains unresolved (the pilot's control input fails to rectify it), the integral component will continue escalating the recommended thrust with each passing second.

- **Derivative**: Unlike the proportional and integral components, the derivative component disregards the error itself, concentrating solely on the change in error. Put simply, it disregards whether the quadcopter is rolling to the left while the pilot desires a rightward roll; it simply aims to stabilize the situation. It is specifically engineered to counteract abrupt changes. In essence, it focuses on the change in error, rather than the error itself. Within the context of a quadcopter flight control system, it can be likened to a "fast twitch" mechanism that rapidly corrects momentary errors.

In a quadcopter flight control system, **three PID controller systems operate in unison: one for each axis (roll, pitch, and yaw).**

In addition to independently calculating the proportional, integral, and derivative terms, a unique *gain* value is applied to each term, allowing for the scaling up or down of the calculated value to a suitable level for the specific system utilizing the PID controller. These **gains** are typically denoted as kP, kI, and kD, representing the proportional, integral, and derivative gains, respectively.

The PID controllers function within a **PID controller loop**. Multiple times per second, the *desired* rates and the *actual* rates for each of the three axes (roll, pitch, and yaw) are fed into their corresponding PID controller systems. Each PID controller system calculates the proportional, integral, and derivative components, scales them using their respective gain values, combines them, and assigns the resulting thrust value to the applicable motors.

Integrating the PID Controller Output into the Flight Controller

As mentioned previously, the thrust levels for all four motors are determined by considering the outputs of all three PID controllers. Understanding the **flight mechanics of a quadcopter** outlined in the first chapter, we can assign each PID controller's output to the throttle of the corresponding motor:

Snippet of code from the Scout Flight Controller:

```
#              calculate              throttle              values
t1:float  =  adj_throttle  +  pid_pitch  +  pid_roll  -  pid_yaw
t2:float  =  adj_throttle  +  pid_pitch  -  pid_roll  +  pid_yaw
t3:float  =  adj_throttle  -  pid_pitch  +  pid_roll  +  pid_yaw
t4:float = adj_throttle - pid_pitch - pid_roll - pid_yaw
```

In the provided code snippet, the appropriate thrust levels (throttle input) for each of the four motors are calculated. `t1` corresponds to the front left motor, `t2` to the front right, `t3` to the rear left, and `t4` to the rear right. Each throttle value is calculated based on the pilot's desired throttle (adj_throttle), incorporating inputs from the three PID controllers and factoring them into the computation.

Given that `t1` and `t2` relate to the thrust of the front motors, while `t3` and `t4` pertain to the thrust of the rear motors, we need to inverse the application of the pitch PID value. Consequently, we add it to the front two motors and subtract it from the rear two motors. As explained in the first chapter of this quadcopter flight mechanics series, a quadcopter pitches forward by increasing the thrust of the rear motors while simultaneously reducing the thrust of the front motors. Hence, we reverse the operation by flipping the addition/subtraction operators. The same principle applies to the roll and yaw axes.

PID Controller Code

Below, you will find a straightforward implementation of a PID controller in Python. This mathematical system is utilized within the Scout Flight Controller to maintain stable flight.

Upon creating an instance of the `PIDController` class, you can assign the desired values for the proportional gain, integral gain, and derivative gain. To ensure proper functioning of the integral and derivative terms (which rely on time-based calculations), you must provide the `cycle_time_seconds` parameter. This value corresponds to the delay or interval between each PID loop iteration. In my case, the Scout Flight Controller operates at 250 Hz, resulting in a cycle time of 0.004 seconds (1 second / 250 = 0.004 seconds). Additionally, an optional parameter `i_limit` exists to mitigate risks associated with unbounded integral term growth. As mentioned earlier, the integral term could inadvertently cause the motors to reach 100% thrust due to an uncorrected error (e.g., if the drone becomes entangled in a tree). Such a thrust level presents safety concerns such as the risk of injury or fires.

After instantiating the `PIDController` class, you can utilize the `calculate` function to obtain the recommended thrust value based on the error (the difference between the actual and goal values). This function calculates the precise thrust by factoring in the proportional, integral, and derivative terms.

Remember that the integral and derivative terms are time-dependent during flight, necessitating the use of the `previous_error` and `previous_i` variables mentioned earlier. It is vital to reset these values before each new flight. Resetting the PID state variables to 0 using the `reset` function ensures that no carryover occurs from previous flights, facilitating a fresh start.

What's next?

In this chapter, we learned about the role of a PID controller in determining the appropriate thrust levels for each of a quadcopter's four motors. By leveraging the PID controller's recommendations and adjusting the throttle inputs in a specific manner, we enable the PID controller to regulate the motor thrust and achieve stable, level flight.

In the next chapter, we will learn how we can set each motor's input throttle to the recommended throttle by the PID controllers using **pulse width modulattion (PWM)**.

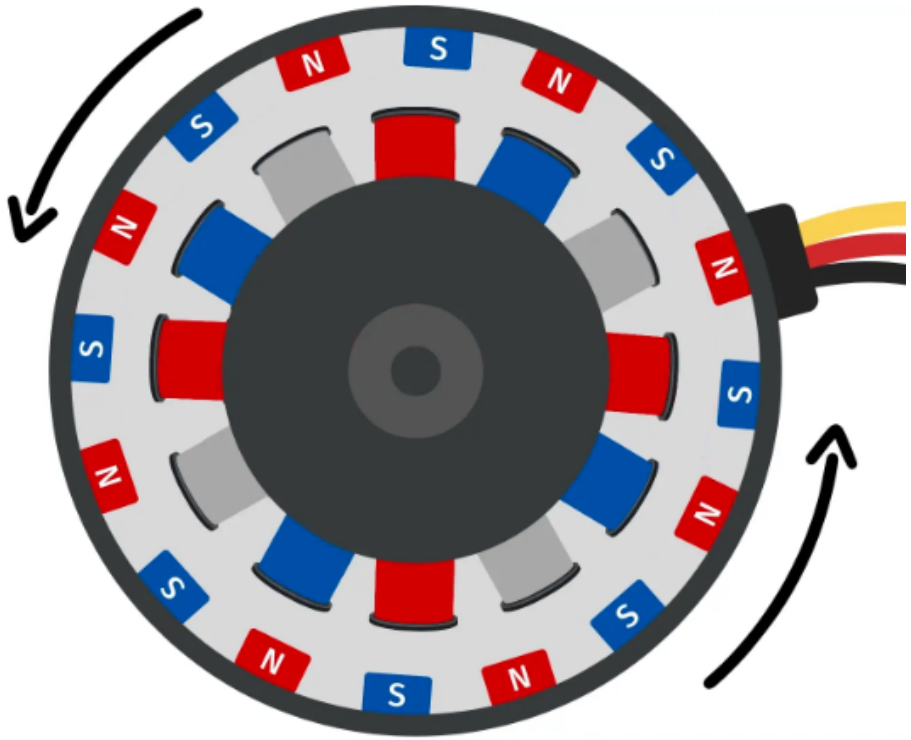
Controlling Brushless Motors with ESC's and PWM

In the previous chapters, we learned that a quadcopter flight controller achieves stable flight by continuously comparing its *actual* attitude against its *desired* attitude. This discrepancy, known as the **error**, is then passed to a crucial component called the **PID Controller**. The primary responsibility of the PID controller is to determine the precise level of thrust needed for each of the four motors in order to attain the desired attitude or attitude rate of change mentioned earlier.

Now that we have the recommended thrust (throttle) for each motor from the PID controller (refer to the code snippet in the previous chapter), the question arises: How do we effectively apply this throttle to the corresponding motor?

How a Brushless Motor Rotates

A brushless motor operates through the interaction of electric currents and magnetic fields to generate rotational motion. More specifically, brushless motors rely on the principle of magnetic attraction and repulsion.



The above image depicts the basic components of a brushless motor. It consists of two main parts: the stator and the rotor. The stator remains stationary and houses coils of wire, while the rotor, which rotates around the stator, is composed of magnets arranged in a specific pattern.

To achieve rotational motion, the stator's coils must rapidly alter the magnetic force in each coil shown above. This causes the rotor, which is magnetically attracted and repelled by different poles within the stator, to initiate motion gradually. By rapidly changing the magnetic force in each coil according to a specific sequence, the rotor begins to rotate.

The speed at which these magnetic fields are switched within the motor directly affects the rotation rate of the rotor, thereby generating thrust. To attain the necessary revolutions per minute (RPM) for lifting the quadcopter off the ground, this magnetic field must be switched at an extremely rapid pace — approximately 21,600 times per second, or every 0.00004 seconds! That's fast!

However, microcontrollers such as the Raspberry Pi Pico used in the Scout Flight Controller, as well as other microcontrollers, lack the capability to switch at such high speeds. As a result, quadcopters rely on a crucial piece of hardware called an **electronic speed controller (ESC)**.

The Role of the ESC

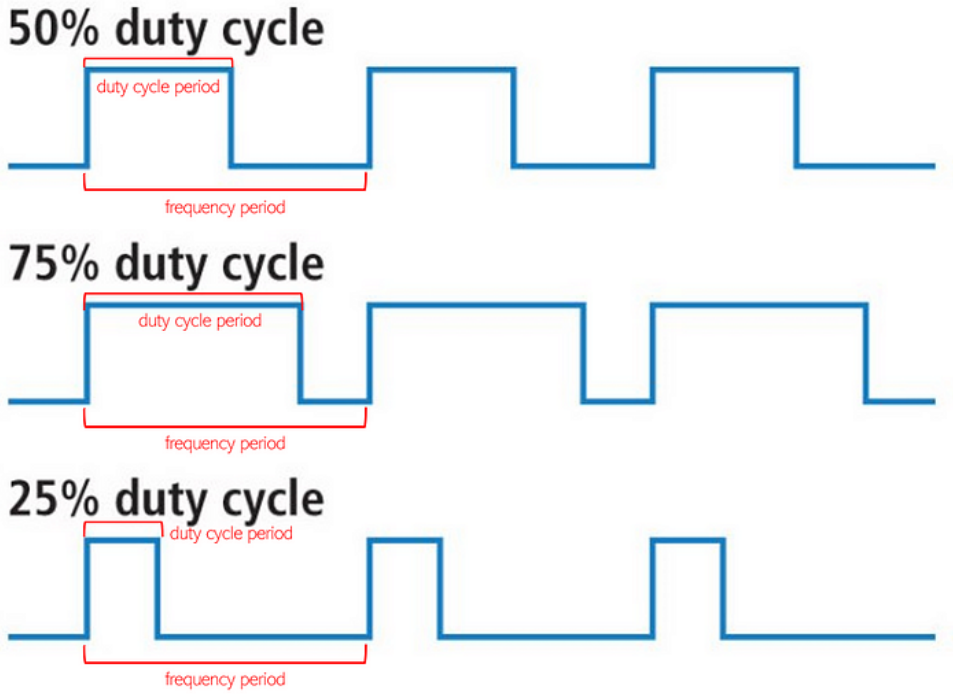
The ESC acts as an intermediary between the microcontroller and the brushless motor. It receives the desired throttle level (RPM speed) specific to each motor from the flight controller and performs the necessary conversion to switch the magnetic fields within the motor at the appropriate rate, achieving the desired throttle level.

The ESC is capable of providing the brushless motor with the required power in the form of three-phase power. The motor's three wires are soldered directly to the ESC, which converts the desired throttle input from the microcontroller into three-phase power and supplies it to the motor, thus generating rotational force.

Now, the question remains: How does the microcontroller transmit the desired input throttle to the ESC? The ESC employs a communication format known as **pulse width modulation (PWM)** as the input signal.

Pulse Width Modulation

Pulse Width Modulation, abbreviated as PWM, is a modulation technique commonly used in electronics to control the average power delivered to a device or component. It achieves this by varying the width of pulses in a digital signal. Consider the following graphic:



A PWM signal consists of two essential components: the **frequency** and the **duty cycle**.

The **frequency** of the PWM signal determines the duration of each period within the signal. For instance, if our PWM signal

operates at a frequency of 50 Hz, this means each period lasts for 0.02 seconds (1/50 = 0.02).

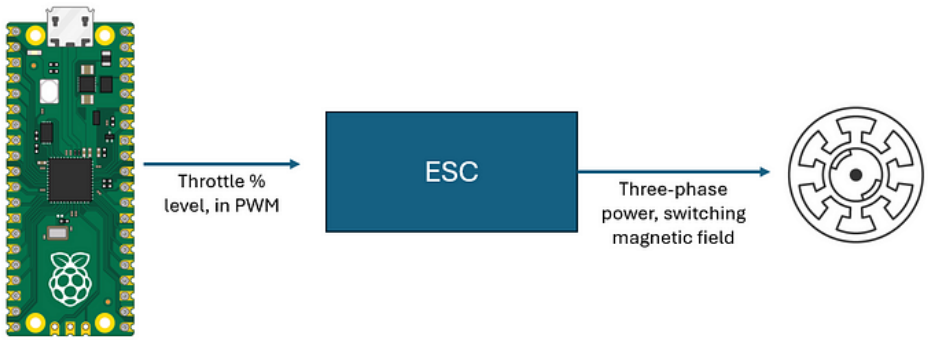
The **duty cycle** of the PWM signal defines the duration for which the signal remains at a high voltage level during the frequency period, with the remaining time being at a low voltage level.

For example, at a frequency of 50 Hz, a 50% duty cycle implies that the voltage of the signal remains high for 50% of the time, which is 0.01 seconds, while it remains low for the remaining 50% of the time, also 0.01 seconds. This signal is repeatedly transmitted via the PWM signal wire: high for 0.01 seconds, low for 0.01 seconds, high for 0.01 seconds, low for 0.01 seconds, and so on.

If the frequency remains at 50 Hz but the duty cycle is set to 75%, the voltage will be high for 0.015 seconds and low for 0.005 seconds. This cycle continues to repeat.

Controlling Motor Speed from the Microcontroller

Visually, this is the system implemented to control each of the quadcopter's four motors:



- The microcontroller conveys the desired throttle level to the ESC using a PWM signal.
- The ESC receives the input throttle level, converts it to the appropriate three-phase speed, and switches the magnetic field within the motor's stator accordingly.
- Simultaneously, the magnets in the rotor attract and repel against the various coils in the stator, resulting in rotation.

Each of the quadcopter's four motors will be equipped with an ESC, and the microcontroller will have one PWM signal wire for each motor, enabling precise speed control for each individual motor.

What's next?

At this stage in the series, we understand that a quadcopter achieves stable flight by continuously comparing its actual rate of attitude change with the desired rate of attitude change. This disparity, known as the error, is subsequently provided to a PID controller, which determines the necessary adjustments in motor thrust to achieve the desired rate of attitude change.

Finally, the PID controller's recommended thrust level is transmitted to the corresponding motors via a PWM signal into an electronic speed controller (ESC).

In the next chapter, we will explore the hardware side — specifically, how everything is wired together to create a functioning quadcopter.

Hardware

In the past several chapters of this series, we learned:

- The flight controller compares the actual attitude rate of change with the desired attitude rate of change, using inputs from the onboard gyroscope and RC receiver module.
- This discrepancy, known as the *error*, is passed to several **PID Controllers** that determine what level of thrust to apply to each of the four motors.
- The flight controller then sets each motor's speed to the recommendation of the flight controller using **pulse width modulation** (PWM).
- This cycle is repeated many times per second to achieve stable flight.

The above procedure runs in a continuous loop while in flight. For every second in flight, the Scout Flight controller runs repeats the above loop 250 times (250 Hz). So, for a flight as short as 60 seconds, the thrust of each motor was updated 15,000 times!

Now, with a foundational understanding of how the flight controller software works, let's take a closer look at the hardware involved.

Frame

The quadcopter structure based on the F450 Frame, a widely used and inexpensive drone frame originally created by DJI.

This lightweight and durable frame comes with an integrated PCB (Power Circuit Board) for convenient power distribution to each of the four motors.

Battery

Scout uses a 3,300 mAh 4S lithium polymer (LiPo) battery. The "4S" designation indicates that the battery consists of four LiPo cells connected in series. Each fully charged LiPo cell provides 4.2V, resulting in a total voltage of 16.8V when all four cells are fully charged (4.2 x 4 = 16.8).

Lithium Polymer batteries offer several advantages over other types of batteries, making them popular in quadcopter applications:

- **High energy density**: despite their compact size, LiPo batteries can store a significant amount of energy.
- **Lightweight**: LiPo batteries are lighter compared to other battery types, making them ideal for aviation.
- **High discharge rate**: LiPo batteries are capable of providing *a lot of current* at any moment. In other words, they can "empty the gas tank" quite quickly. This is important in quadcopters as this will allow the quadcopter to provide heavy amounts of thrust when

needed to maintain stable flight or comply with a pilot's maneuver.

Motors

Scout is equipped with four XING-E Pro 2207 2750 KV Brushless Motors, known for their exceptional power, particularly when operating at 4S. These motors are commonly used in high-speed and performance-oriented FPV drone racing.

In the full flight controller code, I have limited the maximum throttle for each motor to only 22% of their peak thrust level. Even at this level, the motors provide satisfactory performance and are capable of recovering from moderate dives. With the knowledge I have gained throughout this project, I would have chosen less powerful motors if given the opportunity to make the selection again.

Propellers

Scout uses four 8x4.5 2-blade propellers, with two rotating clockwise and two rotating counterclockwise. The "8x4.5" specification indicates that the propellers have an 8-inch diameter and a pitch angle of 4.5 inches. The pitch represents the theoretical distance the propeller would move forward in one revolution if it were moving through a solid medium.

Propellers come in various configurations, including two-blade, three-blade, and four-blade options (and beyond). After some light research, I decided to use two-blade propellers due to their efficiency. Although three-blade propellers may offer enhanced performance, my project focused on stability rather than pure performance.

Electronic Speed Controllers

As mentioned in the previous chapter on ESCs and PWM, the Scout quadcopter employs four electronic speed controllers (ESCs) to drive each of the four motors. These ESCs are responsible for supplying three-phase power to each motor, as directed by the flight controller through a PWM signal.

One of the challenges in electronics is working with components that operate at different voltages. In this project, the components and their respective nominal operating voltages are as follows:

Component	Operating Voltage
Motors	16.8
Raspberry Pi Pico	5
FlySky FS-iA6B	5
MPU-6050	5

As seen in the table, apart from the motors, every electronic device operates at a standard 5 volts. This aligns with the common voltage used in consumer electronics. To address this discrepancy, many ESC manufacturers integrate a **battery eliminator circuit (BEC)** into their designs.

The BEC, a separate circuit within the ESC, provides power to other 5V components on the quadcopter. Instead of needing to

install a secondary 5V battery, we can tap into the BEC of one of the ESCs. It handles the conversion from the LiPo's 16.8V to a stable 5V, which we can utilize to power these devices.

Please note that not every ESC includes a BEC. The ESCs I purchased come with an integrated BEC, but it is essential to verify this feature when selecting ESCs for a quadcopter build.

Microcontroller

The Scout Flight Controller uses a Raspberry Pi Pico as the "brain" of the quadcopter. The Raspberry Pi Pico, priced at only $4, is an inexpensive yet capable microcontroller.

I am overclocking the RP2040's 125 MHz processor to 250 MHz during flight, providing the necessary computational power to execute the adjustment loop at 250 Hz.

Gyroscope & RC Receiver

As mentioned in previous chapters, the Scout Flight Controller continuously monitors both the actual attitude of the vehicle and the desired attitude from the pilot.

For monitoring the actual attitude, an onboard MPU-6050 gyroscope is employed. The MPU-6050 is a widely used motion-tracking sensor with 6 degrees of freedom.

To capture the desired attitude from the pilot, Scout relies on the FlySky FS-iA6B Receiver. This receiver receives RC commands from the pilot via an RC transmitter.

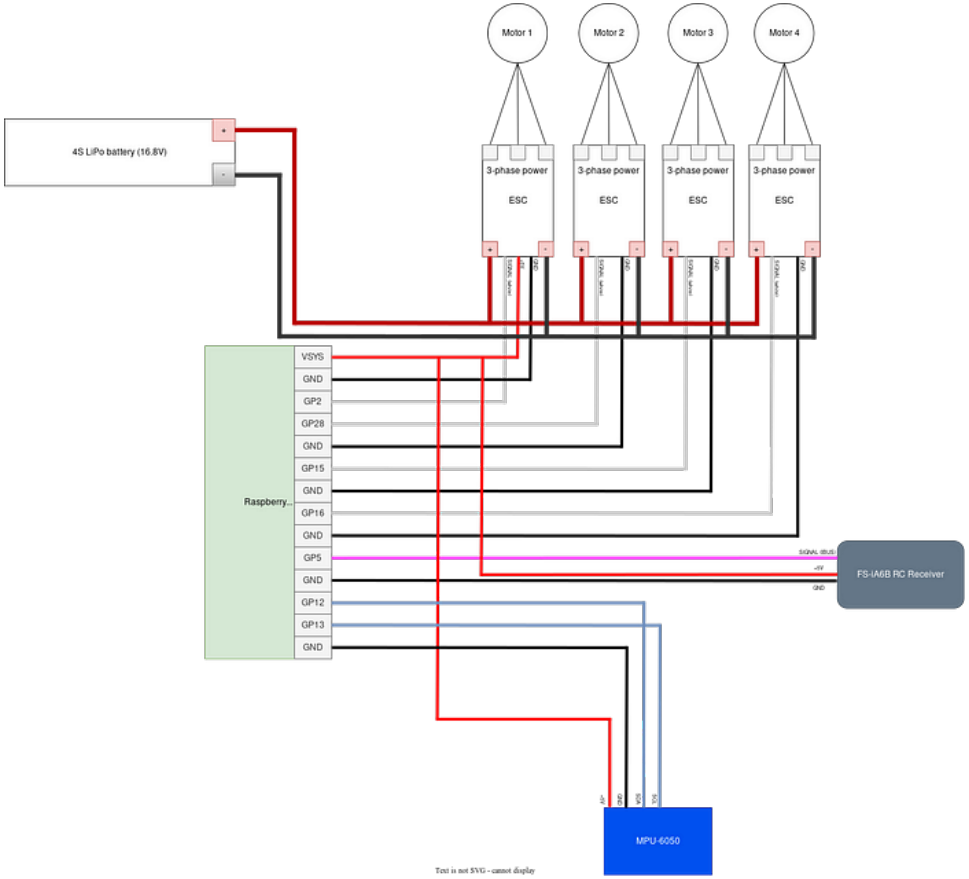
RC Transmitter

The pilot uses a FlySky FS-i6 to send control commands to the quadcopter. The only modification I made to the FS-i6 transmitter's out-of-the-box settings was changing the output of the fifth channel. This channel, one of the two auxiliary channels, was altered to output the value of **SWA** (a switch on the transmitter) instead of **VRA** (a potentiometer on the transmitter).

The position of the SWA switch is used by the Scout Flight Controller to determine the flight mode, as explained in the previous chapter on the RC receiver.

Wiring

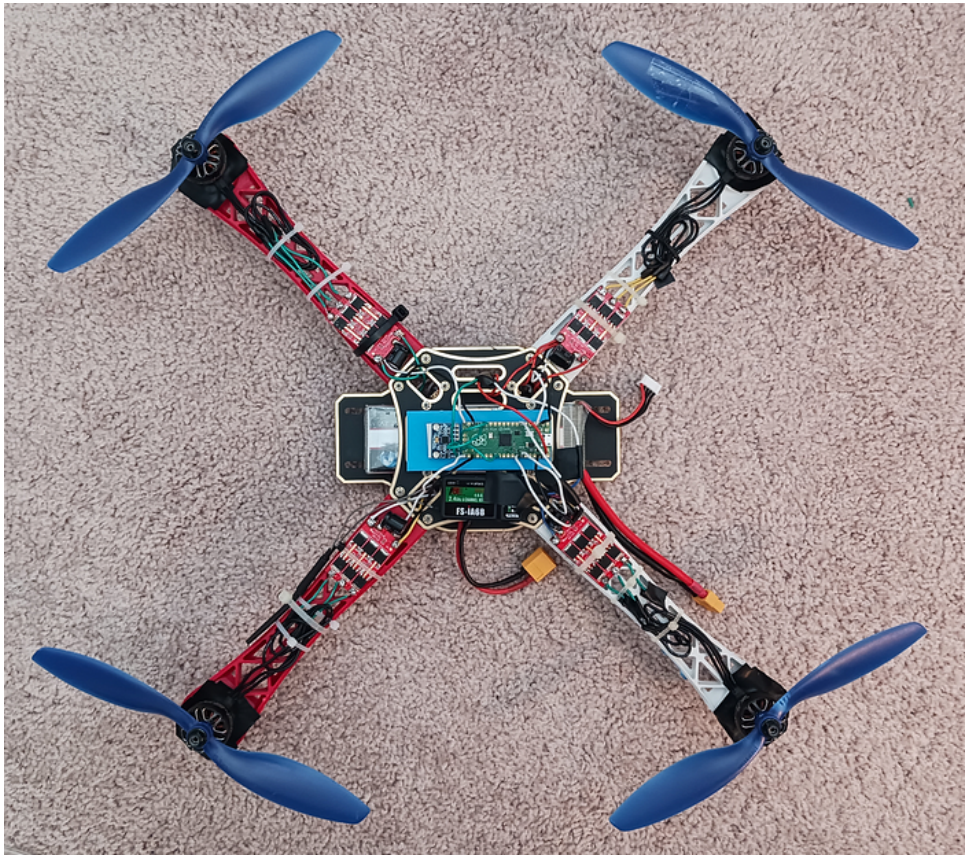
Finally, I have provided an illustration of how all these components are interconnected to form a fully functional quadcopter. Please refer to the wiring diagram below:



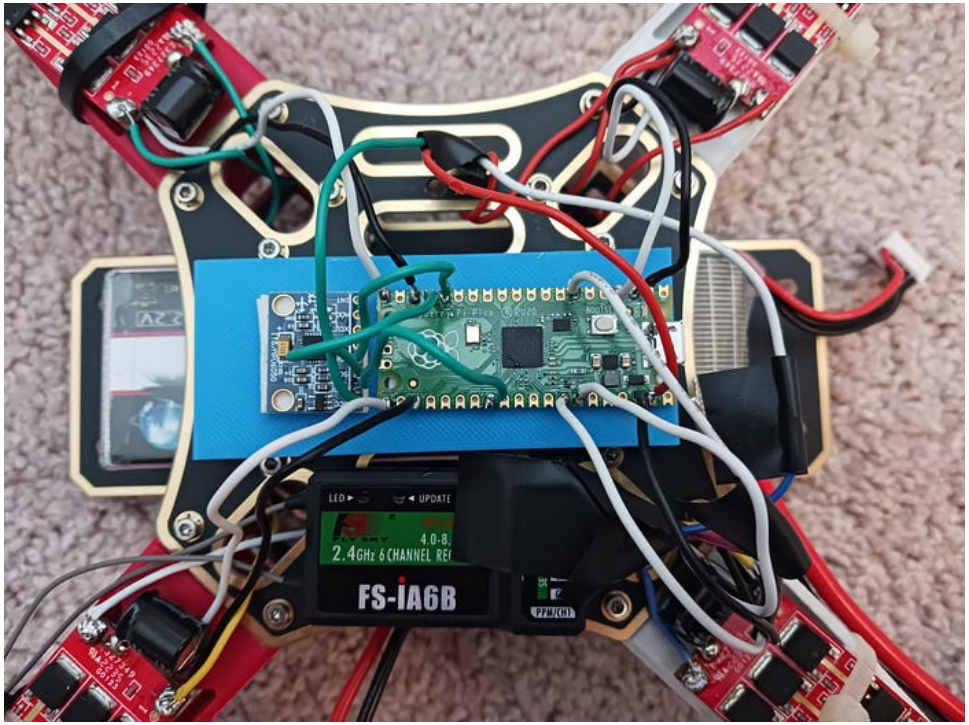
Scout Circuitry

Pictures

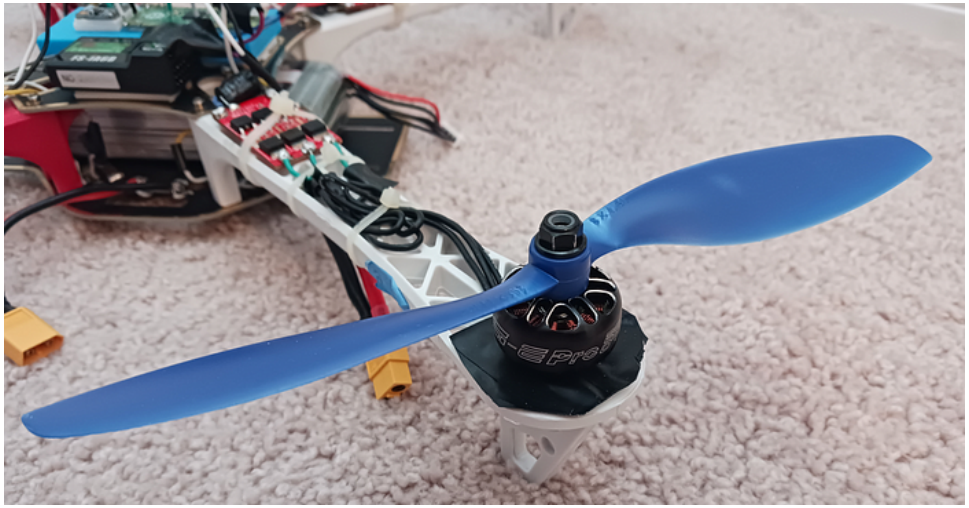
Scout uses an **F450** frame:



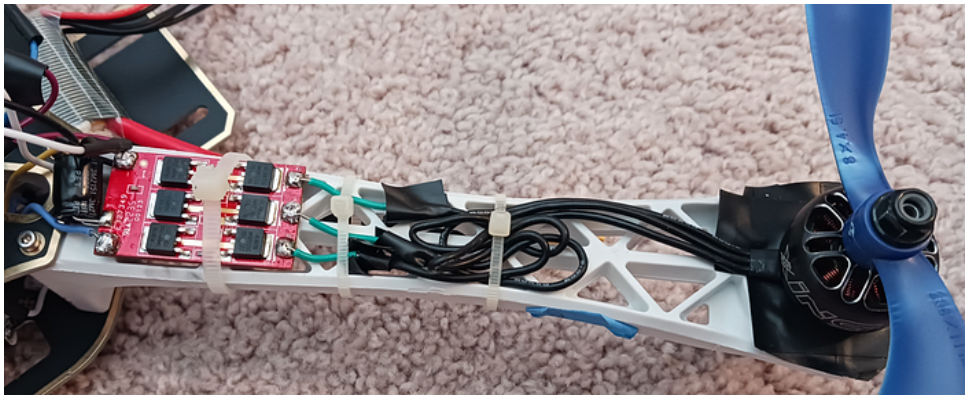
The Scout Flight Controller runs on a **Raspberry Pi Pico**, mounted on a small 3D-printed tray, attached to the frame with double sided foam tape. Adjacent to the Raspberry Pi Pico is an **FS-IA6B** receiver, allowing Scout to receive command input from the pilot via an RC transmitter. Behind the Raspberry Pi Pico is an **MPU-6050** which is used as the inertial measurement unit (IMU) that captures in-flight telemetry.



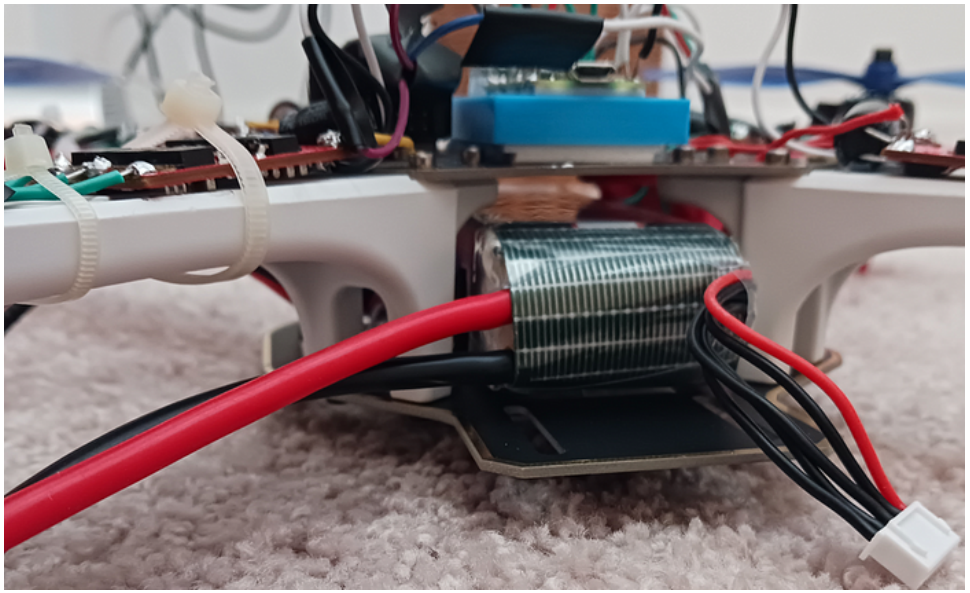
Scout uses two-blade 8-inch propellers on each of its 2750 KV motors. You may notice clear tape on several of the propellers in the image below. This is a propeller balancing measure.



An electronic speed controller (seen in red below) is mounted to each arm and is used to control that arm's respective motor via 3-phase power, a requirement for brushless motors. I have removed the ESCs from their casings and directly soldered the input and output wires to the board.

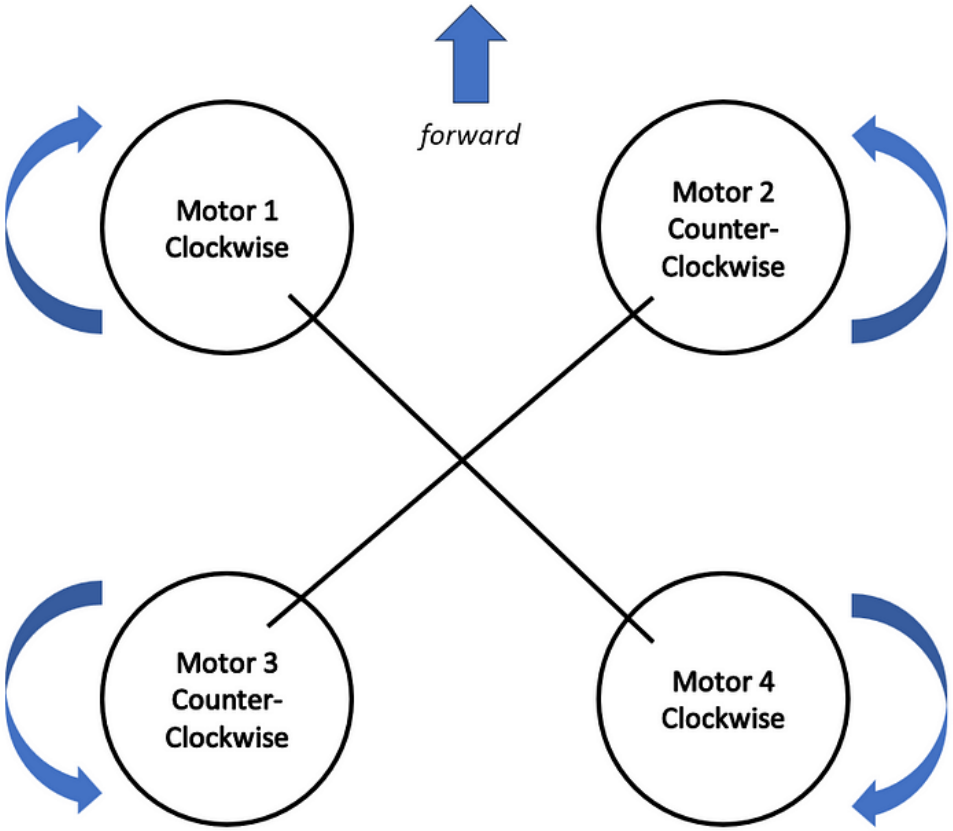


A 4S (4 cells in a **S**eries) is stored in the bottom tray of the frame.



Motor Numbering & Prop Direction

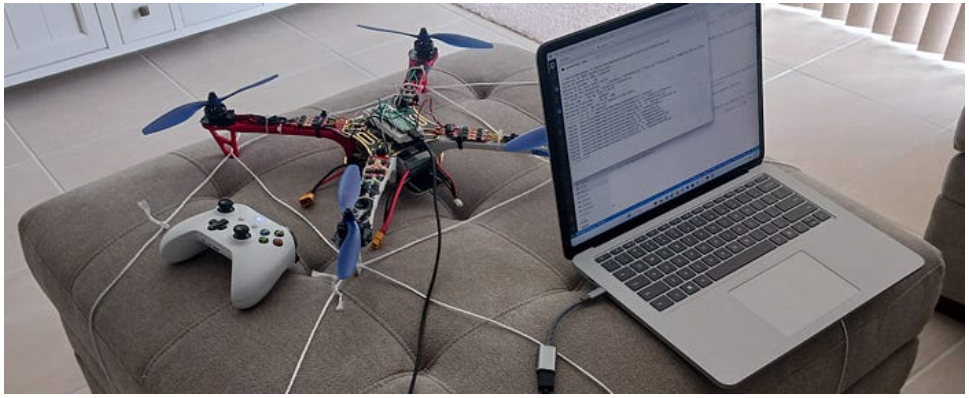
I have provided a diagram below illustrating the motor numbering scheme and the required spinning direction for each propeller. It is **extremely important** to ensure that the PWM signals are correctly mapped to the corresponding motor and that the propellers spin in the direction shown below.



What's next?

With the hardware assembled and the components properly wired together, the next step is to flash the Scout Flight Controller software onto the microcontroller. In the next chapter, we will dissect the complete code of the Scout Flight Controller, gaining a comprehensive understanding of its functionality.

Full Flight Controller Code



Upload the Scout Flight Controller code to your quadcopter microcontroller

Having now covered both the core *software* concepts that allow a quadcopter to stay airborne and achieve stable flight and the *hardware* that is required, we are now prepared to take a look at the **complete flight controller code**. This code stitches together each core concept we discussed earlier.

Full Scout Flight Controller Code

The full code behind the Scout Flight Controller can be found below. I will break down and explain each piece of the code in subsequent sections.

Design Considerations

Before we begin to dissect each piece of the code above, it is important to note several important design considerations I had to keep in mind when developing this software:

This code needs to run quickly — I mentioned in previous articles that the Scout Flight Controller runs the PID adjustment loop at 250 Hz, or 250 times per second. I learned that achieving this speed is a challenge using MicroPython on the inexpensive Raspberry Pi Pico's CPU hardware. In contrast to the *relatively* slow MicroPython, if Scout were implemented in C/C++, I'm sure the code would execute far more efficiently, possible 10x+ faster.

To achieve these speeds, you'll notice that I am not using the basic principles of programming — modularity, reusability, etc. I am keeping my usage of *classes*, *functions*, and *modules* to the bare minimum. This is because usage of these simple Python concepts actually requires a bit of extra processing time that would slow the execution of the code down enough for the 250 Hz adjustment loop to be out of reach. So, while in past articles in this series I shared class definitions for things like reading gyroscope telemetry from the MPU-6050 via I2C, I'm actually *not* using that class in my code; instead, I am extracting the logic from that class and applying it in a *procedural* manner in the flight controller code. The same goes for modules; while it would make more sense to store some functionality in different modules for the purpose of organization, this would result in minor performance losses that I simply could not afford. For this reason, the code is (mostly) optimized for *performance*, not *readability*, though I did my best to add comments in the code where appropriate and below.

I found that, even with the performance optimizations mentioned above, the code *still* runs too slowly to achieve the 250 Hz loop target. So, the Scout Flight Controller *overclocks* the Raspberry Pi Pico's RP2040 processor from 125 MHz to 250 MHz, double its rated speed. From my experience, this has not caused any adverse effects whatsoever. The power consumption, temperature, reliability, etc., still appear to be normal.

This code is self-contained. Besides the `ibus` module that is used to read and interpret signals from the onboard FlySky FS-i6 receiver that we discussed in a previous chapter on, this code can run independently of any third party libraries. The base (built-in) modules of the MicroPython implementation is all that is required.

There are two critical safety mechanisms built in. This is code that controls four spinning 8-inch propellers that spin at several thousand RPMs running on an untethered lightweight frame. This is *inherently* dangerous and poses significant risks to the safety and wellbeing of you and those around you. To prevent a simple mistake in control input from creating a catastrophic accident (like a high-throttle command being sent accidentally), I included two safety mechanisms in this code: firstly, the flight controller will cease to operate if the "flight mode" switch is in the on position as soon as the flight controller gains power and boots up. Secondly, the flight controller will cease to operate if the throttle is in any position other than 0% as soon as the flight mode is switched from *standby* to *flight*. I'll explain these two safety mechanisms in further detail below.

Settings

```
########################################
###########  SETTINGS  #################
########################################

# Motor GPIO's (not pin number, GPIO number) ###
gpio_motor1 = 2 # front left, clockwise
gpio_motor2 = 28 # front right, counter clockwise
gpio_motor3 = 15 # rear left, counter clockwise
gpio_motor4 = 16 # rear right, clockwise

# i2c pins used for MPU-6050
gpio_i2c_sda = 12
gpio_i2c_scl = 13

# RC-receiver UART
# either a 0 or 1 (Raspberry Pi Pico supports two
UART interfaces)
rc_uart = 1

# throttle settings
throttle_idle:float = 0.14 # the minumum throttle
needed to apply to the four motors for them all to
spin up, but not provide lift (idling on the
ground). the only way to find this value is through
testing (with props off).
throttle_governor:float = 0.22 # the maximum
throttle that can be applied. So, if the pilot is
inputting 100% on the controller, it will max out
at this. And every value below the pilot's 100%
will be scaled linearly within the range of the
idle throttle seen above and this governor
throttle. If you do not wish to apply a governor
(not recommended), set this to None.

# Max attitude rate of change rates (degrees per
second)
max_rate_roll:float = 30.0 # roll
max_rate_pitch:float = 30.0 # pitch
max_rate_yaw:float = 50.0 # yaw

# Desired Flight Controller Cycle time
# This is the number of times per second the flight
controller will perform an adjustment loop (PID
loop)
target_cycle_hz:float = 250.0

# PID Controller values
pid_roll_kp:float = 0.00043714285
pid_roll_ki:float = 0.00255
pid_roll_kd:float = 0.00002571429
```

```
pid_pitch_kp:float = pid_roll_kp
pid_pitch_ki:float = pid_roll_ki
pid_pitch_kd:float = pid_roll_kd
pid_yaw_kp:float = 0.001714287
pid_yaw_ki:float = 0.003428571
pid_yaw_kd:float = 0.0

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

In this first section of code, several programmable setting variables are established. These are values that the subsequent flight controller code will use but that you may want to readily change.

The first portion of this settings section is where we designate the GPIO ("general purpose input/output") that we will map each motor to. I have the signal wire of each motor directly soldered to pins 2, 28, 15, and 16 sequentially. The specific GPIO pins that are used to communicate with the MPU-6050 via I2C are also noted.

The UART protocol is used to read input commands from the onboard FlySky FS-i6 RC receiver. The Raspberry Pi Pico has two UART interfaces and the interface being used is noted here. I am using `1` because I half the FS-i6 signal wire soldered directly to the `GP5` pin of the Raspberry Pi Pico.

Next, the **idle** and **maximum** throttle in *flight mode* is defined. Here, the `throttle_idle` variable is set to the minimum percentage (14% in this case) that spins each of the four motors but does **not** produce enough lift for the quadcopter to take off. You will need to find this value through experimentation.

The `throttle_governor` variable is the *maximum* throttle percentage that should be applied to each of the four motors if the pilot is indicating 100% of the available power via the RC controller. In other words, if the pilot has the throttle stick pushed all the way forward (100% on their controller), the power output to each of the four motors will actually be limited to this governor, of 22% in this case. As mentioned in previous chapters, the motors I used are excessively powerful. To make controlling piloting the quadcopter easier, this governor is implemented.

Next, the maximum roll, pitch, and yaw rates are defined. This is the maximum rate at which the quadcopter can adjust its roll, pitch, or yaw angle. Meaning, in my case, if the pilot pushes the attitude stick all the way forward (indicating a maximum pitch forward), the quadcopter will change its attitude at a rate of 30 degrees per second. The same applies for pitch and yaw with their respective values.

Next, the target cycle time for the internal adjustment loop ("PID loop" as discussed in previous chapters). As mentioned, the Scout Flight Controller targets an adjustment rate of 250 times per second.

Finally, the PID controller parameters are defined. As mentioned previously, each PID controller for each of the three axes requires three variables that control its behavior — a proportional gain, integral gain, and derivative gain. These three variables determine how much influence each controller has on the aggregate output of the PID controller.

**This is not a "one size fits all" approach — the appropriate PID controller gains vary from one quadcopter to another. The values

need to be tuned to accommodate the differences in weight, weight distribution, size, motor power, battery power, etc.

For determining the PID values for your quadcopter, I recommend using the *proportions* of the PID gains I have above to start your tuning process:

1. Take the props off
2. Start by setting each of the PID gains to 50% of the value I have above. Run this in flight mode and ensure the motors do not behave too erratically.
3. Put the props on. Set each PID gain to 25% of the values I have above (proportionally). Run the quadcopter in flight mode and move it around in your hand (be careful of the props!). You should feel the quadcopter fight to stay level.
4. Increase to 50%. Validate.
5. Increase to 75%. Validate.
6. Increase to 100%, or whatever percentage works best for your quadcopter. You may need a PID gain settings well beyond 100% of what I have above, based on the unique attributes of your quadcopter.

When PID tuning, please be **extremely** careful. Only increase the PID gains gradually and test frequently as you do. Placing only *one* of the PID gain settings too high can result in catastrophe as the motors spin out of control.

Startup Routine

The full flight controller program, including the PID loop, is enclosed within the `run` function. The `run` functions starts with several setups:

1. The Raspberry Pi Pico's onboard LED is flashed several time to indicate to the user that the microcontroller has been powered on and the Scout Flight Controller is running.
2. The CPU is overclocked to 250 MHz.
3. An instance of the `IBus` class is created; this will later be used to receive input commands from the pilot.

Safety Mechanism #1: Flight Mode Switch Position

As part of the program boot phase (before the PID loop), a critical safety check is performed:

```
print("Validating that mode switch is not in flight
position...")
for x in range(0, 60):
    rc_data = rc.read()
    if rc_data[5] == 2000: # flight mode on
        FATAL_ERROR("Flight mode detected as on
(from RC transmitter) as soon as power was
received. As a safety precaution, mode switch needs
to be in standby mode when system is powered up.")
    time.sleep(0.025)
```

The value of the fifth channel from the RC receiver is checked sixty times. If it is noticed that the switch is in the "on" position (value of 2000 as opposed to a value of 1000, indicating an "off" position), the program is aborted. An error message is logged to

the device's flash memory and an infinite loop pulsates the onboard LED in a known pattern indicating this.

This is an *important* safety mechanism. The serves to prevent the quadcopter from instantly spinning up its motors when the pilot may not be expecting it (the channel 5 switch being in the "on" position indicates the quadcopter should be in *flight mode*, at the very least idling the propellers).

MPU-6050 Initialization

The MPU-6050 is a *critical* component of a quadcopter flight controller that **must** function properly to achieve any semblance of stable, predictable flight. In a previous chapter on the gyroscope, I provided a lightweight class that can be used to interface with an onboard MPU-6050 accelerometer & gyroscope. For the sake of code performance, the Scout Flight Controller does *not* use this class. It instead communicates with the MPU-6050 procedurally.

The MPU-6050 is set up like so:

```
# Set up IMU (MPU-6050)
i2c = machine.I2C(0, sda =
machine.Pin(gpio_i2c_sda), scl =
machine.Pin(gpio_i2c_scl))
mpu6050_address:int = 0x68
i2c.writeto_mem(mpu6050_address, 0x6B,
bytes([0x01])) # wake it up
i2c.writeto_mem(mpu6050_address, 0x1A,
bytes([0x05])) # set low pass filter to 5 (0-6)
i2c.writeto_mem(mpu6050_address, 0x1B,
bytes([0x08])) # set gyro scale to 1 (0-3)
```

In the above code, the I2C interface is set up that will be used to communicate with the MPU-6050 via the I2C protocol. Three messages are sent to the MPU-6050 to configure settings:

1. `0x01` is written to the `0x6B` register, waking the MPU-6050 up from a low-power mode.
2. `0x05` is written to the `0x1A` register, turning the MPU-6050's onboard low pass filter to level 5.
3. `0x08` is written to the `0x1B` register, setting the gyro scale factor to a level of 1.

Several tests are then conducted to validate that the MPU-6050 responded to the write commands above and is working as expected:

1. One byte is read from register `0x75`, the "WHOAMI" register. The MPU-6050 should always return 0x68, or 104, the I2C address that the manufacturer sets. This is a simple way of confirming that bidirectional communication is working.
2. The low pass filter value is validated to be `0x05` (level 5) in the `0x1A` register.
3. The gyro scale factor is validated to be `0x05` in the `0x1B` register.

If any of these three tests fail, an error message is logged and the execution of the flight controller is aborted.

Gyro Calibration

In a previous chapter on the gyroscope, I mentioned that every gyroscope has some level of bias built in to its readings. Due to various factors, a gyroscopes values will never read perfectly 0.0, 0.0, 0.0 when sitting perfectly still (0 rotation movement along each axis). Instead, there will always be some small reading that persists consistently. To account for this bias, we must run a calibration in which we calculate the average reading in each of the three axes for a period of several seconds:

```
# measure gyro bias
print("Measuring gyro bias...")
gxs:list[float] = []
gys:list[float] = []
gzs:list[float] = []
started_at_ticks_ms:int = time.ticks_ms()
while ((time.ticks_ms() - started_at_ticks_ms) /
1000) < 3.0:
    gyro_data = i2c.readfrom_mem(mpu6050_address,
0x43, 6) # read 6 bytes (2 for each axis)
    gyro_x = (translate_pair(gyro_data[0],
gyro_data[1]) / 65.5)
    gyro_y = (translate_pair(gyro_data[2],
gyro_data[3]) / 65.5)
    gyro_z = (translate_pair(gyro_data[4],
gyro_data[5]) / 65.5) * -1 # multiply by -1 because
of the way I have it mounted on the quadcopter - it
may be upside down. I want a "yaw to the right" to
be positive and a "yaw to the left" to be negative.
    gxs.append(gyro_x)
    gys.append(gyro_y)
    gzs.append(gyro_z)
    time.sleep(0.025)
gyro_bias_x = sum(gxs) / len(gxs)
gyro_bias_y = sum(gys) / len(gys)
gyro_bias_z = sum(gzs) / len(gzs)
print("Gyro bias: " + str((gyro_bias_x,
gyro_bias_y, gyro_bias_z)))
```

The variables `gyro_bias_x`, `gyro_bias_y`, and `gyro_bias_z` will be subtracted from every subsequent gyroscope reading. The resulting measurement is the *calibrated* reading that has been cleansed of bias and can be used as an accurate reading to be used by the quadcopter flight controller.

PID State Variables

As mentioned in the previous chapter on PID Controllers, the **integral** and **derivative** terms of a PID controller require state variables — in each loop, these terms build upon values from the previous loop. Thus, there are variables that must be declared *outside* of the infinite adjustment loop.

```
# State variables - PID related
# required to be delcared outside of the loop
because their state will be used in multiple loops
(passed from loop to loop)
roll_last_integral:float = 0.0
roll_last_error:float = 0.0
pitch_last_integral:float = 0.0
pitch_last_error:float = 0.0
yaw_last_integral:float = 0.0
yaw_last_error:float = 0.0
```

The PID Loop Begins: Telemetry Collection

Next, the infinite `while` loop begins. This loop will continuously run until the Scout Flight Controller loses power supply. As mentioned in previous chapters, a quadcopter flight controller *must* know two things - its current *actual* attitude rate of change the pilot's *desired* attitude rate of change. This is accomplished in the following snippets of code:

Capturing gyroscope telemetry from the onboard MPU-6050:

```
# Capture raw IMU data
# we divide by 65.5 here because that is the
modifier to use at a gyro range scale of 1, which
we are using.
gyro_data = i2c.readfrom_mem(mpu6050_address, 0x43,
6) # read 6 bytes (2 for each axis)
gyro_x = ((translate_pair(gyro_data[0],
gyro_data[1]) / 65.5) - gyro_bias_x) * -1 # Roll
rate. we multiply by -1 here because of the way I
have it mounted. it should be rotated 180 degrees I
believe, but it's okay, I can flip it here.
gyro_y = (translate_pair(gyro_data[2],
gyro_data[3]) / 65.5) - gyro_bias_y # Pitch rate.
gyro_z = ((translate_pair(gyro_data[4],
gyro_data[5]) / 65.5) * -1) - gyro_bias_z # Yaw
rate. multiply by -1 because of the way I have it
mounted - it may be upside down. I want a "yaw to
the right" to be positive and a "yaw to the left"
to be negative.
```

Capturing control inputs from the pilot via the onboard RC receiver:

```
# Read control commands from RC
rc_data = rc.read()

# normalize all RC input values
input_throttle:float = normalize(rc_data[3],
1000.0, 2000.0, 0.0, 1.0) # between 0.0 and 1.0
input_pitch:float = (normalize(rc_data[2], 1000.0,
2000.0, -1.0, 1.0)) * -1 # between -1.0 and 1.0. We
multiply by -1 because... If the pitch is "full
forward" (i.e. 75), that means we want a NEGATIVE
pitch (when a plane pitches it's nose down, that is
negative, not positive. And when a place pitches
it's nose up, pulling back on the stick, it's
positive, not negative.) Thus, we need to flip it.
input_roll:float = normalize(rc_data[1], 1000.0,
2000.0, -1.0, 1.0) # between -1.0 and 1.0
input_yaw:float = normalize(rc_data[4], 1000.0,
2000.0, -1.0, 1.0) # between -1.0 and 1.0
```

Please note that in the above snippet, the received RC values are being *normalized*. The RC receiver typically provides a value of between 1,000 (minimum) and 2,000 (maximum) for each of the receiver's six channels. Instead of working with this range, I'd prefer to work with the more intuitive and easier to understand standard ranges of -1.0 to 1.0 for pitch, roll, and yaw, and 0.0 to 1.0 for throttle input. The `normalize` function I wrote, found at the end of this code file, performs this linear normalization to within the aforementioned desired ranges.

If in Standby Mode

The Scout Flight Controller code then splits into two directions — depending on what "mode" is active, the flight controller performs different actions. The Scout Flight Controller has two modes:

- **Standby** — The Scout Flight Controller is *standing by*, awaiting further instruction. The motors are not turning. The quadcopter is sitting perfectly still.

- **Flight** — The Scout Flight Controller is prepared to fly. At a minimum, the motors are spun up to their *idle* speed — the minimum motor power that will *spin* each motor, but will not provide enough thrust to take off. The flight controller is considering the inputs of the pilot and applying these inputs to the PID controller loop, resulting in aerobatic maneuvers.

The Scout Flight Controller handles the **Standby mode** condition first:

```
# turn motors off completely
duty_0_percent:int = calculate_duty_cycle(0.0)
M1.duty_ns(duty_0_percent)
M2.duty_ns(duty_0_percent)
M3.duty_ns(duty_0_percent)
M4.duty_ns(duty_0_percent)

# reset PID's
roll_last_integral = 0.0
roll_last_error = 0.0
pitch_last_integral = 0.0
pitch_last_error = 0.0
yaw_last_integral = 0.0
yaw_last_error = 0.0

# set last mode
last_mode = False # False means standby mode
```

Firstly, the power to all four motors is set to 0.0% (the motors are not spinning whatsoever). Secondly, the aforementioned PID state variables are reset; this ensures that any "leftover" PID state from the previous flight will not be carried over into the next. Finally, the `last_mode` variable is set to `False`, indicating the most recent mode the Scout Flight Controller was in was **Standby mode**; this will be important for a critical safety check that is performed as soon as Flight mode is activated.

If in Flight Mode

If the Scout Flight Controller is in **Flight mode**, a very different set of steps occurs. Ultimately, this is the mode in which the PID loop occurs — where flight happens!

Firstly, a **critical** safety check occurs:

```
# if last mode was standby (we JUST were turned
onto flight mode), perform a check that the
throttle isn't high. This is a safety mechanism
# this prevents an accident where the flight mode
switch is turned on but the throttle position is
high, which would immediately apply heavy throttle
to each motor, shooting it into the air.
if last_mode == False: # last mode we were in was
standby mode. So, this is the first frame we are
going into flight mode
    if input_throttle > 0.05: # if throttle is > 5%
        FATAL_ERROR("Throttle was set to " +
str(input_throttle) + " as soon as flight mode was
entered. Throttle must be at 0% when flight mode
begins (safety check).")
```

If the most recent last mode the Scout Flight Controller was in was *Standby mode*, this means that **Flight mode** was *just now* activated. Before continuing to apply power to the motors and apply the pilot's input controls, the flight controller checks the position of the **input throttle**. If the input throttle is set to anything beyond 5% throttle, the Scout Flight Controller aborts the program and enters a failsafe mode, pulsating the onboard LED indicating it has aborted.

The reason for this safety check is to mitigate the risk of an extremely dangerous accident from occurring: the pilot flips into **Flight mode** with the throttle stick in a high position. Without this safety check, a large amount of power will instantly be applied to each of the four motors, causing the quadcopter to instantly fly into the air with tremendous power. This simple accident can cause major bodily injury and harm. This simple check ensures that this mistake doesn't happen. The pilot *must* remember to place the throttle stick in the minimum

position before activating **Flight mode**; if he doesn't the program aborts!

After the safety check occurs, the adjusted throttle, `adj_throttle`, is calcualted. This the throttle level that will be applied to the PID loop, and thus, applied to each of the four motors. It is scaled to fit within the confines of the `throttle_idle` (minimum throttle required to spin up each motor at a minimum RPM) and `max_throttle` (throttle governor).

```
# calculate the adjusted desired throttle (above
idle throttle, below governor throttle, scaled
linearly)
adj_throttle:float = throttle_idle +
(throttle_range * input_throttle)
```

Next, the *errors* required for each of the three PID controllers (one per axis) are calculated. As mentioned in previous chapters, the *error* is simply the difference between the pilot's *desired* attitude rate of change in that axis and the *actual* attitude rate of change in that axis.

```
# calculate errors - diff between the actual rates
and the desired rates
# "error" is calculated as setpoint (the goal) -
actual
error_rate_roll:float = (input_roll *
max_rate_roll) - gyro_x
error_rate_pitch:float = (input_pitch *
max_rate_pitch) - gyro_y
error_rate_yaw:float = (input_yaw * max_rate_yaw) -
gyro_z
```

These calculated *errors* are then used in the PID Controller calculations:

```
# roll PID calc
roll_p:float = error_rate_roll * pid_roll_kp
roll_i:float = roll_last_integral +
(error_rate_roll * pid_roll_ki *
cycle_time_seconds)
roll_i = max(min(roll_i, i_limit), -i_limit) #
constrain within I-term limits
roll_d:float = pid_roll_kd * (error_rate_roll -
roll_last_error) / cycle_time_seconds
pid_roll:float = roll_p + roll_i + roll_d

# pitch PID calc
pitch_p:float = error_rate_pitch * pid_pitch_kp
pitch_i:float = pitch_last_integral +
(error_rate_pitch * pid_pitch_ki *
cycle_time_seconds)
pitch_i = max(min(pitch_i, i_limit), -i_limit) #
constrain within I-term limits
pitch_d:float = pid_pitch_kd * (error_rate_pitch -
pitch_last_error) / cycle_time_seconds
pid_pitch = pitch_p + pitch_i + pitch_d

# yaw PID calc
yaw_p:float = error_rate_yaw * pid_yaw_kp
yaw_i:float = yaw_last_integral + (error_rate_yaw *
pid_yaw_ki * cycle_time_seconds)
yaw_i = max(min(yaw_i, i_limit), -i_limit) #
constrain within I-term limits
yaw_d:float = pid_yaw_kd * (error_rate_yaw -
yaw_last_error) / cycle_time_seconds
pid_yaw = yaw_p + yaw_i + yaw_d
```

The resulting calculations of each of the three PID Controllers — `pid_pitch`, `pid_roll`, and `pid_yaw`, are then "mixed" together with the applied throttle (`adj_throttle`) to determine specifically how much power to apply to each of the four motors:

```
# calculate throttle values
t1:float = adj_throttle + pid_pitch + pid_roll -
pid_yaw
t2:float = adj_throttle + pid_pitch - pid_roll +
pid_yaw
t3:float = adj_throttle - pid_pitch + pid_roll +
pid_yaw
t4:float = adj_throttle - pid_pitch - pid_roll -
pid_yaw
```

Each of the four values above — `t1`, `t2`, `t3`, and `t4`, is a percent value between 0.0 and 1.0; what "percent" of throttle to apply to that specific motor. This signal is then sent to each motor via a **PWM Duty Cycle**:

```
# Adjust throttle according to input
M1.duty_ns(calculate_duty_cycle(t1))
M2.duty_ns(calculate_duty_cycle(t2))
M3.duty_ns(calculate_duty_cycle(t3))
M4.duty_ns(calculate_duty_cycle(t4))
```

Finally, the PID loop state variables are saved — remember, we need to do this because the error and integral calculations of *this* loop will be needed for *next* loop!

```
# Save state values for next loop
roll_last_error = error_rate_roll
pitch_last_error = error_rate_pitch
yaw_last_error = error_rate_yaw
roll_last_integral = roll_i
pitch_last_integral = pitch_i
yaw_last_integral = yaw_i
```

Target Cycle Time (250 Hz)

For the quadcopter to achieve stable flight, this PID adjustment loop must occur consistently at a uniform interval. In the Scout Flight Controller's case, this loop is to occur at 250 Hz, or 250 times per second, or at 4 millisecond intervals. To ensure the PID loop does not occur any faster than this, the Scout Flight Controller waits at the end of the `while` loop until the 4 milliseconds (4,000 microseconds) period for the current loop is finished. This ensures that any excess time is burnt here before continuing on with the next loop.

```
# mark end time
loop_end_us:int = time.ticks_us()

# wait to make the hz correct
elapsed_us:int = loop_end_us - loop_begin_us
if elapsed_us < cycle_time_us:
    time.sleep_us(cycle_time_us - elapsed_us)
```

In Conclusion

And that's it! Apart from some simple utility functions you can find below the main `run` function we just dissected, that is the entire **Scout Flight Controller** code. This code merges all the knowledge from preceding chapters, culminating in the creation of a robust flight controller that ensures stable flight.

In the next chapter, I'll explain how to safely and effectively take your quadcopter for its first flight!

Taking Flight

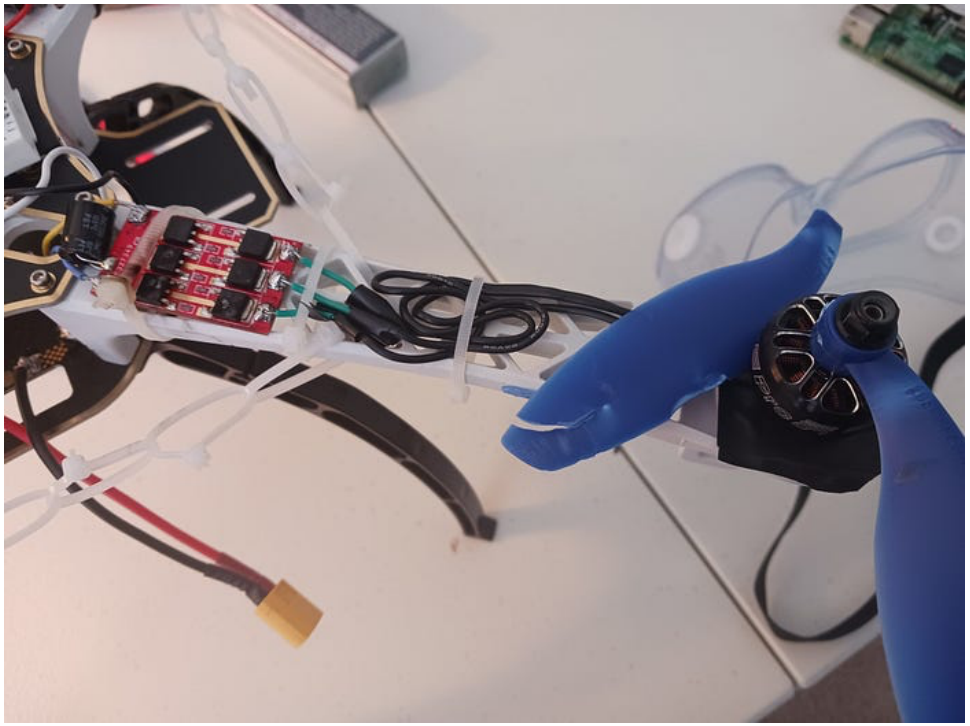
Now with the hardware assembled and Scout Flight Controller software uploaded, the quadcopter is now prepared to be powered up and tested.

Safety First

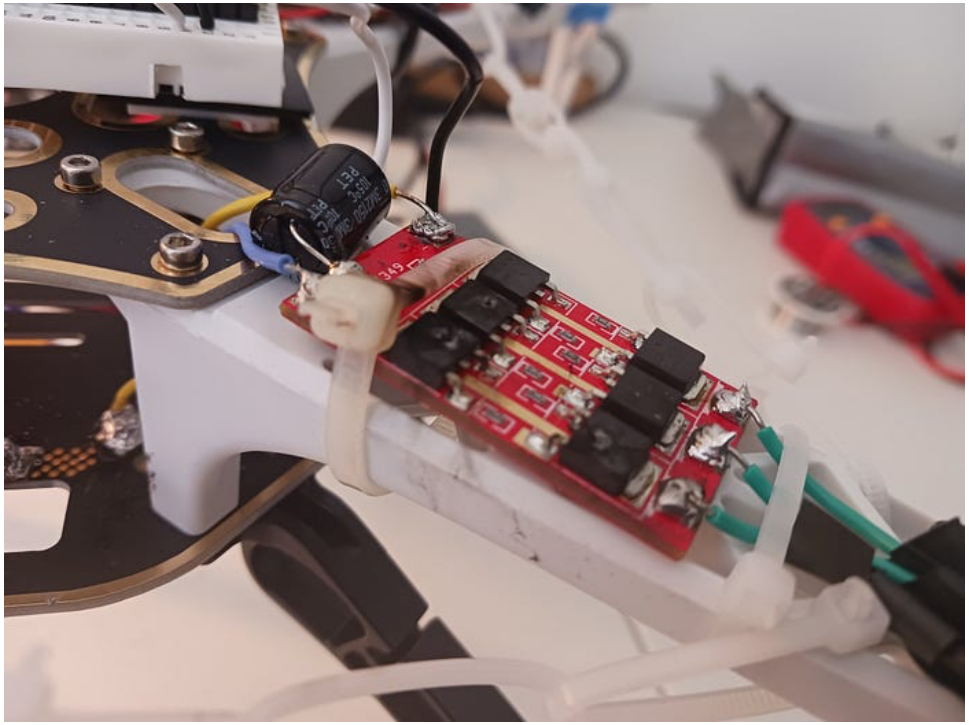
Before doing anything, **take the necessary safety precautions when performing tests**! This is a highly-powered quadcopter with sharp-ended propellers that spin at several thousand revolutions per minute. Any slight error or mistake can have *very* serious consequences. I will do my best to share what I learned while building *Scout*. I am not responsible for any physical harm or accidents.

Use safety goggles — By far, the number one thing you need to protect is your eyes. I had a close call in which the propellers came loose and nearly hit collided with my face or when the quadcopter lost control while in close proximity to my face. Eye protection is **CRITICAL**! Whenever you are powering up the quadcopter, for any reason, please use safety goggles.

Have a fire extinguisher handy — When using electrical components in a high-performance scenario like we are, electrical fires can happen. I had three incidents in which a small fire *nearly* started. On one occasion, an error in code caused too much amperage to travel through a positive and negative wire for one of the motors. The amperage must have been higher than the wires I was using were rated for because the two wires melted their casing and fused together. Clearly a catastrophic problem. Luckily I was able to unplug the battery before anything serious happened besides smoke and a small flame. On the other two occasions, and ESC burned out when the drone crashed. Simple things like this happen so it is best to take precautions. I purchased a small, inexpensive fire extinguisher and never powered up the quadcopter without this by my side.



A propeller snapped after a crash



An ESC burned out mid-flight

Wear gloves — The propellers have nicked me and caused deep cuts many times. In several of these scenarios, the cuts were so deep that they have already formed scars. I've learned that the worst place to have a deep cut from a propeller is on your hands. When powering the quadcopter on and handling it while powered (for testing purposes), wear gloves to provide at least some level of protection.

Make sure the propellers are tightened — When it comes time to power up the motors with propellers on, *always* ensure that each of the four propellers are sufficiently tight. I made the mistake of not tightening them enough during my early test flights. I quickly learned that not tightening the propellers enough will cause each of the four bolts to fly into the air and each propeller to release and act as a flying spinning blade with a random trajectory. Even losing a single propeller during a flight would be catastrophic. Ensure that your propellers are very tight. Retighten every 3–4 flights.

Test Without Propellers

Before fastening the propellers to each of the four motors, it is wise to perform a simple test first *without* propellers. Power the quadcopter up, switch into flight mode, and you should see each of the four motors spool up in their appropriate direction. Gradually apply a bit more throttle and you should see the motors spin faster. Input some basic commands (pitch forward, roll right, yaw, etc), and you should at least *hear* the motors increasing/decreasing speed to accommodate your commands.

This simple test ensures that your motors are working correctly *before* you introduce a safety risk by fastening propellers. Whenever you are unsure about some component of the drone (i.e. new software update, it just crashed, an ESC may not be working, etc.), first test with *no propellers* as a simple (relatively) risk-free assessment.

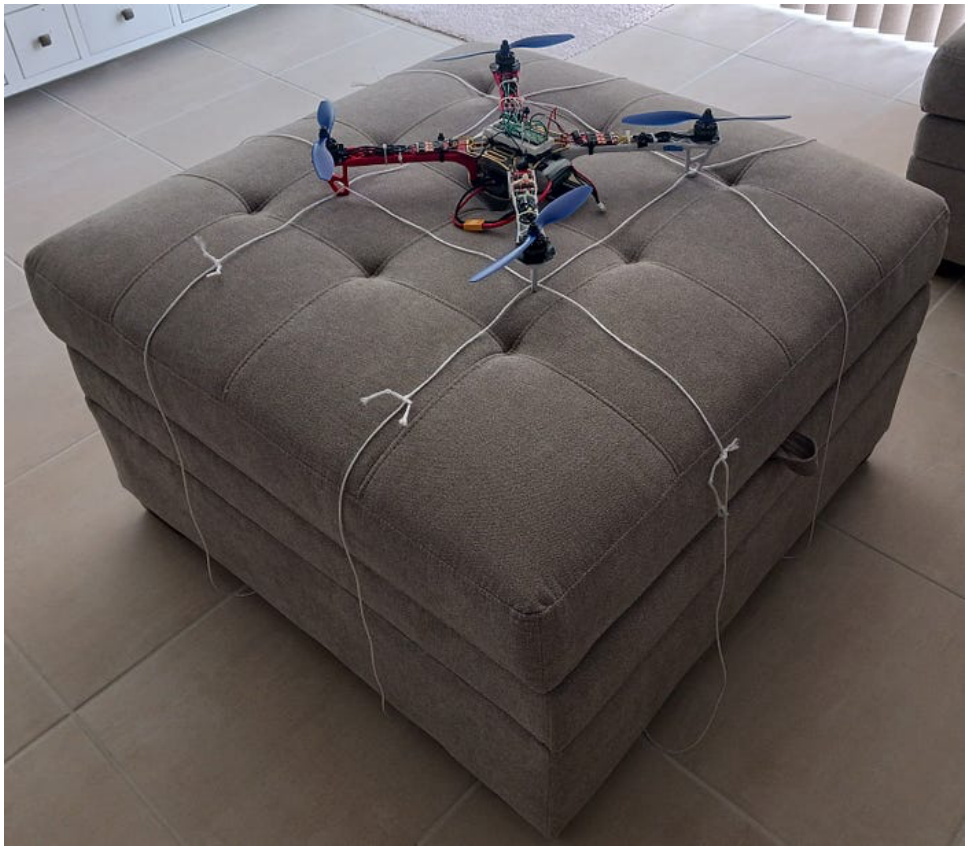
Always Tether

Until you have a certain number of hours of flight time under your belt and you are certain that the drone flies predicably and stably, always **tether** the drone to something heavy on the ground.

As mentioned in the previous article, you will need to find the approximate PID (proportional, integral, derivative) controller gain values that work best for your particular quadcopter. While testing this and handling the quadcopter yourself while in flight mode (manipulating its attitude manually to observe how it

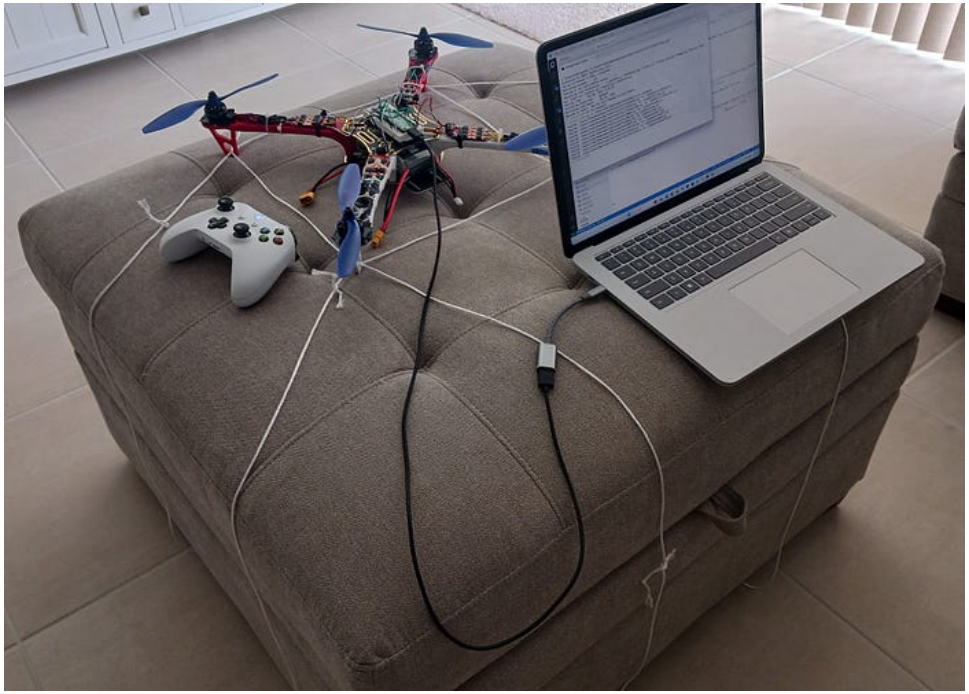
responds), tether it to the ground so **just in case**, if something goes wrong, it is not able to fly uncontrolled and damage something.

During development, I would tether *Scout* to my ottoman. It isn't high-tech, but it works.



I looped four strings around the ottoman and through each mounting hole, below each motor, of the drone's frame. The four strings give about 1.5 feet of free play but arrest any further movement along each axis (for the most part).

I used this set up during development. I would upload code, test the result, make a tweak to the code, and repeat this process until it was tuned properly.



After tweaking the code with the ottoman setup and feeling confident in the flight controller's tuned parameters, I then tethered the quadcopter to the ground with several heavy pots and pans. I positioned each heavy pot/pan so the drone had only about 4–6 feet to fly.

Scout achieved its first flight tethered as specified above:

After ensuring *Scout* can fly predictably with it tethered like this, I then took it outside and opted for a much longer, single string leash. I cut a string of about 50 feet (grants a leash of 25 feet) and tied this through the frame of the drone and then through the handle of a very heavy pot. The pot, upside down, served as the "launch pad" for the quadcopter (it would sit on this when taking off to avoid the propellers hitting the grass). After functioning as the launch pad, it would then serve in flight as an anchor. If the drone lost control, it wouldn't be allowed to fly off as this very heavy pot would hold it down.

You can see clearly see the white string that is serving as a safety tether in Scout's demo flight video:

While testing your drone's flight capabilities early in development, it is important to tether it to the ground! If something goes wrong, this ensures that it can't fly off into the distance and potentially seriously hurt someone!

Airspace Restrictions

Drones like the ones built by DJI often come with sophisticated flight *computer* features. These features are there to aid the pilot in various ways. One feature that these drones come with is a GPS system paired with an airspace restriction map. Knowing its location via GPS and knowing the restricted air spaces in the area, DJI's software *will not* let you fly in restricted air spaces over a certain altitude.

My barebones and homemade *Scout Flight Controller* comes with no such luxury. There is no GPS unit on board and no knowledge of restricted air space. As such, there is no software-based restriction on where the drone can/cannot fly. This puts the burden on you as a pilot to be aware of federal, state, and local airspace restrictions in the areas in which you live and plan to fly.

The responsibilities of piloting a drone to not end there — it is your responsibility to be aware of airspace restrictions, operating rules, drone related laws (i.e. do not fly over people/cars), privacy laws, fly zones, etc.

No Battery Indicator

I did not install a battery level indicator (voltage monitor) onboard Scout. This means that the pilot has no way of knowing the battery level while in flight. Be careful of this — I've learned that you will generally get a warning as the voltage dips to very low levels (the quadcopter begins to feel sluggish/needs more throttle input), but power will rather abruptly be lost when the battery is absolutely dead.

It is generally bad to run a battery down to this level though. To avoid any accidents, frequently change batteries and don't fly for too long on a single battery!

FAA Registration

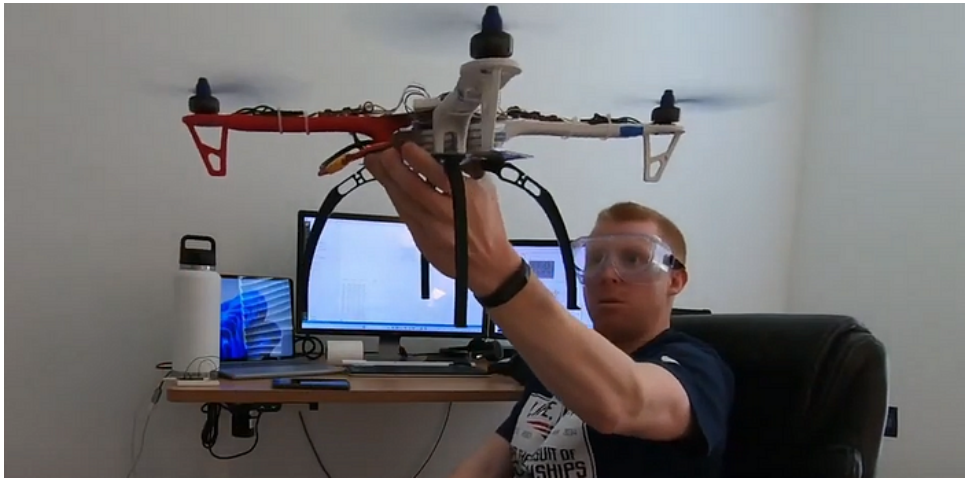
The Federal Aviation Administration (FAA) requires all drones that exceed 250 grams to be registered. If your quadcopter meets or exceeds 250 grams, you legally must register it with the FAA.

Conclusion

With the hardware assembled, *Scout Flight Controller* software uploaded, and functionality tested, the quadcopter is now complete! While this and the last few chapters walked through the development process linearly, it likely won't be as easy as this series leads you to believe. Like with all software/hardware development, there will be challenges. There will be obstacles to overcome.

In the next chapter I'll share some of the challenges I ran into and had to overcome to complete my most demanding project yet.

A Lesson in Persistence



In the previous articles in this series on how I developed a custom quadcopter flight controller from scratch. While my explanations were direct, to the point, and simplistic, getting to this point was anything **but** simple. This was by far **the hardest engineering challenge I have faced to date** — software engineering, electrical engineering, and aviation engineering. This project forced me to use all of the skills I have been honing for the past several years and then pick up many news ones.

I'm thankful for this challenge as I learned a *tremendous* amount throughout this process — a lot of which I will continue to use in my future projects. However, there were many times I doubted I would ever be able to finish this project successfully during its duration.

I would **never** call this an easy task, and this video shows why:

Scout's Development

I started developing the Scout Flight Controller on April 29, 2023. I worked diligently on this project, starting from scratch, and built from ground up until, 70 days later, Scout took its first successful flight:

In the 70 days of development, I made 1,194 commits (code changes/updates/tweaks) to the Scout software code repository. That is an average of 17 commits per day!

During that process, I completely re-wrote the Scout flight controller software *from scratch* 3 times! During its development, I constantly asked myself the question if I was writing the Scout flight controller software on the wrong platform. Is something as inexpensive and underpowered as the Raspberry Pi Pico, with its 133 MHz processor, sufficient for running a PID controller adjustment loop at 250 times per second? Do I need the "luxury" of an operating system like Linux for multi-threading? If so, I should instead be building this on a full microcomputer like the Raspberry Pi Zero W.

I would flip back and forth between targeting development for the Raspberry Pi Pico Microcontroller (written in MicroPython) and the Raspberry Pi Microcomputer (written in Python). While the two are obviously very similar, there are major differences in implementation logic that must be addressed. Additionally, the entire architecture changes as you gain/lose the benefit of things like multi-threading.

Fortunately, I was able to squeeze enough performance out of the Raspberry Pi Pico and architect the code in a way that handle all required functions of a flight controller.

A Lesson in Persistence

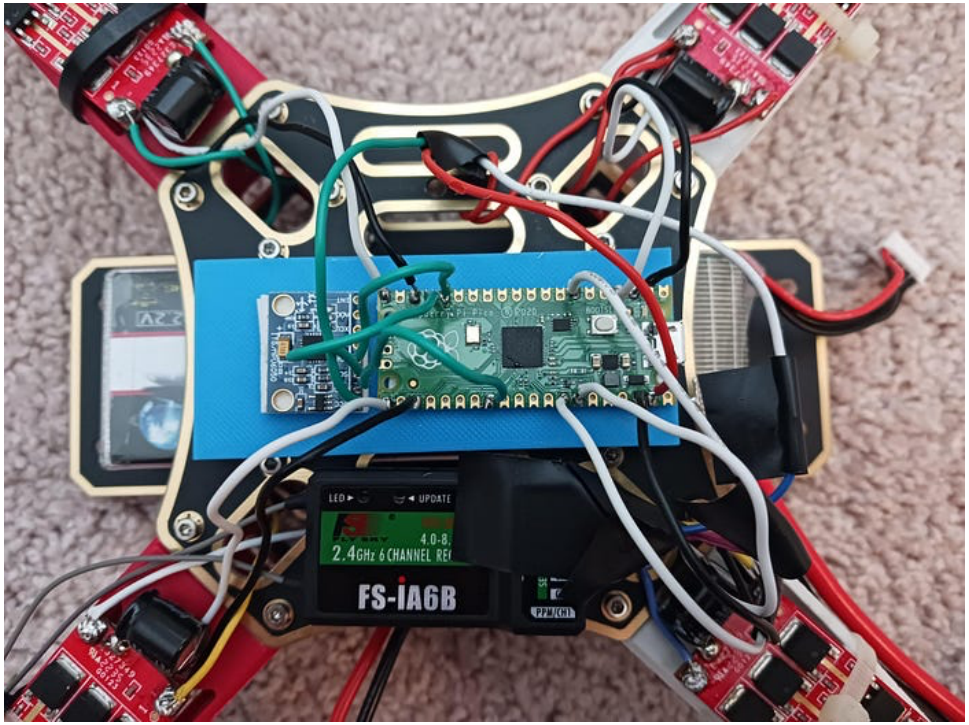
Despite all the challenges, setbacks, headaches, doubts, and more, I'm proud and thankful that I continued with this project. Please do not, in any way, think that this was an easy or simple feat. This took me *a lot* of time to figure out. If you decide to use this series (and others) as a tutorial/guide for building your own drone or flight controller software, be patient with yourself. I hope that, by following the steps and code I provide in this guide, you're able to fair much better than I did.

But if you aren't, be patient with yourself. You'll learn so much during this project — the process is just as valuable as the destination! Afterall, if the destination of having a functioning drone is actually more important to you, just buy one from DJI.

What's next?

I'm incredibly proud of the **Scout Flight Controller**. While it is, in many ways, my greatest technical achievement, it is severely lacking in features to many modern-day consumer drones like those made by DJI. In the next chapter, I will list some of the areas that I will be improving on in my next iteration of quadcopter flight controller software!

Future Improvements



And just like that, you can build your very own quadcopter flight controller firmware, just as I covered in the previous ten chapters in this series.

I'm very proud of what I accomplished in building the *Scout* flight controller from the ground up. However, especially compared to the modern day consumer drones like those made by DJI, my combined hardware and software package is quite rudimentary! There are many improvements/extensions I'd like to add to the design in future iterations:

Angle Mode

In the previous chapter on quadcopter flight dynamics, I explain the two primary modes that a multicopter flight controller can operate in: **rate (or "gyro"/"acro") mode** and **angle mode**.

When a multicopter is in **rate mode**, its goal is for the aircrafts rate of rotation across each of the three axes to *match* the pilot's desired rate of rotation across each axis. In other words, in the absence of any pilot input, the quadcopter will dynamically vary the level of thrust to each of its motors to continue maintaining the aircraft's current orientation. As the pilot applies pitch/roll/yaw input, the motor thrusts are adjusted to achieve the desired rate of rotation.

When a multicopter is in **angle mode**, its goal is for the aircraft to achieve and maintain a specific **angle to the ground**, as specified by the pilot. In the absence of pilot input, the quadcopter will recover from an angle and attempt to maintain a perfectly level orientation to the ground (parallel to ground). As the pilot applies pitch/roll/yaw input, motor thrusts are continuously adjusted to achieve a "translated" angle to the ground.

The Scout flight controller I developed implements the **rate mode** scheme. While this mode is viewed as the more "advanced" form of flight and is preferred for performance-oriented pilots, it is far from easy to use for a beginner. **Angle mode** is far easier to pick up and intuitively understand, and the "if you take your hands off the controls it will level out and not crash" is a tremendous "safety" feature.

The reality is angle mode is significantly more complicated to implement. While the actual code may not be tremendously different (perhaps another 100 lines), integrating noisy accelerometer data into the equation (rate mode only uses gyroscope while angle mode uses both gyroscope and accelerometer) could be daunting.

In future iterations of my flight controller, I'm hoping to have both **rate mode** and **angle mode** built in. The pilot can use a toggle switch to switch between the modes either while stationary or perhaps even during flight.

Battery Voltage Indicator

As mentioned in the previous chapter on taking flight, *Scout* has no means of knowing the state of charge (battery level) of the onboard 4S LiPo battery.

This isn't a problem for the small controlled flights I took my quadcopter on. But for anything more, a battery level indicator would be very important. A simple inexpensive battery voltage indicator could be installed onboard that constantly monitors the voltage of the 4S battery pack.

Firstly, a battery level indicator would make the quadcopter significantly safer. During small test flights, my battery completely lost power two or three times. I found that performance degrades for a short period of time and then suddenly the power goes out completely and the quadcopter tumbles out of the sky. With such little (or no) notice, this is a safety hazard. By knowing when the battery gets to dangerous levels (i.e. below 20%), I would know when to ground the aircraft to avoid any hazards.

Secondly, by knowing when to land the drone due to low battery, I'm preserving the life of the 4S LiPo battery. By discharging to extremely low levels like I originally was (below 12 volts on 4S), I'm reducing the potency and lifespan of my battery.

Position Hold

Along with the addition of the **angle mode** mentioned above, greater "autonomous" capabilities would be of value. More specifically, the components required for a sort of *position hold* capability. In my *Scout Flight Controller*, the only consideration that the flight controller makes when determining what level of thrust to apply to each of the four motors is the *rate of orientation (attitude) change*. If the quadcopter is *also* aware of its position in space (latitude and longitude) and altitude, we could also develop a program that would use these as inputs as it aims to maintain a position in three dimensional space.

First, the quadcopter would need to know its location — latitude and longitude. For this, it would need an onboard GPS module. GPS modules are inexpensive today and easy to work with. The primary challenge here would be getting a latitude and longitude with precise accuracy. The inexpensive GPS modules are fairly inaccurate (off by a meter or more at times) at their best. This would be unacceptable for this use case. A high-resolution GPS module would be required that is capable of accuracy of perhaps a foot or two at the very most. I have not investigated, but there must be a way to accomplish this with either A) a higher-resolution (probably higher cost) GPS module or B) mixing together signals of multiple GPS modules to get an accurate "average".

Secondly, the quadcopter will need to know its altitude. An inexpensive barometer can be used to determine the ambient pressure the quadcopter is experiencing and this can be converted to an altitude. Similar to the GPS accuracy problem, I'd imagine it would be challenging getting an accurate enough reading within a few inches of altitude.

With both an ability to know its latitude, longitude, and altitude, we could, in theory, develop a program that also strives to maintain a specific point in 3D space, similar to what DJI and other drone companies have done.

Dual Flight Controller/Flight Computer System

The *Scout flight controller* is a **Flight Controller**. A flight controller is a system that is purely intended to run a high-speed program that routinely and rapidly determines what levels of thrust to apply to each motor to stay airborne and comply with the inputs of the pilot.

A **flight computer** is a system built *on top of* the flight controller and inputs commands into the flight controller. While flight controller is purely focused on staying airborne and stable flight, the flight controller operates at a "higher level" and can be focused on things like obstacle avoidance, position hold, waypoint missions, communications with ground, etc.

Nicholas Rehm has a fantastic video explaining the differences on YouTube:

https://www.youtube.com/embed/PlKeFj5teo4?si=6MCRu-3CqqTsBLAI

As a future project, I'd like to develop a dual system. While the flight controller code would not need much of a change (*Scout* with some alterations), a flight computer would directly integrate with the flight controller and control things such as position hold, radio communications, telemetry capture, etc. All of this is currently not achievable on the flight controller alone due to resource constraints (needs to run quickly, not enough processor speed headroom to support).

Controlled Landings with Ultrasonic Range Finder

Many modern day consumer drones, including most made by DJI, have an ultrasonic range finder on their underbelly. This range finder is capable of measuring distance to objects below the drone (the ground) and use this measurement for controlled landings. When in a certain proximity to the ground (say, 5 feet), the maximum speed of descent is lowered and continues to lower as the drone gets closer. This allows for a more controlled and precise landing.

With the proximity to the ground (altitude measurement at low altitudes), controlled landings, and perhaps a form of altitude hold, is achievable.

Communications

As discussed in depth in the previous article on the RC controls, a radio transmitter and receiver is exclusively used to control the quadcopter.

This means that the only transfer of data between the ground and the aircraft are control commands. As the drone system becomes more sophisticated (features described above are introduced), the

ability to maintain data communication while flying would be massively beneficial.

During flight, the drone can transmit data such as telemetry, battery level, etc. Conversely, the pilot, using another system, can deliver various commands beyond just roll, pitch, yaw, and throttle. For example, the ground can deliver waypoint mission commands for the drone to follow.

Without investigating heavily into how this could be accomplished, it seems that **Bluetooth** may be a solution. While Bluetooth would only allow for this communication in close proximity, it is a well-understood protocol that is also commonly implemented and available in microcontrollers/computers.

Running Lights

Running lights onboard the quadcopter will assist with navigation and also look very cool. A simple strand of WS2812B individually addressable LED lights will be plenty. The drone's flight computer could also use this to transmit messages optionally as well.

Next Chapter

Next, along with sharing all of the code required in this project, I'll share some bonus code — code that I never ended up using but can easily be used in other projects.

Thanks for reading on!

Bonus Code

At this point in my series, I will share any bits of code that were useful at some point during development, perhaps may still be incorporated into Scout's source code today, or otherwise are no longer used.

Firstly, you can find the full repository of the *Scout Flight Controller's* source code on GitHub here.

Many of these code snippets can be used in similar projects or in other projects using the same components. I hope these are helpful!

MPU-6050 MicroPython Driver

Probably the most reusable component of this project that can easily be used in any other project is my **MPU-6050** driver.

This is a very lightweight module that allows you to very easily read accelerometer and gyroscope data from an MPU-6050.

You can find the source code of my MPU-6050 MicroPython driver here.

I described how to use this class a bit in an earlier chapter about the MPU-6050.

ESC Calibration Script

All four of the electronic speed controllers must be *calibrated* to ensure the throttle commands being sent to each are even across all four. I wrote a basic script for performing this ESC calibration that you can find on GitHub here.

While this script is specifically designed for the ESC's I purchased and used during this project, I believe most ESCs follow the same, if not a very similar, calibration process.

Analog Potentiometer to PWM Converter

I wrote a basic script that converts the signal from an analog potentiometer into a digital pulse width modulation that can be used to drive a motor via an ESC. Thus, you can easily control a BLDC through an ESC with a potentiometer during testing.

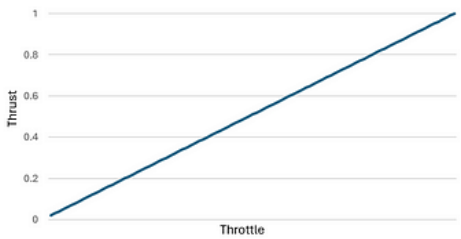
You can find this script on GitHub here.

NonlinearTransformer

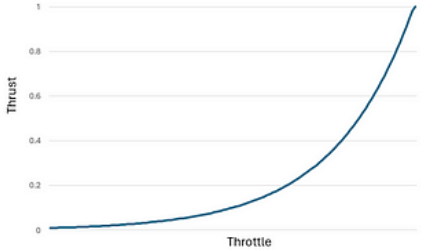
When interpreting the level of thrust to apply given a given level of throttle input, we have to consider that we don't want the quadcopter to become "twitchy"; we don't want it to react too strongly to minor throttle/attitude commands it is receiving from the pilot. So, to "ease in" these inputs while also maintaining the same level of performance at the far reaches of these inputs, I developed the `NonlinearTransformer` class.

This class will transform any percentage level of input (0%, or 0.0 to 100%, or 1.0) to a Nonlinear equivalent. In other words, this

class “softens” the input and introduces the input more gently. For example:



Linear relationship between throttle input and thrust



Nonlinear relationship between throttle input and thrust

We see two graphs above. In the first, there is a linear relationship between input (X) and output (Y). 25% throttle will yield 25% thrust. In the second, on the right, there is a nonlinear relationship between input and output. 25% input will result in 10% throttle. Using this NonlinearTransformer class as a middle layer between the direct control input and the applied levels of thrust can make the drone feel less “twitchy” and smoother to fly.

You can find the `NonlinearTransformer` class in the toolkit.py module that is still part of *Scout's* source code.

ControlCommand

Originally, my plan was to send packets of data over UDP to control *Scout* on a local network. I later pivoted to using the RC receiver and transmitter discussed previously, but some of the remnants of the original implementation still remain.

I wrote the `ControlCommand` class as a data packet that contains the **throttle**, **roll**, **pitch**, and **yaw**. Each of those four are floating point numbers and intended to be between either 0.0 and 1.0 or -1.0 and 1.0. This class also has a **frame** property. This integer is incremented upwards with each packet that is sent; this serves as a validation to ensure the packet the quadcopter received most recently is indeed the most recent and not one that simply arrived late.

This `ControlCommand` class has easy methods for serializing the packet into a series of bytes, using the `encode` method, and then deserializing on the client side (quadcopter side), using the `decode` method.

You can find the `ControlCommand` class in the toolkit.py module that is still part of *Scout's* source code.

PIDCommand

Similar to the `ControlCommand` class, I also originally had an implementation that would allow me to update the quadcopter's PID tuning values on the fly (without having to re-flash the code). I designed the `PIDCommand` class for packaging a series of PID gain values (proportional gain, integral gain, derivative gain) and the intended axis, and then serializing these into a series of bytes which can be delivered to the quadcopter. The quadcopter would then unpackage these bytes and make the PID gain changes in the software.

You can use the `encode` and `decode` methods of the `PIDCommand` class for this. You can find the `PIDCommand` class in the toolkit.py module that is still part of *Scout's* source code.

That’s all… for now!

Thank you to all of you that followed along in this series. I truly hope my writings are of value. I’m sure this series will sit online

for quite some time and maybe be used by an aspiring developer who just wants to make something cool, just as I was when I started this project.

Please reach out to me on Twitter/X **@TimHanewich** with any questions. Thank you!