dRehmFlight VTOL

Teensy Flight Controller and Stabilization

NICHOLAS REHM

January 6, 2020

dRehmFlight VTOL

Teensy Flight Controller and Stabilization

Contents

Introduction

This project is a work in progress, and was originally developed for research vehicle platforms at the University of Maryland Alfred Gessow Rotorcraft Center.

dRehmFlight is a simple, bare-bones flight controller intended for all types of vertical takeoff and landing (VTOL) vehicles from simple multirotors to more complex transitioning vehicles. This flight controller software and hardware package was developed with people in mind who may not be particularly fluent in object-oriented programming. The goal is to have an easy to understand flow of discrete operations that allows anyone with basic knowledge of coding in C/Arduino to peer into the code, make the changes they need for their specific application, and quickly have something flying. It is assumed that anyone using this code has previous experience building and flying model aircraft and is familiar with basic RC technology and terminology. The Teensy 4.0 board used for dRehmFlight is an extremely powerful microcontroller that allows for understandable code to run at very high speeds: perfect for a hobby-level flight controller.

Rather than a comprehensive package suitable for plug-and-play type setup, this package serves more as a toolkit, where all the required difficult computations and processes are done for you. From there, it is simple to adapt to your specific vehicle configuration with full access to every variable and pinout, unlike other flight controller packages. The code is easily modified and expandable to include your own actuators, data collection methods, and sensors.

This code is entirely free to use and will stay that way forever. If you found this helpful for your project, donations are appreciated: https://www.paypal.me/NicholasRehm

Introduction Video:
https://www.youtube.com/watch?v=tlD0C5CrWcA

1 Overview

This package is a flight controller intended to simply stabilize an otherwise unstable vehicle platform such as a multirotor, small helicopter, or other unique VTOL platform. It may also be used to supplement passively stable platforms such as airplanes. This is not a flight computer, which would handle advanced autonomy, computer vision, or path planning. However, dRehmFlight can be modified to accept a serial input with simulated radio pilot commands, which could be generated by a separate onboard companion computer. This is outside the scope of this project. dRehmFlight is meant to accept radio commands from a ground-based pilot, interpret them as vehicle state setpoints (desired vehicle angle or rotation rate), and have the vehicle perform those commands.

The code is designed to run on a Teensy 4.0 microcontroller with the MPU6050 IMU which together cost less than $30. The Teensy 4.0 is an extremely powerful microcontroller with 40 digital pins, 31 of which are PWM-enabled, and all of which are interrupt capable. It's ARM Cortex-M7 microprocessor features a 600 MHz clock speed enabling a flight control loop rate of 2 kHz. The code is compiled and uploaded to the board using the Arduino IDE and the Teensyduino add-on. Up to 6 radio inputs can either be supplied via conventional PWM on individual channels/pins, PPM over a single pin, or SBUS over a single pin. The code is easily modified for additional channels as well by following the general methodology of the existing 6 channels included by default.

The code provides all the necessary radio interface, IMU interface, and simple controller methods to allow for complete stabilization of a small flying vehicle. Radio data is gathered using the Teensy 4.0's interrupt and serial capability from a separate radio receiver. This is done over multiple pins through individual PWM, or one single pin through PPM or SBUS. Any radio controller with either

of these protocols will be able to interface with dRehmFlight, and brand-specific channel mapping can be accounted for in the code or radio itself. IMU data from the default MPU6050 is received over one of the Teensy's I2C buses. Support for the MPU9250 9DOF IMU is also included using one of the Teensy's SPI ports. This data is lightly processed to give a cleaner signal for the raw gyro and accelerometer readings, and fused together using a Madgwick filter to give roll, pitch, and yaw estimates. All of this data is available to the user at any time in the main flight control loop, as the global variables are continually updated for each loop iteration. The radio data is converted into desired state setpoints such as desired angle or angular velocity and combined with the orientation data from the IMU in the actual controller portion of the main loop. Here, stabilized axis variables for roll, pitch, and yaw are generated based on the error in current vehicle state from the desired state. Currently, there are three implemented options for control, all of which are PID-based methods. The stabilized axis variables from the controller are then applied to the actuator outputs of the vehicle in the control mixer. The control mixer is the primary area of interest in the flight control loop for configuring the code for any type of platform. Here, the user is free to assign stabilization to the motors or servos however the vehicle configuration requires. The user may also assign different mixing configurations based on the state of an auxiliary radio channel, allowing for VTOL transition vehicles with variable dynamics. Finally, the stabilized and mixed actuator commands are scaled appropriately and then written out to the actuators themselves at their designated pins. The preferred method for controlling motors is OneShot125 protocol which has been implemented in the code by default. As a result, it is recommended to use BLHeli ESCs capable of interpreting OneShot125. If this method is not desired, the code supports 7 servos operating on conventional PWM which can also be used to command motors through older ESCs that do not support OneShot125.

All of this code is compiled and uploaded through the Arduino IDE and Teensyduino add-on which allows the Teensy 4.0 board to be treated as an Arduino board. The code is written in Arduino which is a variant of C and relatively easy to understand with any prior experience with other coding languages. With the exception of a few areas of the code required for the IMU interface and data processing, everything else is written in such a way to favor understandability rather than performance/speed. Basic knowledge on variables and logical statements will allow for complete understanding of the core processes enabling the operation of this flight controller. The same simple coding practices can then be used to modify the code for your specific application.

To reiterate: this is a simple flight controller designed to be easy to understand, and intended to be modified before put to use in virtually any small VTOL configuration. This package is not intended to be used on very large vehicles (exceeding 20 lbs) or human carrying aircraft, but rather small MAV-scale vehicles used for hobby or academic research purposes. It is not an autopilot capable of path planning or other autonomous functionality by default, but the computing power of the Teensy 4.0 will allow the user to modify the code to potentially add this functionality as they see fit for their application. dRehmFlight should be exclusively thought of as a starting point for flight control and stabilization of small flying vehicles of varying dynamic configuration.

2 Hardware Requirements

The physical flight controller which runs the dRehmFlight code consists of a Teensy 4.0 microcontroller and MPU6050 6 degree of freedom IMU. Currently, only the Teensy 4.0 board has been tested, though the newer 4.1 board with more pinouts should work identically.

2.1 Teensy 4.0

The Teensy 4.0 is an extremely capable and cost-effective microcontroller, intended to be used for robotics, audio projects, or other Arduino project-type applications. The Teensy 4.0 features an ARM-Cortex M7 processor with a clock speed of 600 MHz. It has 31 PWM-enabled pins and 40 digital i/o pins which are all interrupt capable. Additionally, 3 I2C, 3 CAN bus, and 7 serial ports allow for multiple options to interface various sensors. Please note that the Teensy 4.0 operates on 3.3v logic. As a result, all sensor inputs to the board must be 3.3v logic level and not 5v to prevent damage to the board.

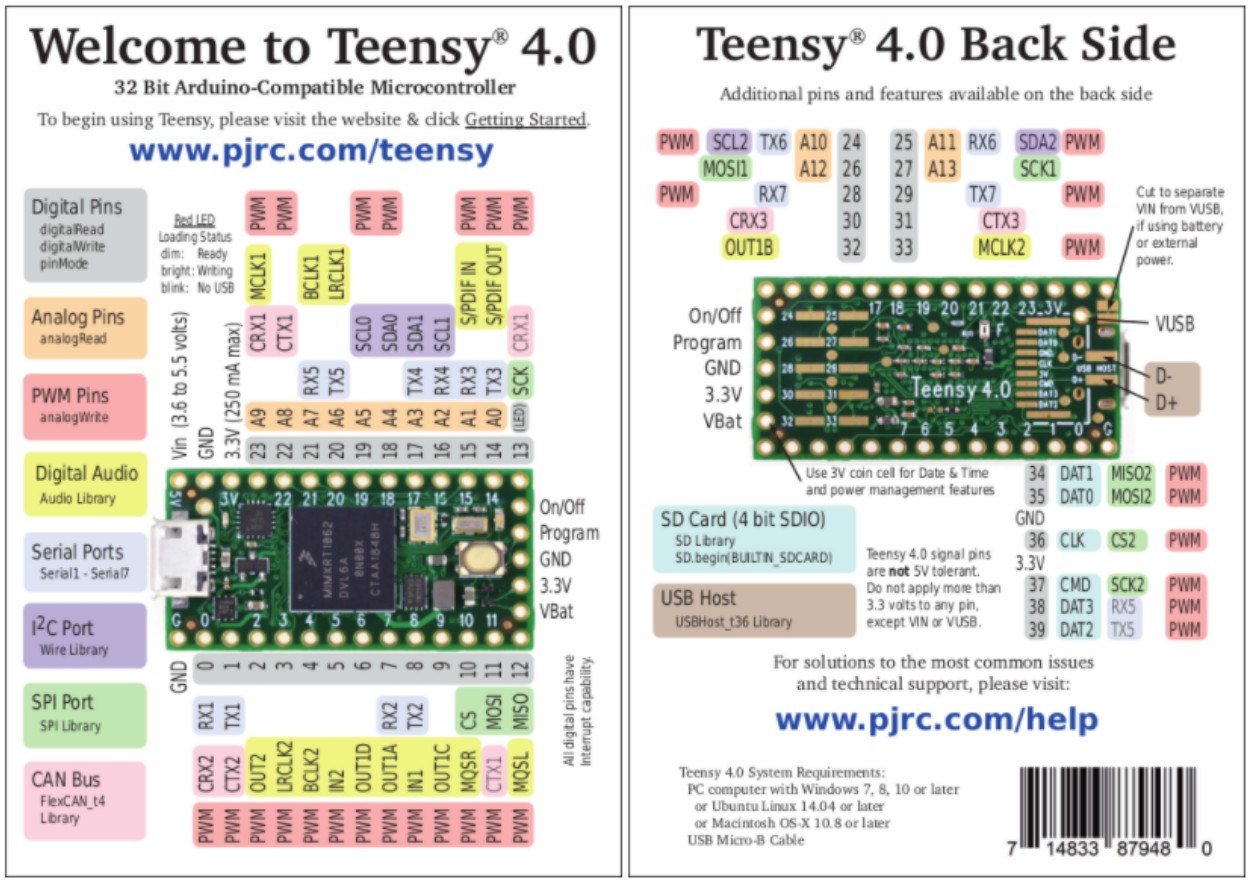

Figure 1: Teensy 4.0 pinout.

For more information and detailed specifications:
https://www.pjrc.com/store/teensy40.html
Amazon purchase link:
https://amzn.to/2V1q2Gw

2.2 MPU6050 IMU

The GY-521 breakout of the MPU6050 6 degree of freedom IMU is used for attitude estimation for the default recommended hardware setup. This board is extremely cost effective, reliable, and gives adequate measurement precision to stabilize a small flying vehicle.

For more information and detailed specifications:
http://www.haoyuelectronics.com/Attachment/GY-521/mpu6050.pdf
Amazon purchase link:
https://amzn.to/3edF1Vn

2.3 MPU9250 IMU

The code also supports the MPU9250 9 degree of freedom IMU. The addition of the 3-axis magnetometer may be desireable for some users for better attitude estimation (particularly in the yaw axis), though it is not necessary for general applications and requires a bit more attention to for proper setup. More information on the implementation of this IMU is provided in a tutorial at the end of this document.

For more information and detailed specifications:
https://invensense.tdk.com/wp-content/uploads/2015/02/PS-MPU-9250A-01-v1.1.pdf
Amazon purchase link:
https://amzn.to/3ijlZOo

3 Hardware Setup

3.1 Suggested/Default Setup

The code by default provides outputs for 6 motors and 7 servos, as well as options for PPM receiver in, SBUS, or standard 6 channel PWM receiver input. The following hardware configuration requires the least amount of soldering with the Teensy 4.0 to get the board up and running with the default code and MPU6050 IMU, which all together weighs less than 15 grams. This is merely a suggested setup that is immediately compatible with the default code. Please feel free to use your own method–from breadboards to PCBs–to set your hardware up for your application.

Tutorial Video:
https://www.youtube.com/watch?v=EBXBEB-Xv7w&

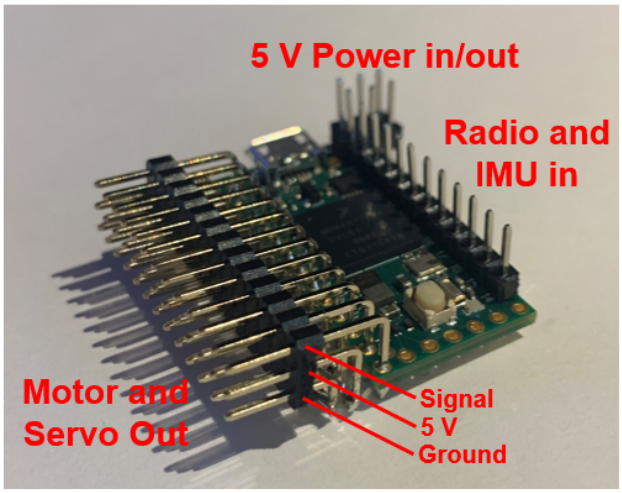

Figure 2: General i/o layout of the broken-out Teensy 4.0 board.

Figure 2 shows the general layout with header pins soldered directly to the Teensy 4.0. The left side contains outputs for 6 motors using OneShot125 protocol and 7 servos/motors using conventional PWM at 50 Hz. The right side contains pins to interface with the MPU6050 IMU and radio input, using either a single PPM or SBUS radio pin, or individual PWM pins for each of the supported 6 channels. The pin headers must be soldered to the board. Figure 3 and Table 1 details the functionality of each numbered pin and corresponding location on the broken out configuration for the default recommended setup.

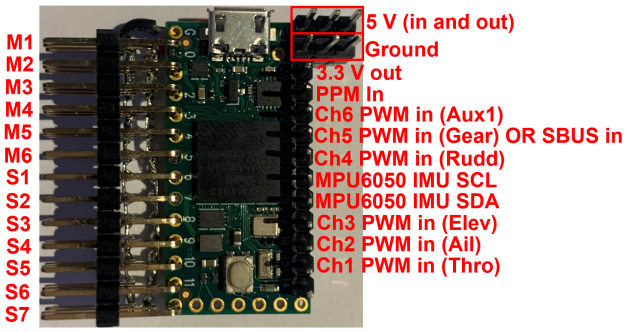

Figure 3: Pinouts on the broken out Teensy 4.0.

Pinout	Pin Number	Notes
M1	0	OneShot125 protocol
M2	1	OneShot125 protocol
M3	2	OneShot125 protocol
M4	3	OneShot125 protocol
M5	4	OneShot125 protocol
M6	5	OneShot125 protocol
S1	6	50 Hz PWM
S2	7	50 Hz PWM
S3	8	50 Hz PWM
S4	9	50 Hz PWM
S5	10	50 Hz PWM
S6	11	50 Hz PWM
S7	12	50 Hz PWM
PPM in	23	If using PPM receiver
Ch1 PWM in (thro)	15	If using PWM receiver
Ch2 PWM in (ail)	16	If using PWM receiver
Ch3 PWM in (elev)	17	If using PWM receiver
Ch4 PWM in (rudd)	20	If using PWM receiver
Ch5 PWM in (gear) OR SBUS in	21	If using PWM receiver or SBUS
Ch6 PWM in (aux1)	22	If using PWM receiver
IMU SDA	18	MPU6050 IMU I2C connection
IMU SCL	19	MPU6050 IMU I2C connection

Table 1: Pinout assignment in default code for recommended board setup.

A combination of angled 3-pin male headers and standard male headers are required to complete the flight controller assembly. First, a row of 3-pin male angled headers (13 pins in length) must be soldered to the left side of the board when looking at the top with the micro USB toward the front (Figure 4). This is soldered on from pin 0 to pin 12 on the board, with the closest row of the 3 pins soldered into the Teensy's pins, and 2 rows hanging over the left side. The protruding pin rows are used for 5 V power to servos (or power in from an ESC) and ground. Next, a row of single male pin headers (11 pins in length) are soldered to the right side of the board from the 3 V pin to pin 14 on the Teensy. Finally, two sets of 3 single male pin headers are soldered on to the top right corner to the 5 V and ground pins on the Teensy. These are soldered such that two additional pins hang over the right side of the board and allow for multiple devices (IMU, receiver) to receive power. These additional pins will be wired together for an accessible 5 V power rail and ground rail.

Angled 3-Pin Headers Amazon Purchase Link:
https://amzn.to/3cM76TK
Single Headers Amazon Purchase Link:
https://amzn.to/2GhbVsv

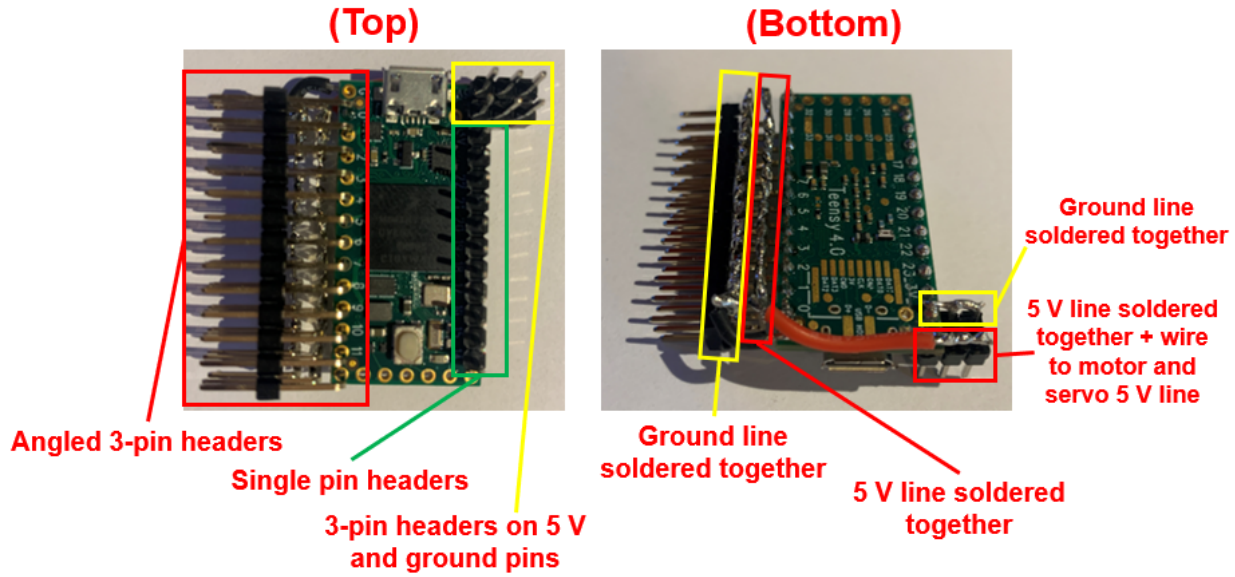

Figure 4: Special notes for soldering the Teensy 4.0 for the default setup.

Once the pin headers are soldered on to the Teensy, there is minimal wiring required to complete the assembly. First, the 3-pin headers extending over the side of the board must be wired together to create a 5 V rail and a ground rail. To do this, 22 AWG wire can be stripped and a few strands weaved between the pins before soldering the complete rail together. This must also be done for the 3 pin long power pins on the other side of the board. In this case, pins are connected such that it will create 3 total 5 V pins and 3 total ground pins extending out the side. Once done for all of the rails, the outermost rail of the 3-pin angled headers must be soldered to the nearest ground pin on the Teensy with a short piece of 22 AWG wire. Finally, the 5V rail of the 3-pin angled headers must be connected to the 5 V power rail on the other side with a short piece of 22 AWG wire. Please note that if using many or high current draw servos, the 3-pin angled header power

line should be powered by a different power source than the 5 V pin powering the Teensy. This is because the Teensy is sensitive to voltage dips or spikes that servos may cause on smaller voltage regulators. For general use, however, the Teensy and servos can operate off of the same 5 V power source such as an ESC's BEC.

One final note: if using more than 2 servos, it is recommended to cut the two small pads on the underside of the board to prevent a USB connection from attempting to power the servos (Figure 5, Figure 1 shows the location on the board). If the servos are drawing too much current, this could damage the computer's USB port. The downside of this method is that external power must be supplied to the board in order to connect to the computer over USB. An alternative to this is to either unplug all servos before plugging into the computer, or wire a SPST switch between the servo power rail and 3 pin power rail, which can be switched on and off when plugging into the computer.

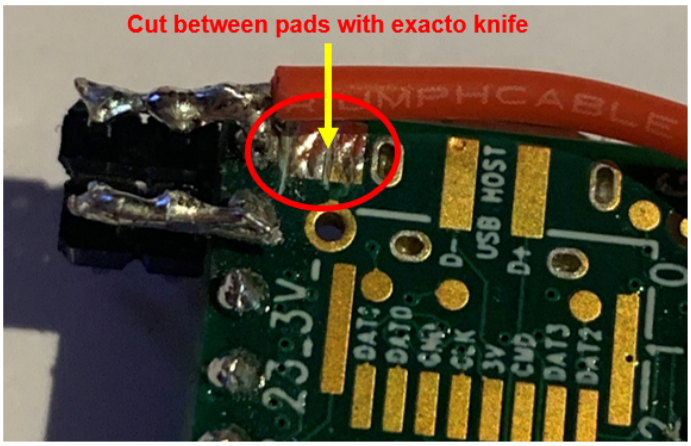

Figure 5: Pads to separate in order to isolate USB power and prevent a USB connection from powering the board when using many servos.

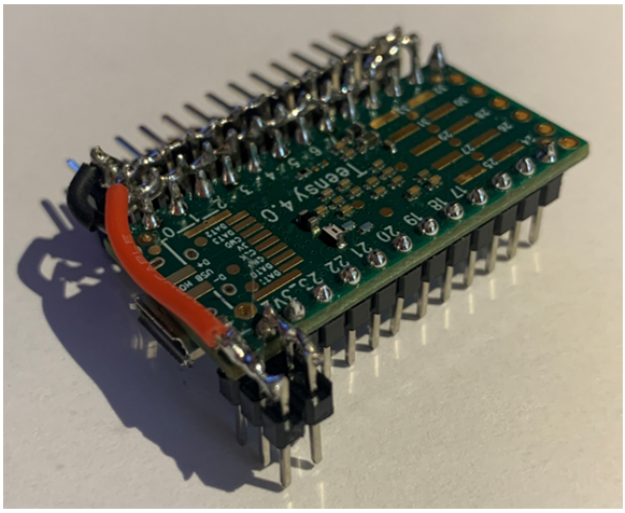

Figure 6: Underside view of completed broken out Teensy 4.0.

Next, the MPU6050 IMU is soldered separately so that it can be placed in a secure location on the vehicle in the correct orientation indicated in Figure 16. Four 22 AWG wires of length of the user's choosing must be soldered to the gy-521's VCC (5 V), GND (ground), SCL, and SDA pins. Two sets of two female pin headers (JST connectors would work too) are soldered to the 5 V and ground wires, and the SCL and SDA wires. This process is shown in Figure 7. If using the MPU9250 IMU instead of the default MPU6050, please see the tutorial at the end of this document for the hookup procedure.

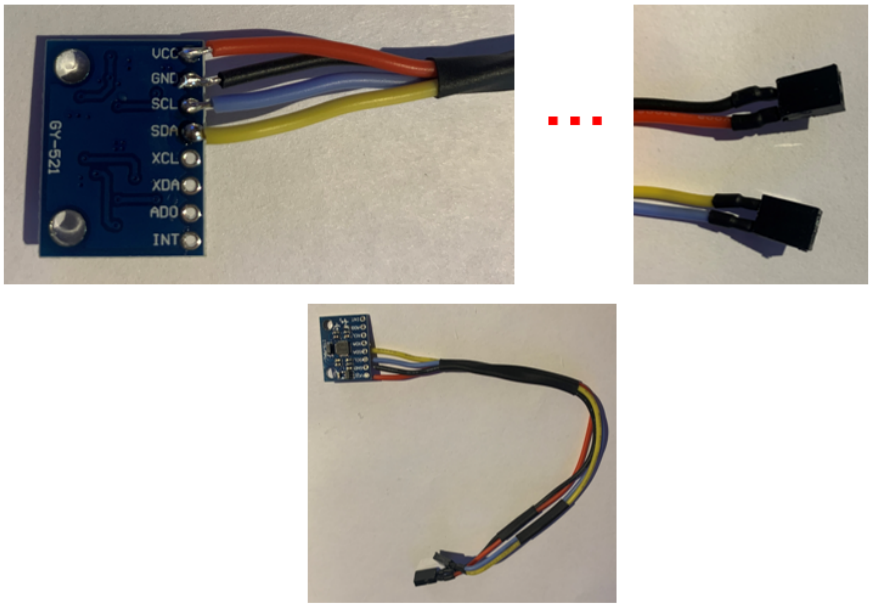

Figure 7: Soldering the connection for the flight controller's MPU6050 IMU.

After soldering the IMU properly, the two sets of two female pins can plug into the broken out Teensy assembly, shown in Figure 8. The VCC and GND pins from the MPU6050 gy-521 board plug into one of the three 5v and ground pin options on the broken out Teensy, and the SDA and SCL pins plug into pins 18 and 19 on the Teensy, respectively.

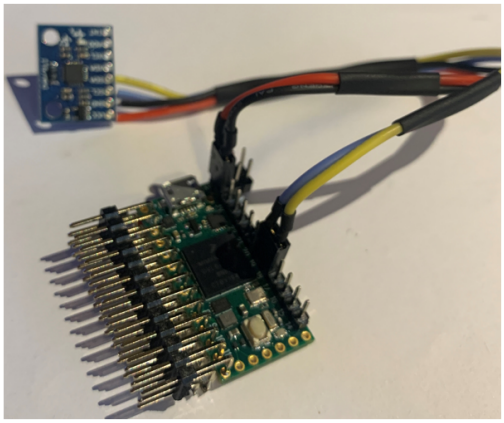

Figure 8: Plugging the IMU into the broken out Teensy 4.0.

To complete the full flight controller assembly, the receiver is plugged into the flight controller. Figure 9 shows this process being done for a conventional Spektrum PWM receiver. Please note the channel mappings in Tables 1 and 2 for their corresponding functionality and pin numbers. In this case, the Spektrum channel mapping corresponds perfectly with the channel mapping designated for dRehmFlight. If using a PPM or SBUS receiver, only a signal pin is required to transmit the full 6 channels. In all cases, the receivers must connect to one of the three 5 V and ground pins on the broken out Teensy to receive power. With this complete setup, 5 V power can be applied to the flight controller anywhere (through the motor and servo rails, extra broken out 5 V and ground pin, or directly to the receiver) and all systems will be properly powered for full operation.

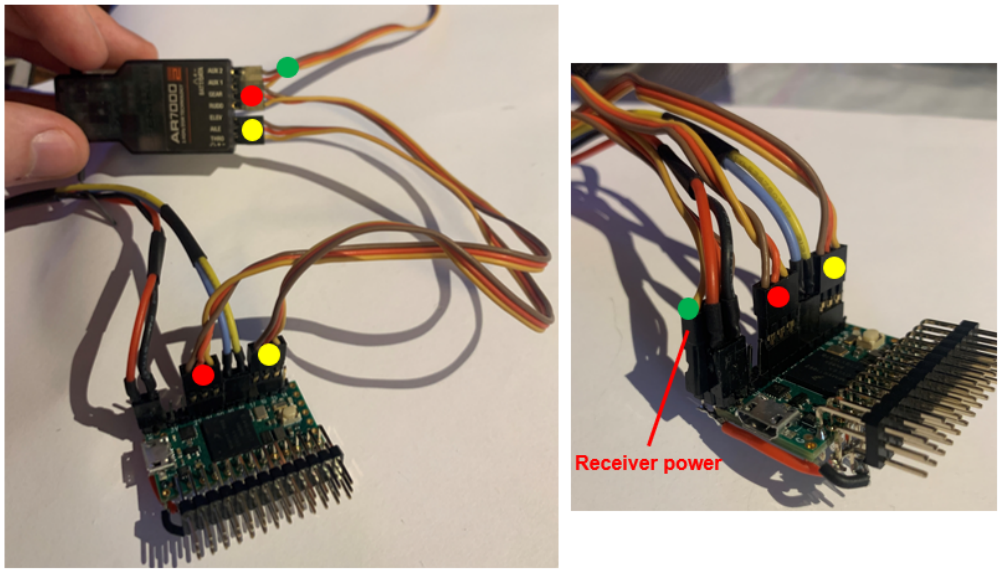

Figure 9: Plugging the receiver into the flight controller.

3.2 Alternative Setups

The suggested hardware setup is only a starting point for this flight controller. Custom setups may be required for different or expanded pin functionality. Figure 10 shows an early flight controller prototype soldered to a small prototyping board. In this case, the IMU is rigidly connected to the assembly rather than separated as is the case with the suggested hardware setup. The Teensy board attaches to the prototyping board with male and female pins so that it is removeable and can be transferred to other vehicle. Investing in creating proper breakout boards for the Teensy may be desirable to avoid having to purchase multiple Teensy boards for multiple vehicles. This particular breakout of the Teensy 4.0 has different pinouts than that of the default code, which must be modified accordingly. The next section as well as a pin selection tutorial at the end of this document detail the process of establishing required pins and modifying the code accordingly.

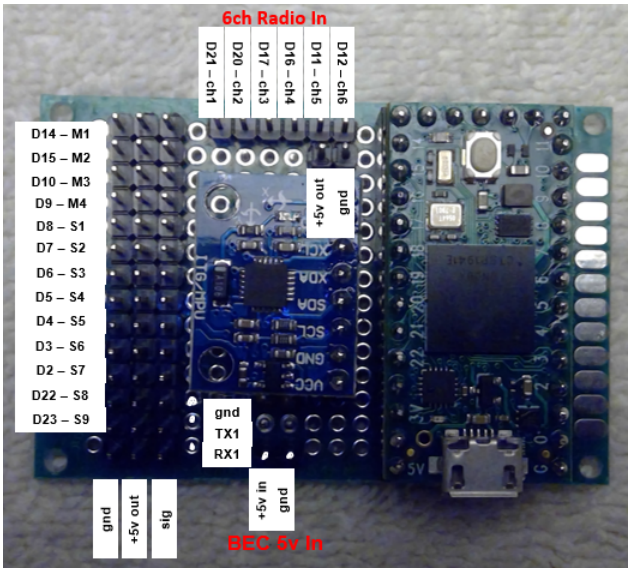

Figure 10: Early Teensy flight controller prototype.

3.3 Notes on Pin Selection

The user is free to select the pins on the Teensy 4.0 as they see fit for their application. Default pins are suggested in the above suggested setup for hobbyists, though the user may be interested in reserving serial or I2C ports for custom sensor integration. With that said, please use the following guidelines in conjunction with Figure 1 when assigning custom pinouts in the code:

- 5v Power is supplied to the Vin pin in the far corner by the micro USB. The board must be grounded on any available ground pin.
- Do not use pin 13; this pin is directly connected to the onboard LED which is used throughout the code to signal proper operation of the flight controller.
- Pins 18 (SDA0) and 19 (SCL0) are used to interface with the MPU6050 IMU over the I2C bus.
- If using the MPU9250, pins 34-37 on the back side of the board are used to interface over SPI connection.
- Any servo actuators must be connected to a PWM capable pin.
- ESCs using OneShot125 protocol may use any digital pin. Otherwise, they must be connected to a PWM capable pin.
- Radio input pins (for PWM in or PPM in) can be connected to any digital pin.
- SBUS radio input should stay connected to serial RX5 (pin 21).
- The Teensy boards operate with 3.3 V logic level. Any sensors must operate at this level or be stepped down accordingly to avoid damaging the board.

4 Software Requirements

The flight control code is written in the Arduino/C language and is uploaded to the board using the Arduino IDE. To connect to the Teensy, you must also download and install the Teensyduino Arduino add-on.

Download the latest version of the Arduino IDE here:
https://www.arduino.cc/en/main/software
Download the Teensyduino add-on for Arduino here:
https://www.pjrc.com/teensy/td_download.html

5 Software Setup

5.1 Downloading the Code

1. Navigate to the GitHub repository for dRehmFlight: https://github.com/nickrehm/dRehmFlight
2. Download the full package:

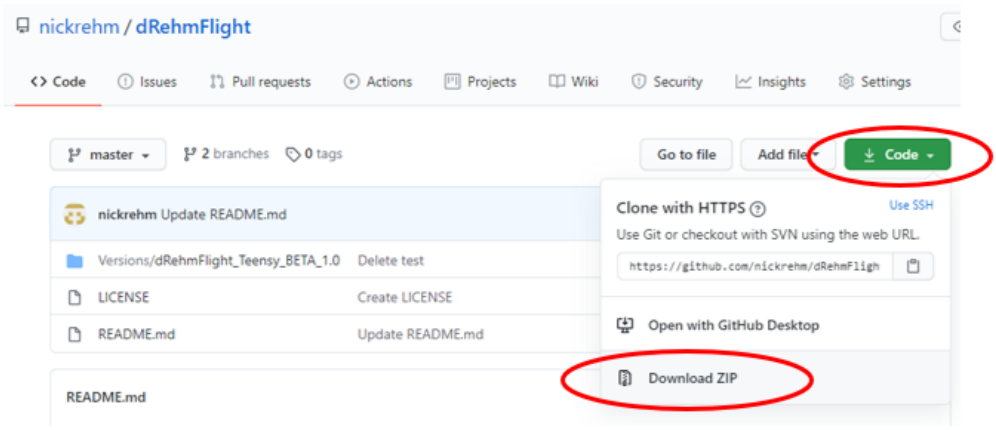

Figure 11: Downloading the repository from Github.

3. Save and unzip the .zip folder to your desired location:

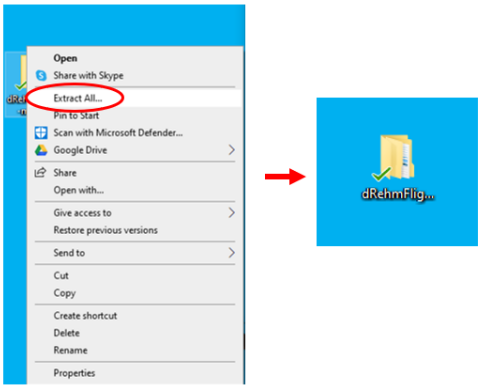

Figure 12: Unzipping the downloaded file.

4. Navigate to 'versions,' then the latest version of dRehmFlight. The user is free to modify the contents of the folder, including the Arduino files. However, the folder name and the main Arduino sketch name must be identical should either of them be changed.

5.2 Selecting the Board and COM Port in Arduino

After installing Arduino and Teensyduino and plugging in the Teensy 4.0 using a micro USB cable, the computer should automatically detect and install the necessary drivers for Teensy. Open the dRehmFlight Arduino sketch and navigate to 'Tools,' then 'Board,' then 'Teensyduino' and select 'Teensy 4.0':

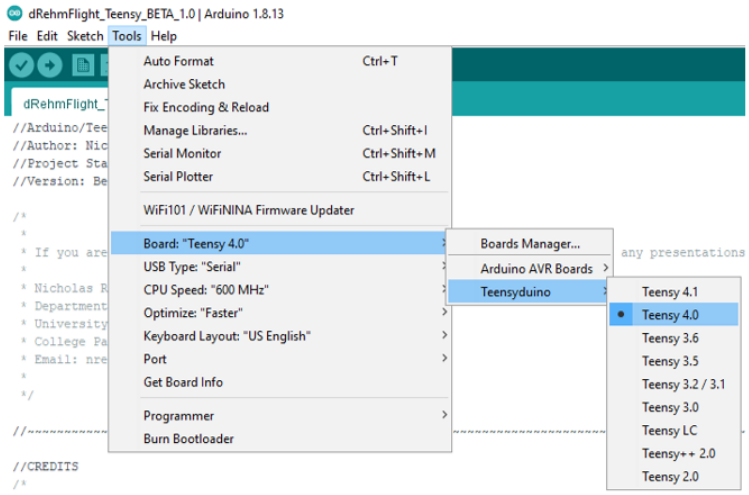

Figure 13: Selecting the correct board in Arduino.

To access the Arduino serial monitor or serial plotter for troubleshooting and setup, navigate to 'Tools,' then 'Port,' and under 'Serial Ports' NOT 'Teensy Ports,' select 'COMX (Teensy) where X is the numbered port arbitrarily assigned by your computer:

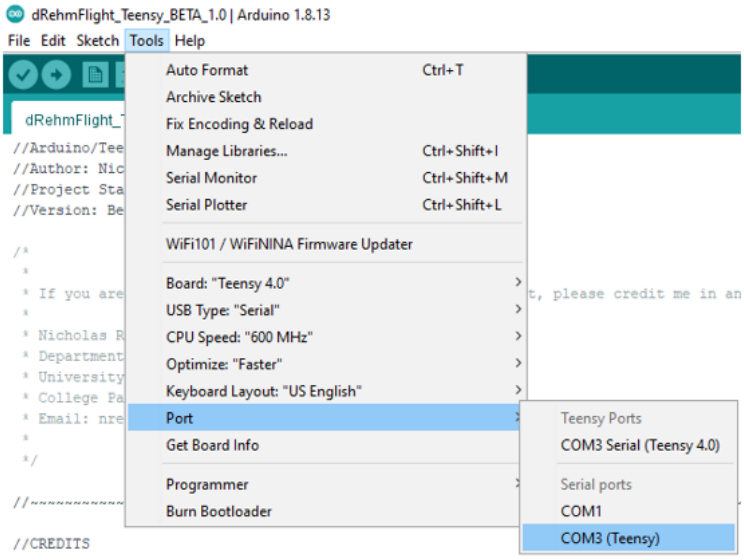

Figure 14: Selecting the correct COM port in Arduino.

Selecting this port will allow the user to use Arduino's serial monitor and plotter with the Teensy. If the serial monitor or plotter is opened while the other is already opened, an orange warning will appear at the bottom of the screen and you may not be able to open either, even after restarting the board. Save and close Arduino, then repeat the above process to fix this issue.

5.3 How to Use Teensyduino

Teensyduino is a passive tool that operates in the background of Arduino to compile and upload Arduino code to the Teensy board. When the user uploads code to the board at the top left of the Arduino IDE, Teensyduino will automatically open in a small window and ask to confirm uploading to the board. The user can click the small green 'Auto' button and Teensyduino will automatically upload to the board when clicking the Arduino upload button in the future. Sometimes it may encounter a problem, in which case it requests the user to press the small reset button on the board itself.

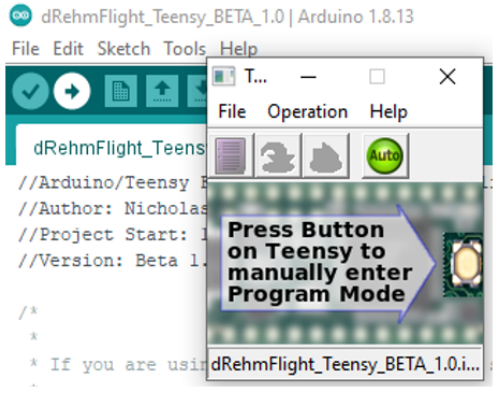

Figure 15: Uploading to the board with Teensyduino.

6 Conventions Used

6.1 IMU and Vehicle Orientation

Figure 16 shows the axis convention used for IMU data and vehicle orientation. +X is denoted as out the nose, +Y is out the left side, and +Z is upward. Pay special attention to mount the flight controller assembly (or just the IMU if it has been wired to be separate from the Teensy) in the correct orientation corresponding to this convention.

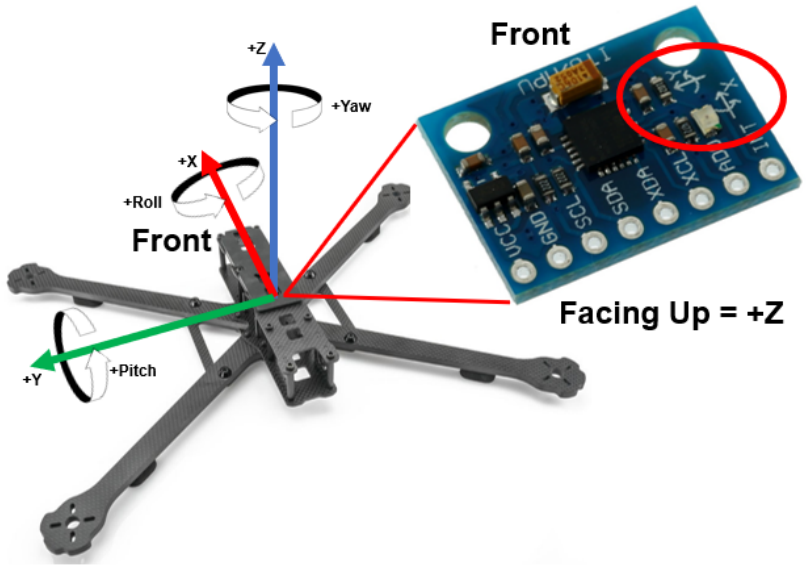

Figure 16: Vehicle and IMU axis convention used.

6.2 Radio Channel Mapping

Table 2 details the channel mapping assumed throughout the code. Channel numbers are used in code which correspond to their respective functionality. If you are unable to change the channel mapping within your radio to match this convention (or plug PWM channels into the correct assumed pin for the expected functionality), a change can be made within the radioComm file included with the current version of dRehmflight to re-assign your radio inputs to these assumed mappings.

Channel Number (In Code)	Function	Notes
1	Throttle	1000 min, 2000 max
2	Roll	1000 full left, 2000 full right, 1500 center
3	Pitch	1000 pitch up, 2000 pitch down, 1500 center
4	Yaw	2000 full left, 1000 full right, 1500 center
5	Gear	Throttle cut when greater than 1500
6	Aux1	Free auxiliary channel

Table 2: Channel convention used in code.

7 The Code

The main code is contained entirely within the dRehmFlight Arduino sketch (with the exception of radio communication functions which are included separately in the radioComm file in the same folder). Figure 17 shows the general sequence of contents that can be found in the main Arduino sketch.

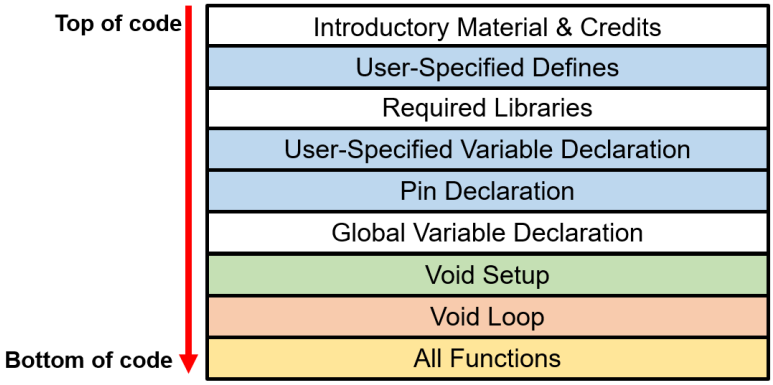

Figure 17: General contents in the main Arduino sketch.

The user-specified defines section is where receiver type, IMU type, and IMU sensitivity values are selected. The user-specified variable declaration section contains numerical variables such as failsafe values or controller gains which can be readily changed or tuned. Pins are assigned and declared in the pin declaration section (all sections are marked by comments within the code as well). Most variables used throughout the code are global variables. This is generally bad practice for coding, but allows for access to nearly every variable and parameter wherever we want within the code. All new variables created can be declared here. Next, the void setup section contains the setup code and functions which are executed one time on startup. Figure 19 expands on the processes occurring within the void setup. The void loop section contains the flight control code and functions which is run continuously after initial setup. Figure 18 shows the general architecture for a flight control loop and Figure 20 expands on the specific processes occurring within the void loop to achieve this functionality. Finally, all of the critical functions which are called within the void setup and void loop are contained at the end of the Arduino sketch. A specific operation is performed within each function, and they have been segmented in such a way that these key operations are easily understood and followed.

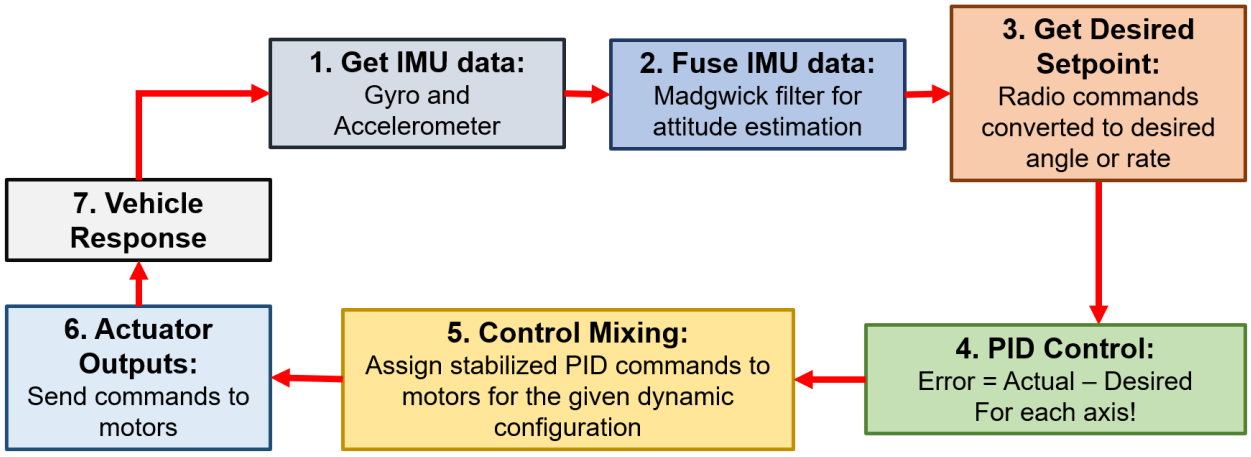

Figure 18: General overview of a flight control loop allowing stabilization of a small aerial vehicle.

7.1 void setup()

The void setup portion of the code is run one time at startup and is used to establish proper communication with pins, sensors, and the radio. It also allows for some safety features to be established such as failsafe functionality before any ESCs are armed in preparation for the main flight control loop. Detailed description of each function, relevant variables, and other information is provided in the Functions List below.

Step	Function/Process	Description
1	Serial.begin()	Establish serial connection to onboard USB
2	Initialize all pins	Motor and servo pins are assigned/attached to their designated pins from the pin declaration section
3	radioSetup()	Initialize communication with desired radio type using function within radioComm sketch
4	Failsafe setup	Set radio channel variables to default values before entering main loop
5	IMUinit()	Initialize communication with desired IMU
6	calculate_IMU_error()	Get error offset from IMU gyro and accelerometer to apply to readings in the main loop
7	Arm servo/PWM channels	Write low value to servos/motors using PWM
8	Arm OneShot125 channels	Write low value to ESCs using OneShot125
9	calibrateAttitude()	Warms up IMU filter before entering main loop

Figure 19: Flow of operation in the void setup() run one time on startup.

7.2 void loop()

The void loop portion of the code contains the main flight control code which is segmented into a few different functions. These functions are sequentially called for every loop and each carry out unique tasks for the complete flight control loop. Detailed description of each function, relevant variables, and other information is provided in the Functions List below.

Step	Function/Process	Description
1	Print Data (commented out default)	Multiple print statements used for troubleshooting; prints data to serial monitor at 100 Hz
2	getIMUdata()	Pulls raw gyro, accelerometer, and magnetometer data from IMU and low-pass filters to remove noise
3	Madgwick()	Uses IMU data to estimate absolute vehicle attitude
4	getDesState()	Convert raw radio commands to normalized values based on saturated limits (max rate or max angle in user-specified variables section)
5	Controller	Option of controlANGLE(), controlANGLE2(), or controlRATE() PID control to generate stabilized axis variables based on estimated attitude and desired state; gains adjusted in user-specified variables section
6	controlMixer()	Mixes PID controller outputs to scaled actuator commands based on dynamic configuration
7	scaleCommands()	Scale motor/actuator commands to required range for writing out to designated pins (OneShot125 default)
8	throttleCut()	Directly set motor commands to low/off value if throttle cut switch is engaged
9	commandMotors(), Command servos	Send ESC commands to OneShot125 pins and write servo commands to PWM pins
10	getCommands()	Get current radio commands to be used in next loop iteration
11	failsafe()	Check if received radio commands are valid and default to failsafe values if they are not
12	loopRate()	Idle in main loop until specified loop frequency is achieved (2 kHz default)

Figure 20: Flow of operation in the void loop() run continuously after startup.

8 Functions List

This section gives a detailed description of every function contained within the code. It is recommended to be familiar with the general flow of operations in Figures 19 and 20 first, and then become more familiar with each function, the important variables created or modified, and the options for customization. When in doubt, Ctrl + f will help to navigate through the code itself to find the discussed functions and variables, though the contents described in Figure 17 gives a good general starting location.

8.1 radioSetup()

Please note that this function is contained within the radioComm file, not the main Arduino sketch. This function sets up the radio connection for the desired radio type selected in the user-specified defines section of the code. For PWM-type receivers, the radio pin variables chXPin corresponding to the hookup on the receiver are declared as interrupt pins. This means that when any of the radio pins changes states from high or low, the corresponding interrupt service routine function (getChX, where X is the channel number) is immediately run, regardless of what the rest of the code is doing. These 'sub' functions measure the total time in microseconds for a high pulse. This high pulse duration is the radio command from the receiver, typically in a range of 1000 - 2000 us. This occurs for every individual channel separately. The generated 'raw' channel readings are not directly accessed in the main code, but rather the most current raw channel values are provided to the getCommands() function whenever that is called.

For PPm-type receivers, the PPM_Pin variable is declared as an interrupt pin. This means that when the PPM radio pin changes state from high or low, an interrupt service routine function

(getPPM()) is immediately run, regardless of what the rest of the code is doing. This 'sub' function measures the total time in microseconds for each high pulse in succession for the complete PPM pulse train. These high pulse durations are the radio commands from the receiver in order, typically in a range of 1000 - 2000 us. The order at which specific channels arrive and are assigned in the PPM pulse train can be modified in getPPM() to match the convention used in Table 2. The generated 'raw' channel readings are not directly accessed in the main code, but rather the most current raw channel values are provided to the getCommands() function whenever that is called.

For SBUS-type receivers, the bolderflight library is used to interpret the serial signal on RX5 (Pin 21). If SBUS is selected, communication with the SBUS receiver is started using sbus.begin().

8.1.1 Important Variables

Variable Name	Modifiable?	Type	Notes
ch1Pin, ch2Pin, ...	Yes, in pin declaration	const int	Pins on board assigned to radio channels for PWM mode
PPM_Pin	Yes, in pin declaration	const int	Pin on board assigned to radio PPM output for PPM mode
channel_1_raw, channel_2_raw, ...	No	unsigned long	Created and updated by ISR sub function continuously

Table 3: Important variables related to the radioSetup() function.

8.2 IMUinit()

This function initializes the MPU6050 IMU on the I2C bus at address 0x6B if the MPU6050 is selected in the user-specified defines section. If the MPU9250 is instead selected, this function initializes the IMU using SPI. Please note that by default, the code assumes the MPU6050 IMU is hooked up to SDA0 and SCL0 (pins 18 and 19).

8.2.1 Important Variables

There are no important variables pertaining to this function.

8.3 calculate_IMU_error()

This function reads the raw accelerometer and gyro data for each axis 12,000 times on startup and takes the average of these readings to give offset error values for each measurement. For this reason, the flight controller must remain stationary shortly after startup as these measurements are being taken. The computed offset error values are applied to the IMU measurements in the main loop in getIMUdata() such that every reading is zero (with the exception of the z axis accelerometer reading which should read 1) when the flight controller is level and at rest.

8.3.1 Important Variables

Variable Name	Modifiable?	Type	Notes
AccErrorX, AccErrorY, AccErrorZ	No	float	Computed at startup and applied to accelerometer readings in getIMUdata()
GyroErrorX, GyroErrorY, GyroErrorZ	No	float	Computed at startup and applied to gyro readings in getIMUdata()

Table 4: Important variables related to the calculate_IMU_error() function.

8.4 calibrateAttitude()

This function is used to 'warm up' the IMU and Madgwick filter, which fuses the accelerometer and gyro data for attitude estimation. Contained within is a simulated main loop, containing only the getIMUdata(), Madgwick(), and loopRate() functions. These functions are looped for 5 seconds (10,000 times at 2000 Hz) giving the Madgwick filter enough time to converge on the IMU's correct orientation before the code enters the main loop.

8.4.1 Important Variables

There are no important variables pertaining to this function.

8.5 getIMUdata()

This function pulls the raw accelerometer, gyro, and magnetometer data (if using MPU9250) for each axis from the IMU and applies a simple low pass filter to the readings to reduce noise. It reads accelerometer and gyro data from the IMU as AccX, AccY, AccZ, GyroX, GyroY, GyroZ and magnetometer readings as MagX, MagY, MagZ. These values are scaled according to the IMU datasheet to put them into correct units of g's and degree/sec, and micro Tesla respectively. Filter values B_accel, B_gyro, and B_mag contained in the user-specified variable section are tuned by default to cut off noise above about 80 Hz. A lower frequency cutoff will lead to a non-negligible lag in the raw IMU readings, and a higher frequency cutoff will give a noisier attitude estimate later.

8.5.1 Important Variables

Variable Name	Modifiable?	Type	Notes
AccX, AccY, AccZ	No	float	Units in g's, X and Y should read 0 at rest, Z should read 1 at rest
GyroX, GyroY, GyroZ	No	float	Units in rad/sec, should all read 0 at rest
MagX, MagY, MagZ	No	float	Units in uT
B_accel	Yes	float	Minimum 0, maximum 1. Decrease for lower cutoff frequency (slower response), increase for higher cutoff frequency (noisier)
B_gyro	Yes	float	Minimum 0, maximum 1. Decrease for lower cutoff frequency (slower response), increase for higher cutoff frequency (noisier)
B_mag	Yes	float	Minimum 0, maximum 1. Decrease for lower cutoff frequency (slower response), increase for higher cutoff frequency (noisier)

Table 5: Important variables related to the getIMUdata() function.

8.6 Madgwick()

This function fuses the filtered accelerometer and gyro readings AccX, AccY, AccZ, GyroX, GyroY, GyroZ, MagX, MagY, MagZ for attitude estimation of vehicle body angles roll_IMU, pitch_IMU, and yaw_IMU in degrees. A quaternion-based implementation of the Madgwick algorithm is used to efficiently and accurately estimate these body angles. The quaternion state estimation is available for data logging as q0, q1, q2, q3. If magnetometer data is not available (if using MPU6050 IMU), this function will call Madgick6DOF() which is a 6 degree of freedom implementation of the Madgwick algorithm instead. A tune-able filter parameter B_madgwick in the user-specified variable section adjusts the weight of accelerometer and gyro data in the state estimate. Higher B_madgwick leads to a noisier estimate, while lower B_madgwick leads to a slower to respond estimate. The default value for beta is tuned well and should not be modified for normal use.

8.6.1 Important Variables

Variable Name	Modifiable?	Type	Notes
roll_IMU, pitch_IMU, yaw_IMU	No	float	Units in degrees, body angles of IMU
q0, q1, q2, q3	No	float	Quaternion state estimation, body angles computed from this
B_madgwick	Yes	float	Minimum 0, maximum 1. Decreasing leads to slower estimate, increasing leads to noisier estimate

Table 6: Important variables related to the Madgwick() function.

8.7 getDesState()

This function generates the desired state variables thro_des, roll_des, pitch_des, and yaw_des from the radio PWM commands. With the exception of thro_des which remains normalized between 0 and 1, the remaining variables are scaled to be within the maximum bounds declared in the user-specified variable declaration section of the code. These bounds correspond to either degrees for angle mode, or degrees/second for rate mode operation. This function also creates the roll_passthru, pitch_passthru, and yaw_passthru variables, which can be used directly in controlMixer() to command unstabilized signals to the motors or servos.

8.7.1 Important Variables

Variable Name	Modifiable?	Type	Notes
roll_des, pitch_des, yaw_des	No	float	Desired angle in degrees or rate in degrees/second. Constrained within +/- maxAXIS for each
thro_des	No	float	Normalized desired throttle between 0 and 1
roll_passthru, pitch_passthru, yaw_passthru	No	float	Pass-through command variables. Constrained within +/-0.5

Table 7: Important variables related to the getDesState() function.

8.8 controlRATE()

This function contains a simple PID rate controller implementation and generates stabilized axis variables roll_PID, pitch_PID, and yaw_PID. Error is computed from the difference in the desired axis rate (roll_des, pitch_des, and yaw_des) and the actual angular rate (GyroX, GyroY, and GyroZ). Controller gains P, I, and D for the rate controller and each axis are adjusted in the user specified variable declaration section. Please note that regardless of control method selected, the yaw axis is always stabilized with the rate controller.

Two safety features are implemented within this function regarding the integral (I) terms to prevent excessive buildup which can lead to unsafe motor command generation. First, the I term for each axis is defaulted to zero for low to minimum throttle settings, meaning that the I terms will not build up while the vehicle is on the ground and throttle is set to minimum. Otherwise, small error can build up over time, leading to one or more of the motors beginning to spin up even if minimum throttle is commanded by the transmitter. Additionally, the I terms are saturated within maximum bounds to prevent them from building up past unsafe levels. Realistically, the I terms should not need to increase more than 25% of the total available throttle range in order to properly stabilize a vehicle. The i_limit variable has been set to prevent this unsafe buildup.

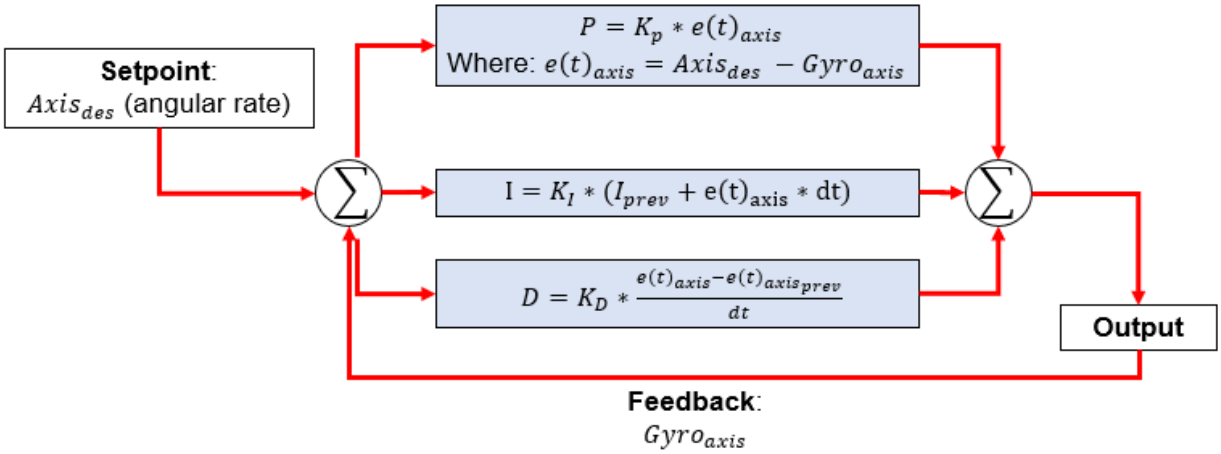

Figure 21: PID controller stabilizing on angular rate setpoint.

8.8.1 Important Variables

Variable Name	Modifiable?	Type	Notes
roll_PID, pitch_PID, yaw_PID	No	float	Stabilized variables for each axis from the PID controller
Kp_AXIS_rate	Yes	float	Proportional gain for each axis of the rate controller
Ki_AXIS_rate	Yes	float	Integral gain for each axis of the rate controller
Kd_AXIS_rate	Yes	float	Derivative gain for each axis of the rate controller
i_limit	Yes	float	Integral term saturation level to prevent unsafe integral buildup

Table 8: Important variables related to the controlRATE() function.

8.9 controlANGLE()

This function contains a simple PID angle controller implementation and generates stabilized axis variables roll_PID, pitch_PID, and yaw_PID. Error is computed from the difference in the desired axis angle (roll_des, pitch_des) and the actual angle (roll_IMU, pitch_IMU). Controller gains P, I, and D for the angle controller and each axis are adjusted in the user specified variable declaration section. Please note that regardless of control method selected, the yaw axis is always stabilized with the rate controller.

Two safety features are implemented within this function regarding the integral (I) terms to prevent excessive buildup which can lead to unsafe motor command generation. First, the I term for each axis is defaulted to zero for low to minimum throttle settings, meaning that the I terms will not build up while the vehicle is on the ground and throttle is set to minimum. Otherwise, small error can build up over time, leading to one or more of the motors beginning to spin up even if minimum throttle is commanded by the transmitter. Additionally, the I terms are saturated within maximum bounds to prevent them from building up past unsafe levels. Realistically, the I terms should not need to increase more than 25% of the total available throttle range in order to properly stabilize a vehicle. The i_limit variable has been set to prevent this unsafe buildup.

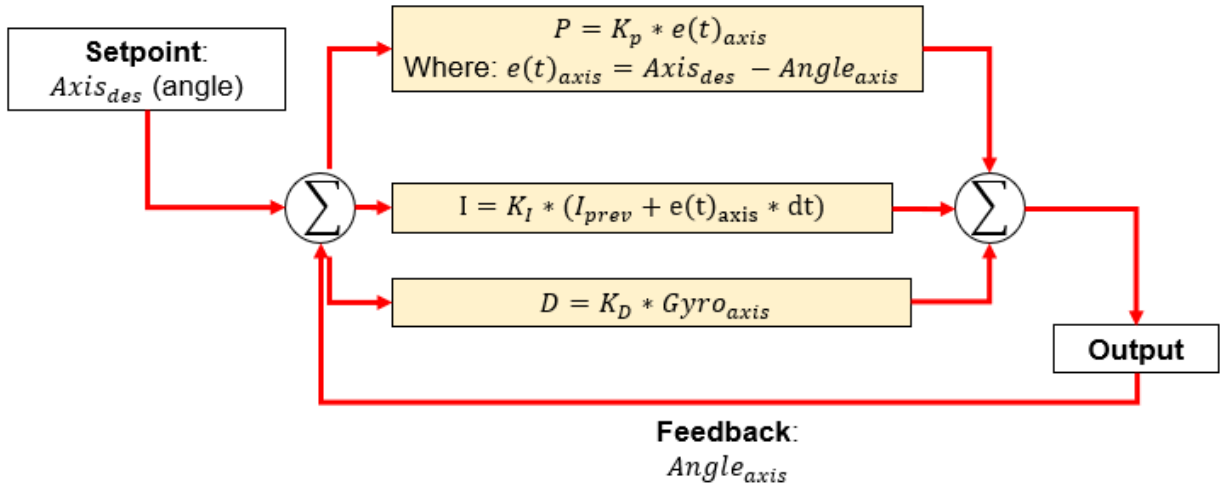

Figure 22: PID controller stabilizing on angle setpoint.

8.9.1 Important Variables

Variable Name	Modifiable?	Type	Notes
roll_PID, pitch_PID, yaw_PID	No	float	Stabilized variables for each axis from the PID controller
Kp_AXIS_angle	Yes	float	Proportional gain for each axis of the angle controller
Ki_AXIS_angle	Yes	float	Integral gain for each axis of the angle controller
Kd_AXIS_angle	Yes	float	Derivative gain for each axis of the angle controller
i_limit	Yes	float	Integral term saturation level to prevent unsafe integral buildup

Table 9: Important variables related to the controlANGLE() function.

8.10 controlANGLE2()

This function contains an advanced cascaded PID angle controller implementation and generates stabilized axis variables roll_PID, pitch_PID, and yaw_PID. A P-I controller on angle serves as the outer loop controller, which generates requested rate commands for the standard PID rate controller inner loop. These requested rate commands are filtered with a first order low pass filter which serves as damping of the angle controller.

Error in angle is computed from the difference in the desired axis angle (roll_des, pitch_des) and the actual angle (roll_IMU, pitch_IMU). Controller gains P and I for the angle controller and each axis are adjusted in the user specified variable declaration section. Damping of the angle controller is adjusted with the B_loop_roll and B_loop_pitch parameters, where decreasing this parameter gives more damping but a slower response.

Error in rate is computed from the difference in the desired axis rate (roll_des, pitch_des) and the actual angular rate (GyroX, GyroY). Controller gains P, I, and D for the rate controller and each axis are adjusted in the user specified variable declaration section. Please note that regardless of control method selected, the yaw axis is always stabilized with the rate controller.

Two safety features are implemented within this function regarding the integral (I) terms to prevent excessive buildup which can lead to unsafe motor command generation. First, the I term for each axis is defaulted to zero for low to minimum throttle settings, meaning that the I terms will not build up while the vehicle is on the ground and throttle is set to minimum. Otherwise, small error can build up over time, leading to one or more of the motors beginning to spin up even if minimum throttle is commanded by the transmitter. Additionally, the I terms are saturated within maximum bounds to prevent them from building up past unsafe levels. Realistically, the I terms should not need to increase more than 25% of the total available throttle range in order to properly stabilize a vehicle. The i_limit variable has been set to prevent this unsafe buildup.

To tune this controller, it is first recommended to tune the standard rate controller, as the inner loop of this controller is identical to the standard rate controller and uses the same controller gain variables. Then, the angle gains can be adjusted appropriately. Because of the added complexity and required tuning, this controller is not recommended for novices or first-time setup.

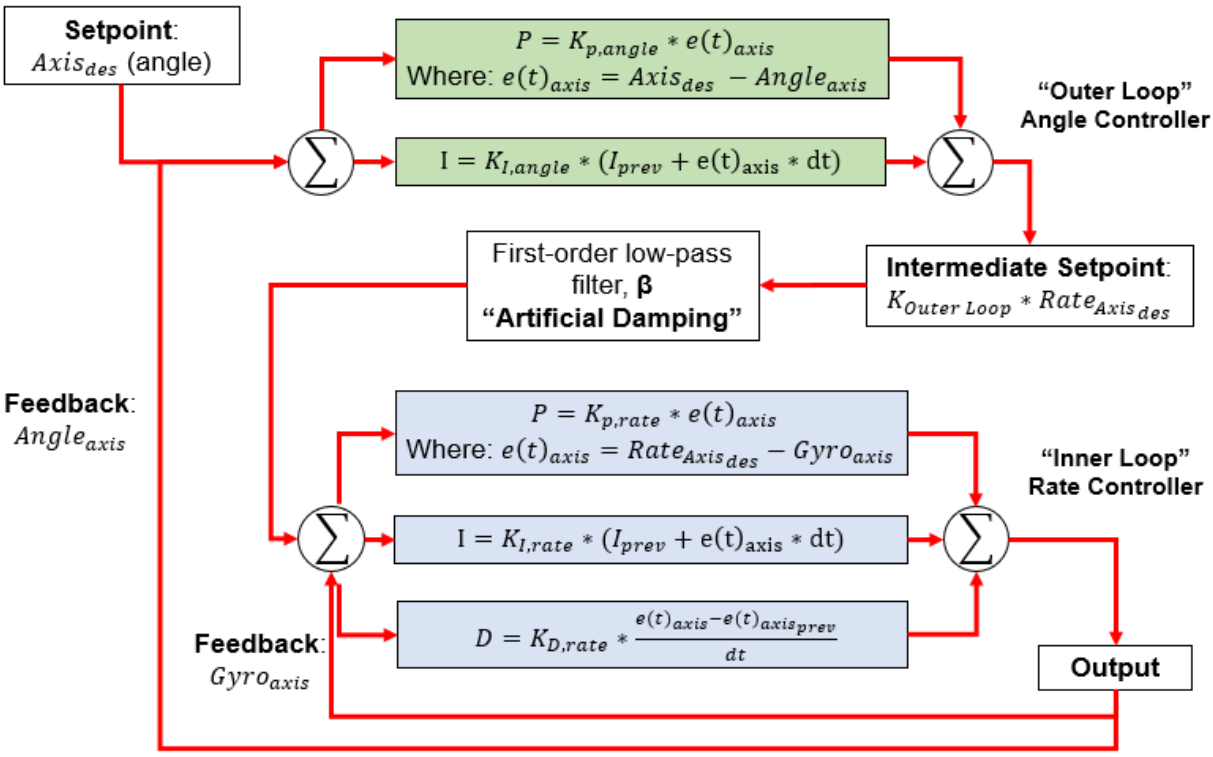

Figure 23: Cascaded PID controller stabilizing on angular rate setpoint (inner loop) generated by outer loop angle controller.

8.10.1 Important Variables

Variable Name	Modifiable?	Type	Notes
roll_PID, pitch_PID, yaw_PID	No	float	Stabilized variables for each axis from the PID controller
Kp_AXIS_angle, Kp_AXIS_rate	Yes	float	Proportional gain for each axis of the angle and rate controllers
Ki_AXIS_angle, Ki_AXIS_rate	Yes	float	Integral gain for each axis of the angle and rate controllers
Kd_AXIS_rate	Yes	float	Derivative gain for the rate controller
B_loop_roll, B_loop_pitch	Yes	float	Outer loop angle damping. Minimum 0, maximum 1. Decrease for more angle damping, increase for less.
i_limit	Yes	float	Integral term saturation level to prevent unsafe integral buildup

Table 10: Important variables related to the controlANGLE2() function.

8.11 controlMixer()

This function mixes the stabilized axis variables roll_PID, pitch_PID, and yaw_PID from the selected PID controller (as well as the scaled throttle thro_des) and creates 'scaled' actuator variables mX_command_scaled and sX_command_scaled which have a magnitude near 0 to 1. The default code provided includes mixing for a standard quadrotor configuration to the first four motors. The stabilized axis variables can also be applied to 'scaled' variables for commanding servos using conventional PWM control. Direct commands from the transmitter can be passed through to the scaled motor or servo variables using the roll_passthru, pitch_passthru, and yaw_passthru variables. These scaled variables are re-assigned to variables with proper magnitude for writing out to the pins in scaleCommands(). For example, mX_command_scaled in a range between 0 to 1 must be scaled within 125 to 250 if using OneShot125 protocol to command an ESC, or between 0 to 180 if using a servo library for standard PWM control.

roll_PID, pitch_PID, yaw_PID, and thro_des are assigned here according to the desired vehicle configuration which is fully customized to the user's needs. For example on a quadrotor, the left two motors should have +roll_PID assigned to them while the right two motors should have -roll_PID assigned to them. The front two should have -pitch_PID and the back two should have +pitch_PID, etc. To check if the stabilized variable assignment is correct for your configuration, tilting the vehicle about an axis should lead to the motors on the falling side to briefly spin up in response. If the opposite is true, the sign of the stabilized axis variable assignment should be reversed for the incorrect motors. If this is correct, but commanding a right roll leads to the vehicle wanting to roll left for example, then the roll stick in the transmitter is reversed. In code, this can be solved with a negative sign in front of any of the roll_des, pitch_des, or yaw_des variables. Following proper radio setup, these should all be correct, though there may be a case with unique

vehicle configurations which have variable dynamics, where an axis of control may need to be flipped by monitoring the state of an auxiliary radio channel. This is done at the user's discretion.

8.11.1 Important Variables

Variable Name	Modifiable?	Type	Notes
m1_command_scaled, s1_command_scaled, ...	Yes	float	Can create additional variables for use case. Typical range is ∼0 to 1

Table 11: Important variables related to the controlMixer() function.

8.12 scaleCommands()

This function creates actuator command variables mX_command_PWM from the mX_command_scaled variables and sX_command_PWM from the sX_command_scaled variables. These are scaled to be within the proper range for writing out to the designated motor or servo pins. For example, if using OneShot125 to control the motors, a range of 125 to 250 is used, or if using a conventional servo library, a range of 0 to 180 is used. The commands are scaled to their respective range and then constrained so that any error in the code leading up to this step does not generate invalid commands that will ultimately be written out to the pins. By default, this function contains 6 variables for ESCs using OneShot125 protocol and 7 variables for servos or ESCs using conventional 50 Hz PWM. If additional actuators are created in the controlMixer() function, corresponding actuator command variables should be created here as well.

8.12.1 Important Variables

Variable Name	Modifiable?	Type	Notes
m1_command_PWM, s1_command_PWM, ...	No	int	Can create additional variables for use case. Must use proper range for writing out to pins

Table 12: Important variables related to the scaleCommands() function.

8.13 throttleCut()

This function is called before writing out the mX_command_PWM variables to the respective pins, where the state of the channel 5 auxiliary switch is monitored for motor arming functionality. If the switch is high (greater than 1500 PWM pulse), then all of the mX_command_PWM variables are overwritten to low values and the motors will not spin. This safety feature ensures that the flight control loop is always checking if the user has given permission via channel 5 of the radio to send command pulses to the motors. Please note that the default failsafe value for channel 5 should remain greater than 1500, so that if an error in the radio communication occurs, throttleCut() defaults to disarming the motors. If additional motors have been created and assigned in the controlMixer() and scaleCommands() functions, or the servo variables are being used to command motors using conventional PWM, they should be added here with proper low values as well.

8.13.1 Important Variables

There are no important variables pertaining to this function.

8.14 commandMotors()

This function is used to command ESCs using OneShot125 protocol. The mX_command_PWM variables scaled to 125 to 250 give the time in microseconds that the respective motor pin should go high for. To do this, all motors are set to digital high, and then a loop runs until the proper time for each motor pin has passed, where that pin is written low and a flag variable indicates this has occurred. When every pin has successfully written low, the loop exits. If additional motors are desired to be commanded other than the default 6, please follow the pattern of this function to expand to the desired number of motors.

8.14.1 Important Variables

There are no important variables pertaining to this function.

8.15 getCommands()

This function gets the most current radio commands as channel_X_PWM, where X is the channel number. The simple getRadioPWM() utility function contained within the radioComm file is called for each channel is using a PWM or PPM type receiver. This function takes the channel number as input and returns the most recent channel_X_raw variable, which are being continuously updated from the interrupt service routines setup in radioSetup(). The current channel_X_PWM variables are then filtered using a simple low pass filter to prevent any noise spikes from interfering with the code.

8.15.1 Important Variables

Variable Name	Modifiable?	Type	Notes
channel_1_PWM, channel_2_PWM, ...	No	unsigned long	Expected range between 1000 to 2000

Table 13: Important variables related to the getCommands() function.

8.16 failSafe()

This function checks the radio commands channel_X_PWM for erroneous values and sets all of them to default failsafe values if any are detected. If any of them exceed expected bounds, failsafe values channel_X_fs established in the user-specified variable section are instead used. Please note that the default failsafe values correspond to: minimum throttle, centered aileron, elevator, and yaw sticks, and channel 5 and 6 auxiliary switches set to 2000. Also note that channel 5 is used for throttle cut functionality, and the default failsafe value for this channel is set above 1500 to enable throttle cut if radio connection is disturbed. For safety reasons, this should not be modified.

8.16.1 Important Variables

Variable Name	Modifiable?	Type	Notes
channel_1_fs	Yes	unsigned long	Default 1000
channel_2_fs	Yes	unsigned long	Default 1500
channel_3_fs	Yes	unsigned long	Default 1500
channel_4_fs	Yes	unsigned long	Default 1500
channel_5_fs	Yes	unsigned long	Default 2000 (enables throttle cut when loss of radio signal detected)
channel_6_fs	Yes	unsigned long	Default 2000

Table 14: Important variables related to the failSafe() function.

8.17 floatFaderLinear()

This function is a simple utility function which linearly interpolates, or fades, a specified variable between two high and low values. Rather than hard changes in parameters such as servo locations or controller gains between flight modes, linearly interpolating with this function will give a smoother transition. Please note that the variable must be of float type to properly work with this function. Inputs to the function include the variable to be modified (such as controller gains, or direct scaled motor or servo commands), a minimum bound on that variable, maximum bound, time to complete the fade between the bounds in seconds, current desired state (0 for minimum bound, 1 for maximum bound), and loop rate in Hz (default code is 2000 which should be used unless changed in calling loopRate()). The function returns the modified variable for the current loop iteration. The state of either 0 or 1 corresponds to the desired value of the parameter, either the minimum specified or the maximum specified. This function is intended to be called within controlMixer() where changes in dynamic configuration may be taking place, and may be toggled within a logical statement monitoring the state of an auxiliary radio channel. An example of this use case is included in controlMixer().

8.17.1 Important Variables

There are no important variables pertaining to this function.

8.18 switchRollYaw()

This function switches the roll_des and yaw_des variables. The intended use case for this function is for tailsitter-type configurations where the roll and yaw axis switch between hover and forward flight. This function takes two integers (either -1 or 1) as inputs, corresponding to the desired reversing of the output roll_des and yaw_des variables. For example, switchRollYaw(-1, 1) will reverse the output roll_des variable (now the original yaw_des, but reversed), but not the output yaw_des variable (now the original roll_des). While the majority of control mixing for transition vehicles is done in the controlMixer, this function must be called right before the selected PID controller in the main loop in it's own conditional statement.

Please note that this function may be replaced in a future release by a function that switches the IMU data orientation so that the attitude can still be estimated even when the IMU is tilted at 90 degrees, allowing the use of the angle controllers in both flight modes.

8.18.1 Important Variables

There are no important variables pertaining to this function.

8.19 loopRate()

This function takes in a desired loop rate in Hz and idles at the end of the main flight controller loop until this rate has been achieved, maintaining a constant loop rate. The default loop rate is set to 2000 Hz, and all filtering parameters by default are tuned to this frequency. The code is able to run above 3000 Hz, but IMU data is not available at this rate and filtering parameters would need to be re-tuned. The selected rate gives excellent performance and leaves plenty of additional computing power to remain well within 2000 Hz if additional/custom code is implemented.

8.19.1 Important Variables

There are no important variables pertaining to this function.

8.20 setupBlink()

This is a simple utility function for the void setup() to indicate specific operations have occurred with a prescribed blink on the Teensy's onboard LED. By default, this function is only called once at the end of the void setup() to indicate the code is entering the main loop with 3 quick blinks. It requires 3 values be passed into it: the number of desired blinks, the time in ms to be on, and the time in ms to be off for the blinks. Please note that pin 13 on the Teensy 4.0 is reserved for the onboard LED and should not be used for any other purpose.

8.20.1 Important Variables

There are no important variables pertaining to this function.

8.21 loopBlink()

This is a simple utility function for the void loop() to indicate that the loop is properly running on the Teensy's onboard LED. By default, a quick blink every 1.5 seconds indicates the main loop is running. Please note that pin 13 on the Teensy 4.0 is reserved for the onboard LED and should not be used for any other purpose.

8.21.1 Important Variables

There are no important variables pertaining to this function.

8.22 Print Statements for Troubleshooting

There are multiple functions used for troubleshooting and ensuring proper operation of the flight controller for first time setup. These functions print data to the serial monitor (or plotter) at 100 Hz and all follow the same formatting, which can be borrowed for additional troubleshooting print statements.

8.22.1 printRadioData()

Variable Name	Expected Value Range	Location Where Generated/Modified
channel_1_pwm, channel_2_pwm, ...	1000 to 2000	getCommands() function in main loop

Table 15: Variables printed to the serial monitor in the printRadioData() function.

8.22.2 printDesiredState()

Variable Name	Expected Value Range	Location Where Generated/Modified
thro_des	0 to 1	getDesState() function in main loop
roll_des	-maxRoll to maxRoll	getDesState() function in main loop
pitch_des	-maxPitch to maxPitch	getDesState() function in main loop
yaw_des	-maxYaw to maxYaw	getDesState() function in main loop

Table 16: Variables printed to the serial monitor in the printDesiredState() function.

8.22.3 printGyroData()

Variable Name	Expected Value Range	Location Where Generated/Modified
GyroX, GyroY, GyroZ,	∼-250 to 250, zero at rest	getIMUdata() function in main loop

Table 17: Variables printed to the serial monitor in the printGyroData() function.

8.22.4 printAccelData()

Variable Name	Expected Value Range	Location Where Generated/Modified
AccX, AccY	∼-2 to 2, zero at rest	getIMUdata() function in main loop
AccZ	∼-2 to 2, 1 at rest	getIMUdata() function in main loop

Table 18: Variables printed to the serial monitor in the printAccelData() function.

8.22.5 printMagData()

Variable Name	Expected Value Range	Location Where Generated/Modified
MagX, MagY, MagZ,	∼-400 to 400	getIMUdata() function in main loop

Table 19: Variables printed to the serial monitor in the printMagData() function.

8.22.6 printRollPitchYaw()

Variable Name	Expected Value Range	Location Where Generated/Modified
roll_IMU, pitch_IMU, yaw_IMU	-90 to 90	Madgwick() function in main loop

Table 20: Variables printed to the serial monitor in the printRollPitchYaw() function.

8.22.7 printPIDoutput()

Variable Name	Expected Value Range	Location Where Generated/Modified
roll_PID, pitch_PID, yaw_PID	∼-1 to 1	controlANGLE(), controlANGLE2(), or controlRATE() function in main loop

Table 21: Variables printed to the serial monitor in the printPIDoutput() function.

8.22.8 printMotorCommands()

Variable Name	Expected Value Range	Location Where Generated/Modified
m1_command_PWM, m2_command_PWM, m3_command_PWM, m4_command_PWM m5_command_PWM, m6_command_PWM	120 to 250 (OneShot125)	scaleCommands() function in main loop

Table 22: Variables printed to the serial monitor in the printMotorCommands() function.

8.22.9 printServoCommands()

Variable Name	Expected Value Range	Location Where Generated/Modified
s1_command_PWM, s2_command_PWM, s3_command_PWM, s4_command_PWM s5_command_PWM, s6_command_PWM, s7_command_PWM	0 to 180	scaleCommands() function in main loop

Table 23: Variables printed to the serial monitor in the printMotorCommands() function.

8.22.10 printLoopRate()

Variable Name	Expected Value Range	Location Where Generated/Modified
dt * 1,000,000	500 (for 2000Hz loop rate) *should remain below 500 when main loop is unregulated with loopRate()	Top of main loop

Table 24: Variables printed to the serial monitor in the printLoopRate() function.

9 Tutorials

Please note that for any of these tutorials, while testing functionality or modifying the code, motors should have propellers removed or be unplugged entirely for safety.

9.1 General Instructions for First-Time Setup

This section outlines the road map of required tasks contained within this document to go from absolute beginner to this code to flying. This represents the absolute minimum required effort to get flying, but it is recommended to read the entire documentation first to get a feel for dRehmFlight before trying to work with it.

1. Hardware Requirements: Ensure that you have the required hardware components to assemble the basic flight controller. Details can be found in the Hardware Requirements section of this document.

2. Hardware Setup: Follow the default recommended hardware setup procedures, outlined in the Hardware Setup section of this document.

3. Mounting the Flight Controller: Follow the Mounting the IMU to Your Aircraft tutorial in the Tutorials section of this document.

4. Setting Up Your Radio Connection: Follow the Setting Up Your Radio Connection tutorial in the Tutorials section of this document.

5. Verifying Good IMU Data: Follow the Verifying Good IMU data tutorial in the Tutorials section of this document.

6. Control Mixing: It is finally time to customize the flight control code for your specific application. Follow the three Control Mixing tutorials contained within the Tutorials section of this document. Take note of the motor and servo assignments you make for each variable and the corresponding pin number for the default setup in Table 1.

7. Verify Your Code Changes: Using the print statements for troubleshooting (details contained within the Functions List section of this document), verify that important variables such as motor or servo commands are behaving as expected in the Arduino serial monitor.

8. Plug In Your Hardware: Plug in the ESCs and servos according to the assignments you made in step 6.

9. Select the Controller Type and Tune: Follow the Controller Selection and Tuning Tutorial in the Tutorials section of this document. Here, the controller type (rate, basic angle, or advanced angle) is selected and steps to tune for your vehicle are outlined.

These steps represent the basic steps needed to get up and running as quickly as possible. It is highly recommended to read the full documentation to gain a complete understanding of the code in its entirety so that you will be equipped to modify it in any way required for your application.

9.2 Pin Selection

For a custom hardware setup different from the default recommended setup for hobbyists, choosing pins on the board is an important process. The following procedure for an example hardware setup is used to demonstrate the process of proper pin selection, documentation, and modification of the code.

For this example, a tricopter configuration is required. Three OneShot125 ESCs are used to control the motors, and 2 servos are required: one for tilting one of the motors for yaw control, and another for a simple payload drop. In addition to the default IMU and selected PPM receiver, an analog pressure sensor needs to interface with the board, as well as an optical flow sensor that communicates with the board over a serial connection, and finally a landing light system that is triggered by a digital high or low signal. Rather than modify the existing default hardware setup for this use case, a custom setup is desired that will allow for many more sensors and actuators in the future if needed.

The first step is to establish all of the required inputs/outputs and their functionality so that a pin with correct functionality can later be assigned. Table 25 has compiled the information for this example setup.

Component	Requirement
IMU	I2C Bus - SCL and SDA
PPM Receiver	Single digital pin
Motor1	Digital pin
Motor2	Digital pin
Motor3	Digital pin
Tilt Servo	Digital PWM pin
Drop Servo	Digital PWM pin
Pressure Sensor	Analog pin
Optical Flow Sensor	Serial RX pin
Landing Lights	Digital pin

Table 25: Establishing required pin functionality for the example case.

Next, pins must be assigned that can fulfill these requirements. For this case, we will arbitrarily attempt to reserve as many analog and serial pins as possible (perhaps more analog pressure sensors or additional sensors using a serial connection are planned for later). The first pins assigned are the I2C bus pins reserved for the IMU, which are on pins 18 and 19. This I2C bus is the default within the code (so it will not require modification), though it does take up 2 analog pins. All I2C

pins lie on an analog pin on the board, so this sacrifice of 2 analog pins must be made. Next, pin 12 is semi-arbitrarily selected for the PPM receiver pin in order to avoid any analog or serial pins. Similarly, three digital pins (2, 3, and 4) are reserved for the motors. Two PWM-enabled digital pins are also allocated to the servos on pins 5 and 6. Analog pin 23 is semi-arbitrarily selected for the analog pressure sensor, and the optical flow sensor is assigned to RX1 and TX1 on pins 0 and 1. Finally, the digital landing light pin is assigned to 9, avoiding RX2 and TX2, as well as any analog pins which we have set out to preserve. Table 26 shows the final pin selection, and Figure 24 shows the physical pin location on the board. Please note that in addition to these pin assignments, the board must be powered on the 5 V Vin pin and grounded on any ground pin. The Teensy 4.0 has many more pins on the back side pads, however these are difficult to solder and otherwise access, so they were ignored for this example. They may be considered for more advanced setups with far more i/o requirements and a user who is proficient at soldering.

Component	Requirement	Pin Assignment
IMU	I2C Bus - SCL and SDA	SCL0 19, SDA0 18
PPM Receiver	Single digital pin	12
Motor1	Digital pin	2
Motor2	Digital pin	3
Motor3	Digital pin	4
Tilt Servo	Digital PWM pin	5
Drop Servo	Digital PWM pin	6
Pressure Sensor	Analog pin	23
Optical Flow Sensor	Serial RX and TX pin	RX1 0, TX1 1
Landing Lights	Digital pin	9

Table 26: Selecting pins for the example case.

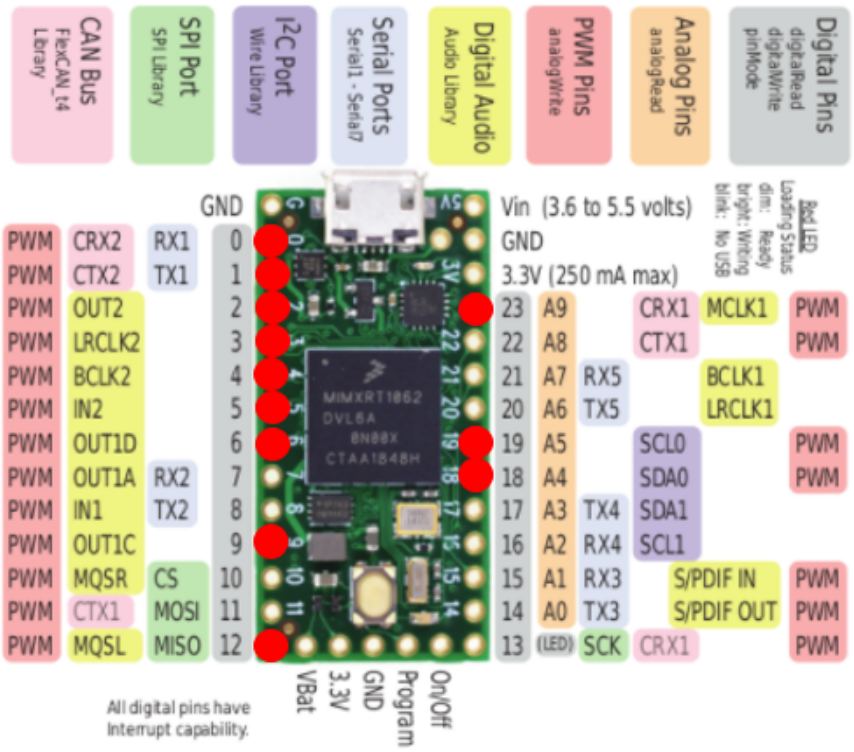

Figure 24: Selected pins on the Teensy 4.0 for the example case.

Now that the analog pins and serial pins have been carefully reserved and every i/o requirement for the vehicle has been met, the selected pin assignments need to be updated in the code's pin declaration section. Additional motors or servos not used may be commented out in the pin declaration section as well as throughout the code where command variables for those extra actuators remain. These can be found in the void setup() section (comment out the pinmode declarations for the extra servos and motors as well as servo object creating and writing), in controlMixer(), scaleCommands(), throttleCut(), and commandMotors().

9.3 Setting Up Your Radio Connection

After successful completion of the physical flight controller assembly, the first thing to do is to compile and upload code to it and check for proper connection to the radio receiver.

First, in the user-specified defines section of the code, uncomment your receiver type: PWM, PPM, or SBUS. Whichever ones not being used should remain commented out. Next, in the top of the void loop(), uncomment the printRadioData() function and ensure that no other printing functions are uncommented. The code is now ready to be uploaded to the board.

Ensure that the receiver is properly connected to the flight controller assembly, either following the suggested default setup, or your custom setup. Please follow the proper channel convention (Table 2) if using a standard PWM receiver, as pins can be plugged in and interchanged to get the correct functionality. If using a PPM or SBUS receiver and you are unable to change the channel mapping within your radio according to Table 2, the end of this section will describe how to adjust the code for proper operation.

Once all connected and the code has been uploaded to the powered board, open the Arduino

serial monitor and you should see the 6 PWM channels on screen (this usually takes about 5-6 seconds for the code to finally begin running the main loop as indicated by a short blink every 1.5 seconds). If the serial monitor looks like Figure 25 and moving the transmitter sticks does not change any of the values, then there is a problem with the connection between the receiver and flight controller. The displayed values are the default failsafe values, which failSafe() is writing to the radio channel variables channel_X_PWM after detecting a bad radio connection. Check that the receiver is properly powered, bound to the transmitter, and all connecting wires between the receiver and flight controller have continuity.

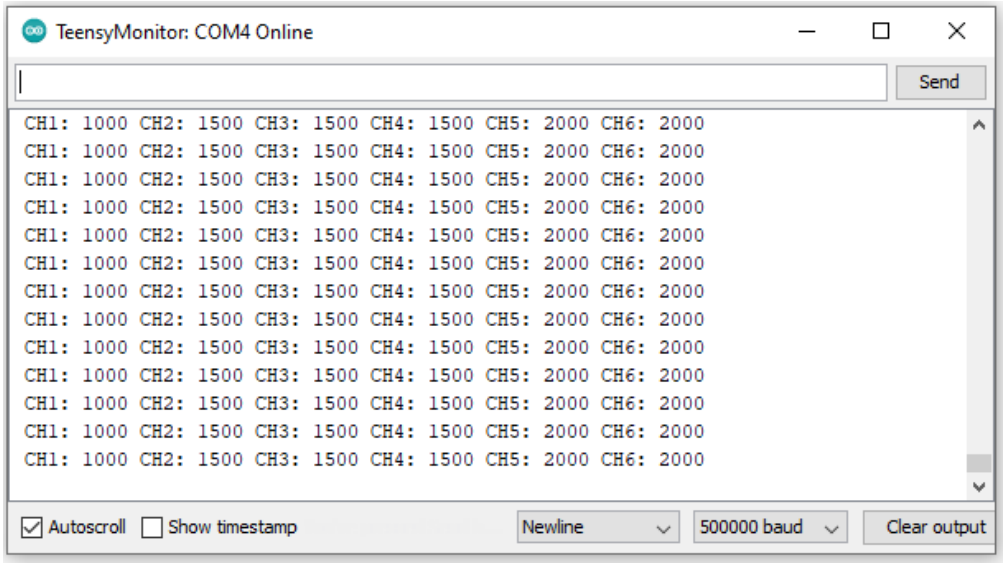

Figure 25: Arduino serial monitor showing radio data where there is an issue with the connection and radio failsafe is being triggered.

If the radio is properly connected, the serial monitor should look similar to Figure 26 where values are not yet centered, but they respond to transmitter stick movement. First, check that each channel corresponds to the correct functionality in Table 2 i.e. CH1 should respond to throttle stick input, CH2 should respond to roll input, CH3 should respond to pitch input, CH4 should respond to yaw input, and CH5 and CH6 should respond to the auxiliary switch inputs. If this is not the case, change the physical pin placement for the channels for a PWM type receiver. The end of this section details how to correct this for a PPM or SBUS type receiver.

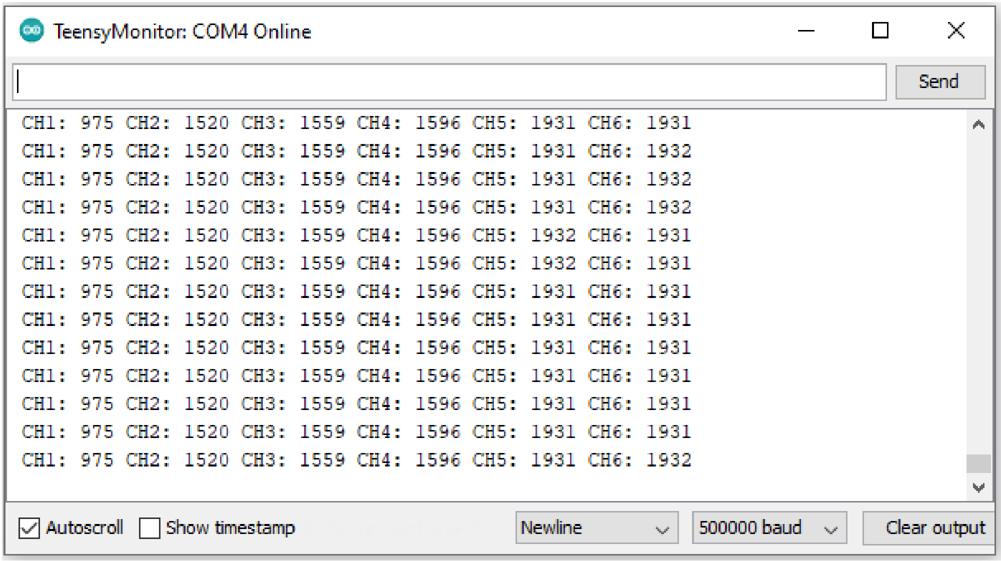

Figure 26: Arduino serial monitor showing radio data where the radio is properly connected but channels are not properly centered.

Next, verify correct direction for each channel and reverse incorrect channels within the radio manually. CH1 (throttle) should increase in value with increasing throttle. CH2 (roll) should increase in value with right roll input. CH3 (pitch) should increase in value with pitch down (stick upwards) input. CH4 (yaw) should increase in value with left yaw input. Please take note of your radio's auxiliary switch assignment for CH5 and CH6–by default, channel 5 is used for throttle cut when above 1500, and channel 6 is a free channel. You may wish to reverse either of these or reassign physical switches within your radio for more comfortable operation.

Next, center all sticks on the transmitter including throttle and apply subtrim within your radio to each channel to center Ch1-4 on 1500. It will be difficult to get it perfect with your radio's resolution, but within 5 is close enough. CH5 and CH6 are used as 'binary' switches (or sometimes channel 6 can be a 3 position switch or even a fader) in which there is no real 'center.' Just be sure that they have a definitive 'high' (above 1500) and a definitive 'low' (below 1500).

Once correct direction corresponding to stick movement has been verified and everything has been centered, it is time to adjust endpoints for each channel within your radio. Adjust each channel such that the minimum value is 1000, and the maximum value is 2000. Within 5 is close enough. Figure 27 shows the Arduino serial monitor after the radio has been properly setup with the throttle stick in the lowest position.

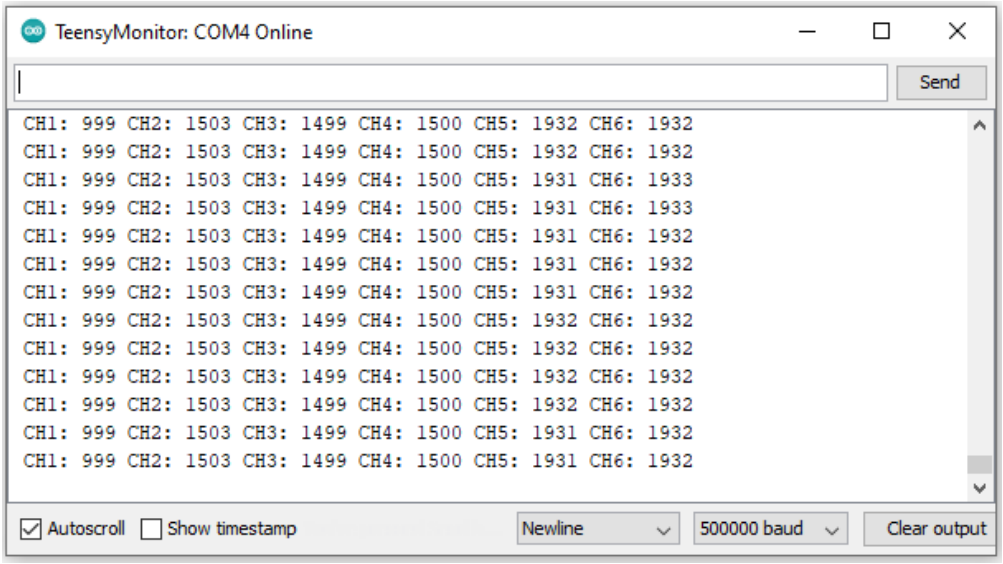

Figure 27: Arduino serial monitor showing radio data where the radio is properly connected and all channels have been adjusted correctly.

If using a PPM or SBUS receiver and the channel mapping is not correct as indicated by incorrect values in the serial monitor for certain stick movement, the code will need to be modified. In the getCommands() function, the order in which radio channels are retrieved can be changed accordingly to match the correct channel mapping given in Table 2.

9.4 Verifying Good IMU Data

After verifying good and proper communication with the radio receiver, the next step for first time setup is to verify proper functionality of the IMU.

First, in the user-specified defines section of the code, uncomment your IMU type: MPU6050 or MPU9250. Whichever ones not being used should remain commented out. You may also select the gyro and accelerometer sensitivity ranges. The default gyro and accelerometer sensitivity values will give the cleanest attitude estimate, but if a higher maximum angular velocity is needed (for example on an aerobatic aircraft), then a higher value can be chosen. A higher accelerometer value may also be needed if flying an aircraft that undergoes very high accelerations.

Next, in the top of the void loop(), uncomment the printGyroData() function and ensure that no other printing functions are uncommented. Ensure that the IMU is properly connected to the flight controller assembly following the suggested default setup and then compile and upload the code to the powered flight controller. The board's onboard LED will turn on to indicate that it is calibrating and warming up the IMU. Do not touch the IMU until 3 quick blinks indicate that the flight controller is beginning the main flight control loop, indicated by a quick blink every 1.5 seconds. Open the serial plotter or monitor and verify that all three axis of the IMU's gyro are properly zeroed (or at least within 5 degree/sec of 0). There will be a small amount of noise but that is okay. Rotate the IMU about each axis following the convention in Figure 16 and verify that the values roughly correspond to your movement.

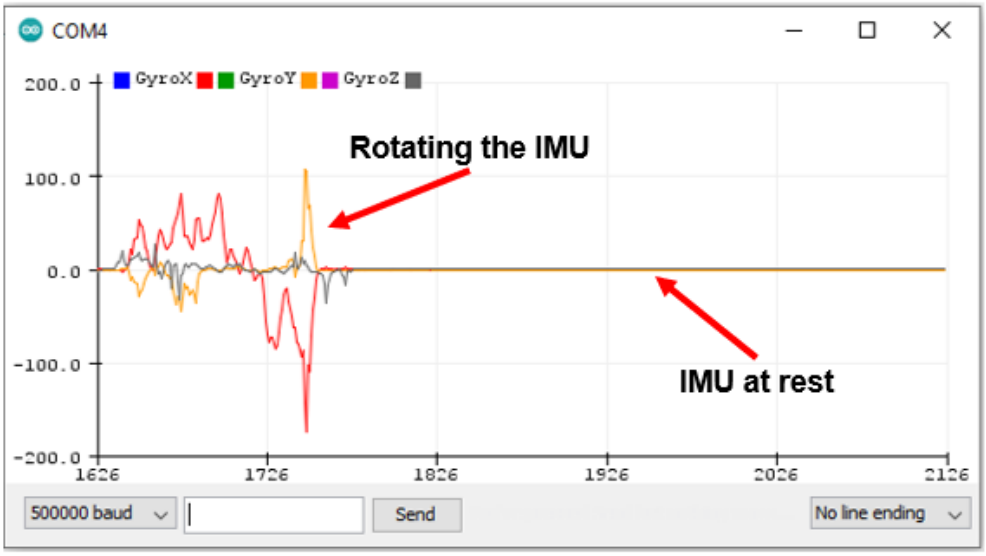

Figure 28: Arduino serial plotter showing proper IMU gyro data.

Next, in the top of the void loop(), uncomment the printAccelData() function and ensure that no other printing functions are uncommented. Level the IMU facing upwards and compile and upload the code to the powered flight controller. Do not touch the IMU until 3 quick blinks indicate that the flight controller is beginning the main flight control loop, indicated by a quick blink every 2 seconds. Open the serial plotter or monitor and verify that the x and y axis of the IMU's accelerometer are properly zeroed (or at least within 0.05 of zero) and that the z axis reads very close to 1 g. If on another planet with different gravity than that of Earth, the z axis will read something different. Support for different planet gravity is sadly not planned for future releases. Move the IMU about each axis following the convention in Figure 16 and verify that the values roughly correspond to your movement.

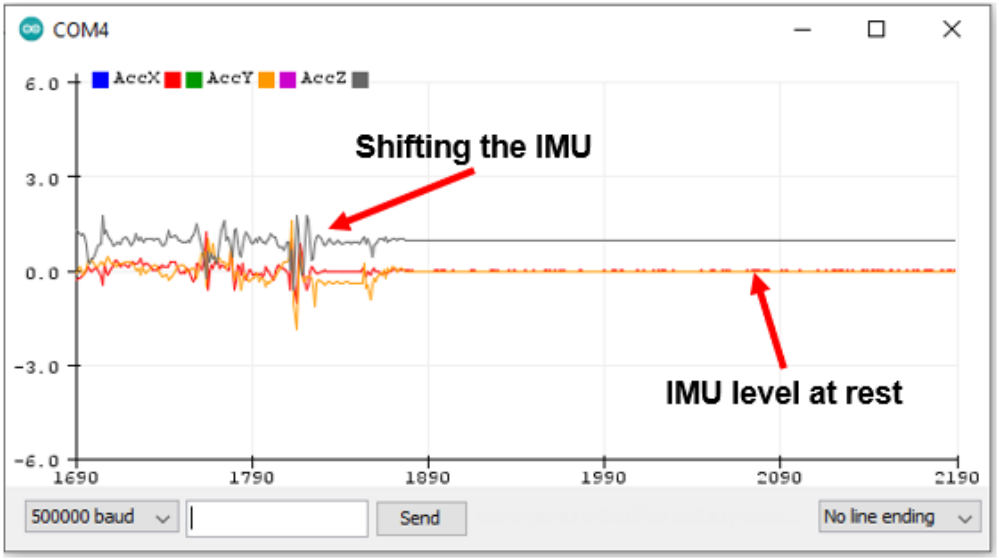

Figure 29: Arduino serial plotter showing proper IMU accelerometer data.

If using the MPU9250 IMU, uncomment the printMagData() function and ensure that no other printing functions are uncommented. Level the IMU facing upwards in an area away from metal or magnetic fields and compile and upload the code to the powered flight controller. Do not touch the IMU until 3 quick blinks indicate that the flight controller is beginning the main flight control loop, indicated by a quick blink every 2 seconds. Open the serial plotter or monitor and verify that there is good data for each magnetometer axis. Next, uncomment the calibrateMagnetomer() function at the end of the void setup(). Compile and upload the code to the board and follow the instructions printed to the serial monitor. Magnetometer calibration parameters will be printed to the serial monitor upon completion which should be added to the code in the user-specified variable section. This calibration process needs to be completed whenever flying in a new area.

Finally, in the top of the void loop(), uncomment the printRollPitchYaw() function and ensure that no other printing functions are uncommented. Level the IMU facing upwards and compile and upload the code to the powered flight controller. Do not touch the IMU until 3 quick blinks indicate that the flight controller is beginning the main flight control loop, indicated by a quick blink every 1.5 seconds. Open the serial plotter or monitor and verify that when the IMU is level, all readings are approximately zero (yaw may slowly drift when using the MPU6050 which is okay). Move the IMU about each axis following the convention in Figure 16 and verify that the angles in degrees roughly correspond to your movement.

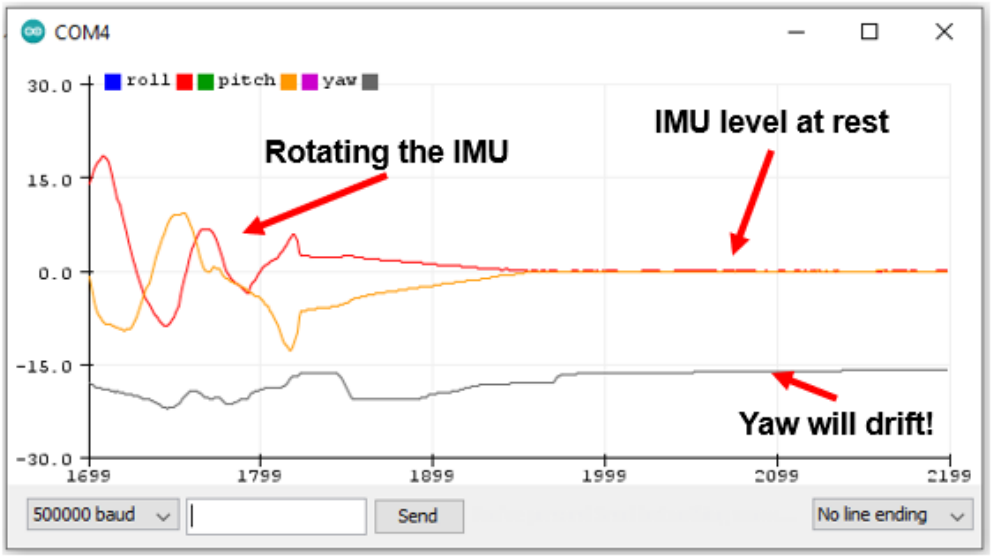

Figure 30: Arduino serial plotter showing proper IMU angle data. When using the MPU6050, the yaw estimate may slowly drift. When using the MPU6050, the yaw estimate will be noisier but should not drift.

Any errors encountered during this process could be the result of many things. The IMU itself may be defective, in which case try a different board. If that does not solve the issue, the wiring or soldering may be faulty–this can be checked with a simple continuity test between the Teensy and the IMU.

9.5 Mounting the IMU to Your Aircraft

It is important to mount the IMU to the aircraft in the correct orientation and in such a way to mitigate vibration. A thick foam double-sided tape is recommended to attach the underside of the IMU to the aircraft. Figure 31 shows this process, where 2-3 layers of foam tape are used to build up a ˜6mm pad under the IMU. Another option is to use hot glue and 6mm foamboard to mount to the IMU, and then the aircraft. Anything that rigidly attaches the IMU to the airframe while still maintaining a 'soft' connection between the two will suffice. If not following the suggested hardware setup where the IMU is wired separately from the Teensy, but rather the two are located on the same breakout board for example, the entire flight controller board must be soft-mounted and in the correct orientation.

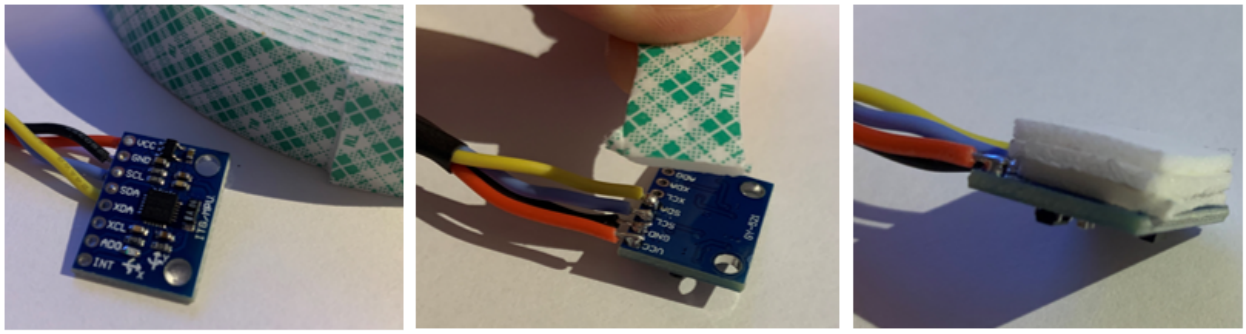

Figure 31: Attaching thick foam tape to the underside of the IMU.

Figure 16 shows the axis convention used for IMU data and vehicle orientation. This should be followed when mounting the IMU to the airframe. Choose a rigid location near the aircraft's center of gravity and mount the IMU using the foam tape or foam and hot glue method described above. Please note the IMU's orientation in Figure 32 where the front of the aircraft is noted and the exposed circuit components are facing up toward the sky. When using the suggested default hardware setup, the rest of the flight controller components can be mounted anywhere else on the aircraft.

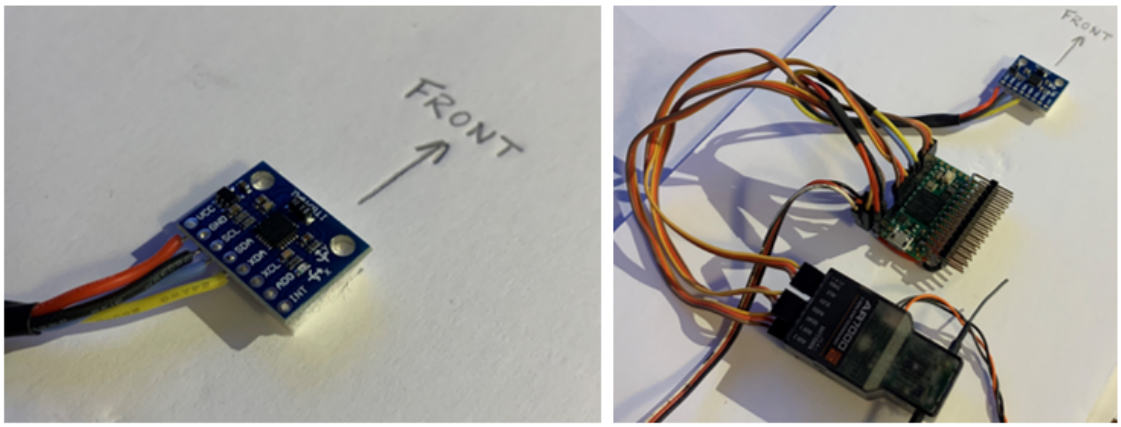

Figure 32: Mounting the IMU and remaining flight controller components in the correct orientation.

9.6 Alternative Actuator Outputs: OneShot125 and Conventional PWM

By default, the code supports 6 ESCs using OneShot125 protocol, and 7 servos/ESCs using conventional PWM control. Please see the suggested default hardware setup section for information regarding the i/o capability of the default code. This section explains the required process to add additional actuator outputs for a custom setup in which the default outputs are not sufficient.

9.6.1 Adding Another OneShot125 Controlled Motor

To add another OneShot125 controlled motor, please follow these steps to modify the code:

1. It is assumed that the pin selection tutorial has been followed and pins are properly reassigned within the code in the pin declaration section first. For this example, the new motor pin variable will be denoted 'm7Pin'.

2. In the bottom of the global variable declaration section, add '...._command_scaled' and '...._command_PWM' variables for the new motor. In this case, they will be named m7_command_scaled and m7_command_PWM.

3. In the top of void setup() in the pin initialization section, initialize the new pin as an output using pinMode(m7Pin, OUTPUT).

4. Toward the bottom of void setup() in the motor arming section, write a value of 125 to m7_command_PWM. This section of the code is used to write a low pulse (125 us) to each motor command variable before writing out to the pins using commandMotors().

5. In controlMixer(), add the m7_command_scaled value to the list of motors, and assign whatever stabilization output is required for that motor.

6. In scaleCommands(), add the m7_command_PWM assignment:

m7_command_PWM = m7_command_scaled*125 + 125;

as well as the constraint command:

m7_command_PWM = constrain(m7_command_PWM, 125, 250);

These two lines are used to transform the newly created ...._command_scaled variable into a ...._command_PWM variable, which is in the proper constrained range to be written out the motor pins.

7. In throttleCut(), write a value of 120 to the new ...._command_PWM value.

8. In commandMotors(), follow the general structure within to expand for the new motor, as it only accommodates 6 motors by default.

9.6.2 Adding a PWM Controlled Servo or Motor

To add another PWM controlled servo or motor, please follow these steps to modify the code:

1. It is assumed that the pin selection tutorial has been followed and pins are properly reassigned within the code in the pin declaration section first. For this example, the new servo PWM pin variable will be denoted 'servo8Pin'.

2. Right below the pin declaration section, add a new PWMServo object, 'servo8'.

3. In the bottom of the global variable declaration section, add '...._command_scaled' and '...._command_PWM' variables for the new servo. In this case, they will be named s8_command_scaled and s8_command_PWM.

4. In the top of void setup() in the pin initialization section, initialize the new pin as an output for the newly created servo object using servo8.attach(servo8Pin, 900, 2100).

5. Toward the bottom of void setup() in the servo arming section, write a value of 0 to the servo8 object using servo8.write(0). This section of the code is used to write a low value to the servos (particularly, the ESCs to arm them) before entering the main loop.

6. In controlMixer(), add the s8_command_scaled value to the list of servos, and assign whatever stabilization output is required for that servo.

7. In scaleCommands(), add the s8_command_PWM assignment:

s8_command_PWM = s8_command_scaled*180;

as well as the constraint command:

s8_command_PWM = constrain(s8_command_PWM, 0, 180);

These two lines are used to transform the newly created ..._command_scaled variable into a ..._command_PWM variable, which is in the proper constrained range to be written out the servo pins.

8. If using the servo to command an ESC using conventional PWM, in throttleCut(), write a value of 0 to the new ..._command_PWM variable.

9. In the actuator command section of the void loop(), write out the ..._command_PWM variable to the new servo object: servo8.write(s8_command_PWM).

9.7 Control Mixing: Basic Fixed Dynamics

Control mixing means to assign correct stabilization of each axis to the actuator outputs so that they all work to correctly stabilize the vehicle. This is done in the controlMixer() function where the thro_des, roll_PID, pitch_PID, and yaw_PID variables are assigned to the ..._command_scaled variables. The ..._command_scaled variables should have magnitude between 0 and 1, since thro_des is scaled from 0 to 1, and with correctly tuned PID values, roll_PID, pitch_PID, and yaw_PID will be well within -1 to 1. Assignment for a standard X quadrotor configuration is provided as an example in the default code, where motor 1 is the front left, motor 2 is the front right, motor 3 is the back right, and motor 4 is the back left:

m1_command_scaled = thro_des - pitch_PID + roll_PID + yaw_PID;
m2_command_scaled = thro_des - pitch_PID - roll_PID - yaw_PID;
m3_command_scaled = thro_des + pitch_PID - roll_PID + yaw_PID;
m4_command_scaled = thro_des + pitch_PID + roll_PID - yaw_PID;

The best way to decide assignment and sign of the stabilized axis variables roll_PID, pitch_PID, and yaw_PID is through trial and error with propellers removed from the motors. If there is a motor on the centerline of the vehicle toward the nose, then it can be assumed that that motor, when spun up and slowed down, will correspond to a response in pitch. Only pitch_PID should be assigned to that motor (in addition to thro_des if that motor should respond to throttle input too). If there are two aileron servos for a conventional fixed wing platform, then one servo should receive +roll_PID and the other -roll_PID. Please note that servos are configured such that 0 is minimum

throttle if being used for a PWM ESC, and 1 is maximum throttle. As a result, an offset of 0.5 should be added to the ..._command_scaled servo variable to center the servo.

After removing propellers from all motors, the flight controller can be powered and the assignments to each actuator checked. For motors, lowering the aircraft toward that motor should cause it to briefly spin up. This indicates that the flight controller is stabilizing in the correct direction. For servos, for example on a fixed wing platform, pitching the aircraft back should cause the elevator to actuate downward. If any of these are reversed, the sign of the assigned stabilized axis variable roll_PID, pitch_PID, or yaw_PID for that actuator should be flipped.

One final sanity check for the assignments involves placing the aircraft on the ground and moving the transmitter sticks. If commanding a right roll for a quadrotor, for example, both of the left motors should spin up. If this is opposite, but the motors stabilize correctly when lowing the quadrotor to the right side, then the aileron stick in the radio is reversed. Please see the radio connection setup tutorial for proper setup of your transmitter.

After proper assignment is verified, the user is free to scale the stabilized axis variables or add custom offsets to each motor or servo. Scaling the assignments by multiplying by a coefficient may be useful in the case of a particular motor or servo requiring less stabilization or total travel. Custom offsets are useful for centering control surfaces as a form of sub trim. It is recommended to create custom variables within the controlMixer() function for these purposes which can quickly and easily be tuned. Please note that the ..._command_scaled variables modified in controlMixer() should always have a value between 0 and 1. This means that any scaling or offsetting should respect that at the risk of saturating a motor to zero or full throttle, or a servo to full deflection. For example, stabilized axis variables should only be scaled between 0 to 1, and offsets should remain between 0 and 1 as well.

9.8 Control Mixing: Unstabilized Commands Direct From the Transmitter

Please thoroughly read and understand the basic fixed dynamics control mixing tutorial first before reading this section.

Direct, unstabilized commands from the transmitter can be applied to the ..._command_scaled variables in controlMixer() as well using the roll_passthru, pitch_passthru, and yaw_passthru variables. For this first example, we will consider direct control of an elevator servo (arbitrarily assigned to servo 5) in a fixed wing type aircraft. The ..._passthru variables are all constrained within +/- 0.5, which means when assigned to the ..._command_scaled variables, they will correctly stay between 0 and 1 when an offset of 0.5 is added. This offset is only needed if commanding a servo variable to center it. Back to the simple elevator servo example, the assignment will look like:

```
float pitchAmount = 0.7;
float pitchOffset = 0.5;
s5_command_scaled = pitchAmount*pitch_passthru + pitchOffset;
```

The pitchAmount and pitchOffset variables are arbitrarily created by the user in controlMixer(). pitchAmount, in this example, will scale the total amount of elevator 'throw'. A value of 1 will max out the servo bounds, and a value of 0.5 will cause the servo to only move half of the total available amount. The pitchOffset variable is used to 'trim' the elevator servo. A value of 0.5 will center the servo, and can be further adjusted to properly level the control surface. Again, these two variables must be created and tuned by the user in controlMixer(). If the elevator moves in the incorrect direction, a negative sign can be added in front of the pitchAmount scaling factor to reverse the direction.

For the next example, we will consider basic, unstabilized elevon mixing for two elevon servos (arbitrarily assigned to servos 6 and 7). Following the same process in the above elevator servo example, we can assign both roll and pitch to both of the servos. Generally for elevon mixing, both servos receive the same sign of roll, and opposite sign of elevator (because of how servos are generally symmetrically mounted):

float pitchAmount = 0.7;
float rollAmount = 0.5;
float leftElevonOffset = 0.54;
float rightElevonOffset = 0.6;
s6_command_scaled = pitchAmount*pitch_passthru + rollAmount*roll_passthru + leftElevonOffset;
s6_command_scaled = -pitchAmount*pitch_passthru + rollAmount*roll_passthru + rightElevonOffset;

For this final example, we will consider unstabilized differential throttle of two wing mounted motors (arbitrarily assigned to motors 1 and 2) for yaw control. For the motors, the thro_des variable should be assigned to the ..._command_scaled motor variables to give throttle control. Then, yaw control input is achieved by conducting the same process in the above two examples, this time with the yaw_passthru variable:

float yawAmount = 0.15;
float yawOffset = 0.0;
m1_command_scaled = thro_des + yawAmount*yaw_passthru + yawOffset;
m2_command_scaled = thro_des - yawAmount*yaw_passthru - yawOffset;

Note that one yaw_passthru variable is positive and the other is negative. This gives opposite response for each of the motors, providing differential throttle. Also note that the chosen yawAmount (these variables must be created in controlMixer()!) is only 15% of the total throttle range. Too much differential throttle can be problematic and it is recommended to start lower and slowly work upwards. Finally, the yawOffset variable is created and applied to the motor variables, but it is 0.0. No offset is required unlike servos which require 0.5 offset to at least center the servo, though it can be useful for very fine trim adjustments. A significant yawOffset in this case would cause a motor to spin up even at minimum throttle which is unsafe.

9.9 Control Mixing: Advanced Variable Dynamics

Please thoroughly read and understand the basic fixed dynamics and unstabilized command control mixing tutorials first before reading this section.

This section covers only the basics the many options for advanced VTOL configurations where the vehicle dynamics change between different flight modes. This tutorial will serve as the basis for hybrid type platforms, which can be fully customized to your specific need. In the code, channel 6 is unused, which can be leveraged within controlMixer() to assign 2 different mixing configurations (or 3 if you have a 3 position switch, or almost infinite configurations if you have a slider assigned to channel 6 in your radio). For the following examples, we will assume channel 6 is on a two-position switch that toggles between a low PWM value of 1000 and a high PWM value of 2000 on the channel_6_PWM variable. Using a simple if statement, we can toggle between two different dynamic configurations–a quadcopter for hover, and a conventional airplane (one motor, 2 ailerons,

and an elevator all with no stabilization) for forward flight for this example:

```
float pitchAmount = 0.7;
float pitchOffset = 0.5;
float rollAmount = 0.65;
float rollOffsetLeft = 0.5;
float rollOffsetRight = 0.58;

if (channel_6_pwm > 1500) { //hover mode
m1_command_scaled = thro_des - pitch_PID + roll_PID + yaw_PID; //front left
m2_command_scaled = thro_des - pitch_PID - roll_PID - yaw_PID; //front right
m3_command_scaled = thro_des + pitch_PID - roll_PID + yaw_PID; //back right
m4_command_scaled = thro_des + pitch_PID + roll_PID - yaw_PID; //back left
m5_command_scaled = 0; //turn off forward flight motor in hover
s1_command_scaled = pitchOffset; //elevator servo, centered and not moving in hover
s2_command_scaled = rollOffsetLeft; //left aileron, centered and not moving in hover
s3_command_scaled = rollOffsetRight; //right aileron, centered and not moving in hover
}
else if (channel_6_pwm < 1500) { //forward flight mode
m1_command_scaled = 0; //turn off in forward flight
m2_command_scaled = 0; //turn off in forward flight
m3_command_scaled = 0; //turn off in forward flight
m4_command_scaled = 0; //turn off in forward flight
m5_command_scaled = thro_des; //direct throttle control
s1_command_scaled = pitchAmount*pitch_passthru + pitchOffset; //elevator
s2_command_scaled = rollAmount*roll_passthru + rollOffsetLeft; //left aileron
s3_command_scaled = rollAmount*roll_passthru + rollOffsetRight; //right aileron
}
```

The above code represents a very simple mixing configuration for an aircraft similar to Figure 33. In this example, only a two position switch is used. If your transmitter switch has 3 positions, then a third if statement with proper bounds on the channel_6_pwm variable (these can be checked with the printRadioData() function which will display the value received from the radio receiver) can be used to describe an intermediate 'transition' flight mode. This transition flight mode may have stabilization on the hover motors, and stabilization introduced on the control surfaces, for example.

Figure 33: Simple VTOL configuration that acts like a quadrotor in hover and a conventional fixed wing aircraft in forward flight.

There are many different variables throughout the code that can be modified within these 'flight mode' if statements. For example, all controller gains can be changed within these statements so that there is a set of gains tuned for hover, and a set tuned for forward flight. The maxAXIS variables declared in the beginning of the code can be changed here as well to give different control authority between different flight modes. Additionally, scaled servo or motor command variables ..._command_scaled can be manually adjusted between flight modes, for example to actuate a tilt mechanism. Controller types can also be changed between flight modes, but this must be done in it's own logical statement in the main loop where the controller function is called. The basic fading tutorial details the process of fading some of these tuneable variables rather than having them abruptly change between flight modes. At this point, the customization in control mixing is at the discretion of the user who is free to employ whatever advanced techniques they require for their application. Proficiency in Arduino will be a powerful asset for creating incredible transitioning VTOL vehicles with excellent flying qualities and smooth transitions.

9.10 Basic Fading

A simple function, floatFaderLinear(), is provided to aid with some fading between flight modes as discussed in the advanced variable dynamics control mixing tutorial. Linearly interpolating some variables with this function between flight modes will allow for smoother transition and overall better performance. Variables of interest to fade between flight modes in the controlMixer() include controller gains, the maxAXIS variables, servo/motor offset/setpoint variables, or servo/motor scaling variables. The floatFaderLinear() function linearly interpolates between two specified high and low values of a float-type variable. Recall that the main flight control loop is continuously running at 2000 Hz. As a result, traditional delay()-based methods of fading variables or motor/servo commands cannot be used as they would stall the main flight control loop. floatFaderLinear(), when called within the controlMixer(), increases or decreases the variable by a certain small amount for each loop iteration until the minimum or maximum specified amount is reached. The following example shows the use of this function to fade a controller gain variable between 0.1 and 0.3 at different speeds depending on the direction. Other lines of code within the if statements (..._command_scaled assignments) are omitted for clarity:

```
if (channel_6_pwm > 1500) { //hover mode
Kp_pitch_rate = floatFaderLinear(Kp_pitch_rate, 0.1, 0.3, 5.5, 1, 2000);
}
else if (channel_6_pwm < 1500) { //forward flight mode
Kp_pitch_rate = floatFaderLinear(Kp_pitch_rate, 0.1, 0.3, 2.5, 0, 2000);
}
```

The first input to the function is the variable which you wish to modify. The function returns the modified version of that variable, so that is assigned to the variable itself. The second input is the minimum desired value of the variable, followed by the maximum desired value. The fourth input to the function is the fade time, in seconds, which can be different depending on if you are increasing to the maximum desired value (fifth input a 1) or decreasing to the minimum desired value (fifth input a 0). The fifth input specifies whether you'd like the fader to stop at the maximum value (1) or minimum value (0) once reached. The final input to the function is the loop rate of the void loop(), which is 2000 Hz by default and should not be modified. Please note that any variables being used with this function must be declared outside of the controlMixer() function at the beginning of the code so that they are global, not local variables. These may include scaling variables or offset variables used for motor or servo assignment as discussed in the advanced variable dynamics control mixing tutorial.

9.11 Controller Selection and Tuning

There are three types of PID-based controllers provided with dRehmFlight: rate, basic angle, and advanced angle. A rate controller commands a desired aircraft rotation rate for each axis while an angle controller commands a desired aircraft angle for the roll and pitch axis (yaw is always rate controlled). More information on the implementation of the three controller types is provided in the function description section of this document.

The controller type must be selected within the void loop() in the PID controller section. Only one controller type may be uncommented at a time. Controller gains must be manually adjusted for each controller type depending on which one is being used. Please note that the following tutorials are intended for tuning the controllers for a hovering aircraft. Tuning for a fixed wing type aircraft follows a much different procedure, though it is generally easier to do with a passively stable platform.

9.11.1 Rate Controller

The first step in tuning the rate controller after uncommenting the controlRATE() function in the void loop() is to set the desired maximum rotation rates. This is done in the user specified variable declaration section with the maxRoll, maxPitch, and maxYaw variables, which correspond to maximum rates in degree/second when using the rate controller. Good starting values for roll and pitch are about 150 degrees/second, and 120 degrees/second for yaw, though these can be tuned later to taste. These values should not exceed 250 degrees/second.

Next, the P, I, and D variables for each axis ..._roll_rate, ..._pitch_rate, and ..._yaw should be set to starting values. The default code provides well-tuned values for a 1000 g quadrotor vehicle with 7 inch propellers. Take note of these to understand appropriate increments to change the gains by later, and set all I and D terms to 0. Lower the P terms from the default by about 50% and power up the vehicle to check that the motors are stabilized correctly by moving the vehicle

and the transmitter sticks. Attempt a brief flight (if possible, anchored down) and note the flight characteristics. If the vehicle is very slow to respond and there are no significant oscillations, begin to increase the P gains for each axis by about 15-20%. Do this until oscillations begin, in which case it is time to slowly dial in some D gain. Please note that the value of the default D gain is very low. Increasing these parameters too high will cause the motors to overheat. In very small increments (15% of the default values), increase the D gains for each axis appropriately. You should notice that the oscillations from increasing the P gains begin to be dampened out. At this point, you may begin increasing the P gains very slightly in conjunction with the D gains to get a snappier response. Be very careful not to increase the D gains too high. If the motors are becoming too hot or producing weird noises while flying, you have gone too far. Dial the controller gains back to a safe level where the response is still acceptable. Please note that this process requires individual treatment of each axis which can be done at the same time, but should be thought of as independent things being tuned.

Once the P and D gains are adequately tuned, you may notice that the vehicle has some steady state error i.e. it will want to very slowly roll or yaw even when no rates are being commanded from the transmitter. It is now time to dial in the I terms in increments of about 15% of the default gain values. Do this for each axis until letting go of the sticks 'parks' the vehicle at the attitude you let go of the sticks at and there is no more drift. No more I term is needed, as too much can cause low frequency oscillations. The controller is now tuned for normal flight, but can be fine-tuned to your preference.

9.11.2 Basic Angle Controller

Please note that the yaw axis is always stabilized using the method described in the rate controller tuning tutorial, regardless of selected controller type.

The first step in tuning the basic angle controller after uncommenting the controlANGLE() function in the void loop() is to set the desired maximum angles for roll and pitch. This is done in the user specified variable declaration section with the maxRoll and maxPitch variables, which correspond to maximum angles in degrees when using the angle controller. Good starting values for roll and pitch are about 30 degrees, though these can be tuned later to taste. These values should not exceed 60 degrees.

Next, the P, I, and D variables for each axis ..._roll_angle and ..._pitch_angle should be set to starting values. The default code provides well-tuned values for a 1000 g quadrotor vehicle with 7 inch propellers. Take note of these to understand appropriate increments to change the gains by later, and set all I and D terms to 0. Lower the P terms from the default by about 50% and power up the vehicle to check that the motors are stabilized correctly by moving the vehicle and the transmitter sticks. Attempt a brief flight (if possible, anchored down) and note the flight characteristics. Most likely, there will be some oscillations indicating that it is time to dial in some D gain in increments of about 20% of the default value. Do this until the oscillations are dampened out. The P gain may still be too low for adequate control authority, so begin to increase the P gain in 15% increments, increasing the D gain in similar increments as needed. Test the response by commanding a fast roll or pitch change (for example from level to 20 degrees and back). You should observe that the vehicle quickly moves to the desired attitude with no oscillations. Oscillations indicate that more D gain is needed, whereas a very slow response indicates that the system is over-damped and the D gain should be lowered. This process can be refined as the P and D gains are slowly increased together. However past a certain point, the gains will be too high and you may notice that the aircraft has a 'noisy' attitude i.e. it seems to jitter in the air despite a very good response to control inputs. In this case, the gains have been tuned too high; dial the controller

gains back to a safe level where the response is still acceptable and the jitters have stopped. Please note that this process requires individual treatment of each axis which can be done at the same time, but should be thought of as independent things being tuned.

Once the P and D gains are adequately tuned, you may notice that the vehicle has some steady state error i.e. it will want to drift even when no angles are being commanded from the transmitter. It is now time to dial in the I terms in increments of about 15% of the default gain values. Do this for each axis until letting go of the sticks brings the vehicle to level and it does not drift around very much. No more I term is needed, as too much can cause low frequency oscillations. The controller is now tuned for normal flight, but can be fine-tuned to your preference.

9.11.3 Advanced Angle Controller

Please note that the yaw axis is always stabilized using the method described in the rate controller tuning tutorial, regardless of selected controller type.

The first step in tuning the advanced angle controller is to tune the rate controller and set good gains for that. The advanced angle controller's inner loop uses the exact same rate controller and gains as the standard rate controller, so this must be tuned well first in order for the outer loop angle controller to work well. Uncomment the controlANGLE2() function in the void loop() and set the desired maximum angles for roll and pitch. This is done in the user specified variable declaration section with the maxRoll and maxPitch variables, which correspond to maximum angles in degrees when using the angle controller. Good starting values for roll and pitch are about 30 degrees, though these can be tuned later to taste. These values should not exceed 60 degrees.

Next, the P, I, and D variables for each axis ..._roll_angle and ..._pitch_angle should be set to starting values. The default code provides well-tuned values for a 1000 g quadrotor vehicle with 7 inch propellers. Take note of these to understand appropriate increments to change the gains by later, and set all I and D terms to 0. Please note that this controller type requires that the D angle gains Kd_roll_angle and Kd_pitch_angle are set to 0 at all times. Damping is achieved with a different variable B_loop contained within the controlANGLE2() function which is well-tuned and should not require significant, if any, modification.

Lower the P terms from the default by about 50% and power up the vehicle to check that the motors are stabilized correctly by moving the vehicle and the transmitter sticks. Attempt a brief flight (if possible, anchored down) and note the flight characteristics. If there are any oscillations, the B_loop damping term in controlANGLE2() may need to be decreased in 0.05 increments for more damping. The P gain may still be too low for adequate control authority, so begin to increase the P gain in 15% increments. Test the response by commanding a fast roll or pitch change (for example from level to 20 degrees and back). You should observe that the vehicle quickly moves to the desired attitude with no oscillations. This process can be refined as the P gains are slowly increased. However past a certain point, the gains will be too high and you may notice that the aircraft has a 'noisy' attitude i.e. it seems to jitter in the air despite a very good response to control inputs. In this case, the gains have been tuned too high; dial the controller gains back to a safe level where the response is still acceptable and the jitters have stopped. Please note that this process requires individual treatment of each axis which can be done at the same time, but should be thought of as independent things being tuned.

Once the P gains are adequately tuned, you may notice that the vehicle has some steady state error i.e. it will want to drift even when no angles are being commanded from the transmitter. It is now time to dial in the I terms in increments of about 15% of the default gain values. Do this for each axis until letting go of the sticks brings the vehicle to level and it does not drift around very much. No more I term is needed, as too much can cause low frequency oscillations. The controller

is now tuned for normal flight, but can be fine-tuned to your preference.

9.12 MPU9250 Integration

The default code and hardware is intended to be used in conjunction with the MPU6050 6 degree of freedom IMU. Excellent attitude estimates are achieved with this cheap IMU which should provide adequate performance for any use case. With that said, code support for the MPU9250 9 degree of freedom IMU is included as well. The MPU9250 uses SPI connection for faster data transfer compared to I2C. The downside of this is that it requires additional pins. The small pads on the back side of the Teensy 4.0 are used instead of the 2 pins on the top of the board for the MPU6050 default hardware setup; these are selected to free up the more accessible pins on the top of the board with the downside of them being more difficult to solder. Table 27 gives the pins on the IMU and Teensy 4.0 that must be connected for proper operation.

IMU Pin	**Teensy 4.0 Pin**
VCC	Any 5v Power
GND	Any Ground
SCL	37 (SCK2)
SDA	35 (MOSI2)
AD0	34 (MISO2)
NCS	36 (CS2)

Table 27: Pin connection from MPU9250 IMU to the Teensy 4.0.

The MPU9250 has slightly different characteristics compared to the MPU6050, which this code was originally written and tuned for. Before flying, adjust the B_accel parameter to 0.2 and the B_gyro parameter to 0.17. These values may need further tuning. Additionally, please allow 5-10 extra seconds after the flight controller enters the main loop (indicated by a quick blink every 1.5 seconds) before attempting to fly so that the 9DOF attitude estimation has adequate time to converge. If possible, power up the flight controller with the nose of the aircraft facing North to allow the attitude estimate to converge more quickly.

10 Working Vehicles

dRehmFlight has been successfully implemented on the following platforms:

10.1 Quadrotor

The code as provided is configured for a simple quadrotor configuration. This is recommended as a starting point for becoming familiar with the flow of code and options for customization. The following quadrotor is capable of piloted flight, but was also supplemented with an onboard flight computer which sent simulated radio commands (modified in the getCommands() function to monitor the state of an auxiliary switch to either get radio commands from the receiver or through a simple serial connection with the flight computer) enabling complete autonomous flight.

Figure 34: Simple quadrotor platform.

Flight video:
https://www.youtube.com/watch?v=GZLUbTSWPI8

10.2 Quadrotor Biplane VTOL

The simple quadrotor code can be easily modified for a quadrotor biplane type VTOL platform. The state of an auxiliary switch is monitored in the controlMixer() function, where the roll and yaw axis are switched between hover and airplane mode along with appropriate command reversing. The controller gains can also be modified based on the state of this switch for unique tuning between different flight modes.

Figure 35: Quadrotor Biplane VTOL platform.

Flight video:
https://www.youtube.com/watch?v=rk4tUKM6bd0

10.3 Dual Cyclocopter

The dual cyclocopter is a very unique platform that requires special control mixing in the controlMixer() function. A combination of motors and servos for roll, pitch, and yaw control require specific assignment of stabilized axis variables from the chosen controller type.

Figure 36: Dual cyclocopter platform.

Flight video:
https://www.youtube.com/watch?v=uP87-yU1l6I

10.4 F-35 Tricopter VTOL

A tricopter-based hovering F-35 platform is an excellent example of the power of the Teensy 4.0 to do extremely custom control mixing and fading tailored to the user's need. The state of an auxiliary switch is monitored in the controlMixer() function to change assignments to the motors and servos for hover, transition, and forward flight. Controller gains are also adjusted based off of the state of this switch. Additionally, unique fading in code is used to entirely turn off the front lifting motor as the vehicle transitions into forward flight–this can be done with basic knowledge of logical statements in code.

Figure 37: F-35 tricopter VTOL platform.

Flight video:
https://www.youtube.com/watch?v=Ds-ODWydxeY

11 Frequently Asked Questions

- **Question:** "I found a better/faster way to do the same code, why can't you change your code for better performance?"

  Answer: The purpose of this project is not to provide the highest performance code, but to provide the most understandable code that anyone with minimal Arduino experience can easily modify for their use. The Teensy 4.0 (and 4.1) board is more than fast enough to run sloppy code at high speed for a flight controller application. You are more than welcome to modify and improve the code for your application, this is just a starting point!

- **Question:** "I got your code and flight controller setup and running on my vehicle. Why does it not fly itself and still crash?"

  Answer: dRehmFlight is a flight stabilization package, not an autopilot. The pilot remains in full control of the vehicle attitude at all times and must have prior experience flying model aircraft for a successful flight.

- **Question:** "Can I use a different board for my flight controller assembly other than the Teensy 4.0? For example, Teensy 3.6, Arduino Uno, or Arduino Nano?"

  Answer: No. This code is built to run on the Teensy 4.0 with Cortex M7 microprocessor. Other microcontrollers such as Arduino or earlier versions of Teensy are not fast enough and/or do not have the necessary i/o capability. The Teensy 4.1 is the only other board supported, though it has not yet been tested in flight.

- **Question:** "Your code caused my expensive aircraft to crash, what will you do about that?"

  Answer: I assume no responsibility for damages done to your aircraft or property as a result of the use of this code. If you are using this code for a larger or more expensive project, I assume that you are capable enough to ensure proper safety protocol are being followed and that all aspects of the code–default or modified–are properly inspected and checked for correct operation prior to flying. Use and modify at your own risk. With that said, any feedback or data on failure of the code, if provided, will help to improve future versions.

- **Question:** "Will you be updating the code in the future?"

  Answer: Yes, the current version is very early in development. Expect variable and function names to be adjusted to better reflect their use, the implementation of new functions to expand functionality, and support for more radio and ESC protocols in the future.

12 License Information

13 Disclaimer

This code is a shared, open source flight controller for small micro aerial vehicles and is intended to be modified to suit your needs. It is NOT intended to be used on manned vehicles. I do not claim any responsibility for any damage or injury that may be inflicted as a result of the use of this

code. Use and modify at your own risk. More specifically put:

THIS SOFTWARE IS PROVIDED BY THE CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

14 Credits

This project is the result of the need for a simple, easy to understand flight controller solution. Special thanks to Joop Brokking for his work on the YMFC-32 project. Many elements of this code were inspired by his work:
http://www.brokking.net/ymfc-al_main.html

Skeleton code for reading the MPU6050 IMU was borrowed and adapted from How To Mechatronics:
https://bit.ly/3hzCQ03

MPU9250 implementation based on MPU9250 library by brian.taylor@bolderflight.com:
http://www.bolderflight.com

Finally, the Madgwick filter function was adapted from:
https://github.com/arduino-libraries/MadgwickAHRS

Special thanks to 'jihlein' on RcGroups for the IMU implementation overhaul and SBUS integration.

15 Citing This Work

If you are using this for an academic or scholarly project, please credit me in any presentations or publications:

@misc{dRehmFlight,
author = {Rehm, Nicholas},
title = {dRehmFlight VTOL Flight Controller},
month = {Jan},
year = {2020},
url = {https://github.com/nickrehm/dRehmFlight}
}