

Radio Flyers



Final Report
Jay Burns, Julian Corona, Steven Cress,
John Walsh, Karen Yokum

May 9, 2012

Contents

1	Introduction	2
2	Detailed System Requirements	2
3	Detailed Project Description	4
3.1	System theory of operation	4
3.2	System Block diagram	4
3.3	Detailed Design/Operation of the Subsystems	5
3.3.1	PWM	5
3.3.2	Sensors	8
3.3.3	User Interface	10
3.3.4	Motors	10
3.4	Interfaces	10
4	System Integration Testing	10
4.1	Description of how the integrated set of subsystems was tested	10
5	Users Manual	11
5.1	How to install	11
5.2	How to setup	11
5.3	How to tell whether the system is working	11
5.4	How to trouble shoot the system	11
6	To-Market Design Changes	12
7	Conclusions	12
	Appendices	14
A	Setup of the Aircraft	14
B	Debugging the system	15
C	Eagle Documentation	16
D	Parts List	17
E	Code Listing	18

1 Introduction

Existing Unmanned Aerial Vehicle (UAV) technology is expensive and requires constant supervision for control. For example, the Raven system used by the U.S. Army costs around \$200,000 for the airframe alone and requires a remote operator. Our approach offers a simpler, more cost effective way to obtain the same information. This more practical approach involves uploading a pre-defined flight plan to an on-board autopilot and having the UAV fly autonomously. This system will allow the operator to focus only on gathering the required intelligence rather than on flying the aircraft to and from the target area. The system will also allow an operator to take control if desired. An existing remote control aircraft was purchased for the aerial surveillance platform. An inexpensive, yet readily available, GPS receiver, accelerometer, magnetometer, gyroscope, barometer, and sonar sensors are used to guide the microcontroller-driven autopilot on the predetermined track. The main intelligence of the system comes in the form of a proportional integral derivative (PID) controller. This controller takes the data collected from the accelerometer and uses it to adjust the power to the individual on board servo motors as needed to keep the aircraft level when suspended from a string. The system also has a kill switch on the transmitter as a safety precaution in the event that the system becomes unstable. The system is powered by two on-board Lithium-Polymer batteries. The first battery powers the microcontroller and sensors. The second battery, producing higher current, provides sufficient power to the servomotors used to propel the vehicle. The platform requires a transmitter and receiver pair. The first set is for allowing manual control of the airplane. This Tx/Rx contains 6 channels for aircraft control and the autopilot on/off toggle. The user interface is the programming of the PID controller for adjustment of k values in the equations for tuning of the stability algorithm. As the project stands right now, we have a functioning board with sensors attached and interfaced. The accelerometer is the only sensor giving us good data at the moment. The gyroscope is sending some output data, but we have yet to desypher what it is and if it is any good. The GPS has some code developed for both serial and NMEA interfacing, either of which can be developed further for usability. We are able to send signals to the barometer, but unable to obtain temperature readings from the device. For this reason, we did not move on to attempt altitude readings. Analog code has been written for the sonar, but a hang-up is preventing it from compiling. On the controls side, a simple PID controller has been designed and can now begin to stabilize the quadrotor while hanging from a string. The problem here lies in the reaction rate of the PID. When the system is tilted, it takes the controller some time to react to the change. This delay would have to reduce dramatically for sustained flight to be possible. At this point, the quadrotor is in a good position to be handed off to a future group. The next group would first want to finish the interface of between the sensors and microcontroller. Collecting this data strengthens necessary information to create a more robust control algorithm. Then this project just becomes a controls problem.

2 Detailed System Requirements

There are a number of primary system requirements that must be fulfilled to address the problem at hand. In any aerial system, safety must be the first priority. Therefore,

one requirement involves a robust method of allowing the user to control the aircraft if the autopilot system malfunctions. This manual override is implemented through the control algorithm. Take-off, landing, or any emergency situation will be controlled manually as opposed to utilizing the autopilot function. A second requirement regards the stability of the autopilot. This is also implemented in the control algorithm. Without stabilization, the quadcopter could crash, or the platform could vacillate changing the flight dynamics. In order to conduct surveillance, the quadrotor will need to provide a stable platform for a camera that could be attached as a future addition to the project. To that end, we ensure that the autopilot control software provides stabilization and error corrections to fly the quadrotor. Also, the accuracy of the sensors is sufficient to derive accurate position and velocity inputs to the autopilot. This requires that the PIC24FJ256GB110 microcontroller perform integral and differential calculus. A variety of sensors are used to provide the position and orientation of the quadrotor. These sensors include an accelerometer, barometer, gyroscope, magnetometer, and ultrasonic range finder. The I²C and RS232 protocols are required to interface these sensors with the microcontroller. Another design requirement concerns the wireless interface for the UAV. The handheld radio controller uses 5 channels (4 for primary flight control and a 5th to engage the autopilot). The video transmitter will be on a non interfering wireless band that is powerful enough to send usable data over at least the line-of-sight range of the quadrotor. Furthermore, the system is powered by two sets of on-board batteries. The first set powers the microcontroller, camera, and sensors. The second set, providing a higher current, provides sufficient power to the motors that are used to propel and maneuver the quadrotor. The user interface consists of inputting the desired GPS coordinates and taking radio control for takeoff, landing and manual override. The user will input the GPS coordinates via a terminal command prompt.

The autopilot algorithm will start with a user input of GPS coordinates to the microcontroller. This is the vehicle's flight plan. The GPS and barometer provide x, y and z coordinates that determine where the quadrotor is in space while the gyroscope determines the orientation. The autopilot algorithm will track these coordinates, compare them to the GPS and barometer data, and update the flight plan accordingly. The motors also responds to accelerometer/gyroscope readings for orientation in the case of sudden gusts of wind or other acts of God. The camera on board could send real-time visual data back to a user interface. The GPS will interface with the microcontroller using a digital to serial protocol by utilizing the transmit (Tx) and receive (Rx) pins. The barometer connects to the microcontroller using I²C protocol which requires data and clock pins. The accelerometer provides an analog out signal which will connect to the microcontroller via three pins (x, y, z). The camera has a TTL serial to digital output which connects to the microcontroller via Tx and Rx pins. The video transmitter connects to the microcontroller with a serial to digital protocol using Tx and Rx pins. The SD card interfaces using SPI to the microcontroller. A computer uses a USB interface with the microcontroller to program the flight plan. A radio controller transmits user inputs to the receiver on-board the quadrotor. The receiver will then pass these inputs to the microcontroller. An autopilot toggle input will determine whether or not to pass the user inputs to the motors or use the output of the autopilot algorithm. This requires 5 digital I/O pins between the receiver and microcontroller. The barometer is an I²C that will require an I²C serial bus clock input (SCL), as well as an I²C serial bus data line (SDA). The barometer has an operating voltage of 3.3 V and consumes

3 A. The accuracy of the pressure readings is 0.01 hPa. The LSM303DLHC is a combined accelerometer and magnetometer that is also an I²C device. It has an operating voltage of 3.3 V and has full scale readings of 2 to 1.3 gauss. The ultrasonic range finder is a serial device that can detect a distance greater than 20 ft. It also has an operating voltage of 3.3 V and consumes 2 mA. The gyroscope includes a sensing element and an I²C interface. It is a three-axis angular measurement sensor that needs an input of 3.3 V. In total, the microprocessor will require 21 digital I/O pins, 1 analog input pins, a USB interface, an RS232 interface, and an I²C interface.

3 Detailed Project Description

3.1 System theory of operation

Action of the quadcopter will begin with manual take off from the transmitter exhibiting its stabilization ability. After self calibration, the the flight plan is implemented. The GPS is used to automatically track to the user inputted coordinates. All the while, the accelerometer, barometer, gyroscope and magnetometer will collect data concerning the inertia, altitude, angular rate and orientation respectively. The accelerometer utilizes a 3-axis input based on the weight experienced due to gravity. This provides a three dimensional vector associated with angular orientation of the vehicle. From these inputs, the microcontroller computes a running average of the samples to compare to the desired orientation. Similarly, the barometer computes a running average of air pressure readings to determine the altitude of the vehicle. This is also compared to the desired value of elevation. The gyroscope and magnetometer both deal with orientation of the vehicle. The gyroscope, using properties of angular momentum, measures the angular rate, while the magnetometer, using the strength and direction of magnetic fields, measures the azimuth and dip (inclination) of the quadcopter.

From the output data of the sensors, the microcontroller adjusts the power setting of each motor. This is governed by a change in the pulse width modulation duty cycle. If forward mobility is desired, power is increased to the back two propellers providing the necessary angle for quadrotor to advance forward. Then all four motors are then given an equal amount of thrust to keep the quadrotor moving forward with an accurate stability. Likewise, to cease forward progress, the power will be given to the front two propellers. Thus, it will mitigate the original forward angle. The ultrasonic range finder is used primarily during autopilot landings. This sensor, pointed directly down from the quadcopter provides additional altitude data via sonar while approaching the ground.

3.2 System Block diagram

A block diagram detailing all of the major components for the proposed UAV is shown in Figure 1. The devices are connected to the microcontroller as shown in Figure 2. As can be seen in Figure 2, the GPS is connected to the serial transmit and receive pins which are pins 49 and 50 respectively. The other devices, namely the accelerometer/magnetometer, barometer, ultrasonic range finder, and gyroscope, are connected to the data and clock lines

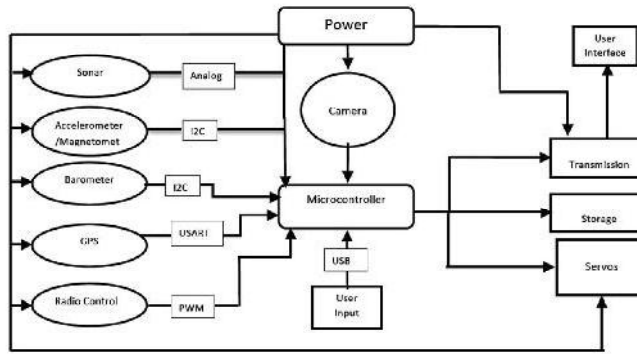


Figure 1: Overall system block diagram

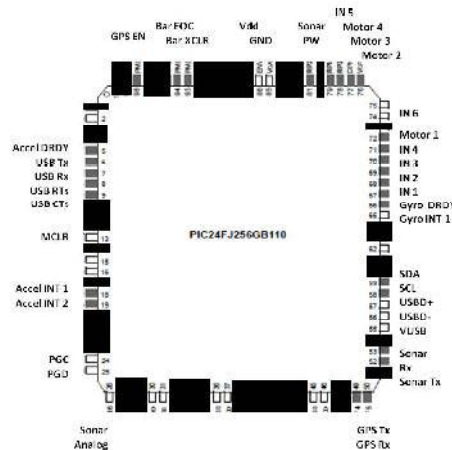


Figure 2: Pin Diagram Connections

of the I²C pins (pins 59 and 58 respectively) of the microcontroller.

3.3 Detailed Design/Operation of the Subsystems

3.3.1 PWM

Pulse width modulation (PWM) is the system by which all command input and output will interface with the microcontroller. The digital radio control system transmits PWM signals on 6 channels at roughly 54 Hz, with the pulse width varying between 6 and 10.5%. The receiver picks up these signals and, with appropriate power, outputs them on 6 pins. The motors can receive PWM signals on a wide range of frequencies and pulse widths, but for purposes of simplicity, we have elected to output 54 Hz signals between 6 and 10.5% to the motors.

In terms of requirements, the output segment needed to generate the aforementioned

signal range and frequency in such a fashion that the motors received a robust and consistent signal. Furthermore, the system needed to function independent of the main loop of the microcontroller, in the event that some computation requires more time than the period of the PWM signal. The PWM input needed to accurately capture the receiver channel signal pulse lengths and, as with the output PWM, function independently of the main loop.

The output segment of the PWM interface was the first designed. The PIC24FJ256GB110 has Output Compare (OC) modules that are designed specifically for this purpose. To function properly, these modules first need to be mapped to pins using the Peripheral Pin Select (PPS) feature of the microcontroller. Then, they must be configured by selecting a timer, a sync source, the specific function required (in our case, edge aligned PWM), and finally, setting the synchronous mode. Next, the timer must be configured. This requires writing the period value to the period register, setting the prescale value, enabling the timer's interrupt and clearing the flag, and turning the timer on. We selected Timer2 to provide the clock source for all OC modules. With the system clock running at 32 MHz, the timer increments at 16 Mhz ($F_{osc}/2$). Thus, to achieve a 54 Hz signal, the prescale value was set at 64, and the period register was set with a value of 4629. The timer then interrupts when the timer register and period register generate a match. This interrupt triggers the timer value to reset to 0, and the OC module to set its pin high and remain high until the value set in its 16 bit register (called OCxR) is matched with the timer value. By writing to the OCxR register, one can change the pulse width. This is illustrated in Figure 3. The code for implementing

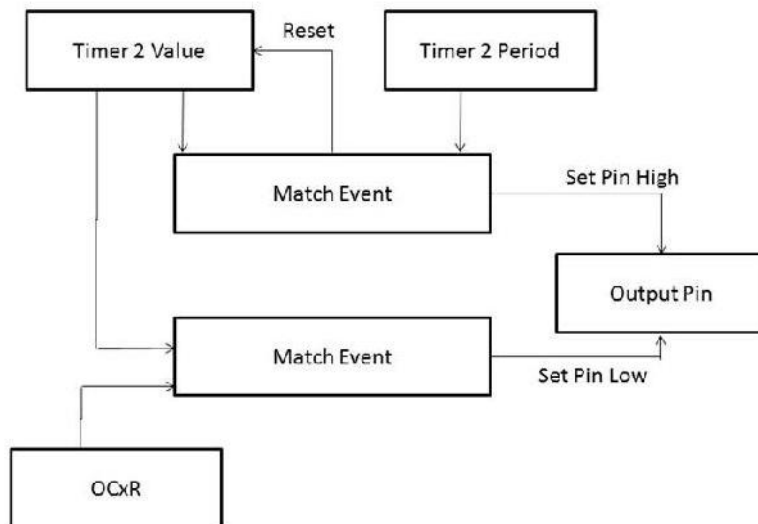


Figure 3: Flow chart of the input capture process

output PWM was written into a function called `initializePWMoutput()`. This function is listed in appendix 1.1. Next, the input PWM system was configured utilizing the Input Capture (IC) modules. This was accomplished by mapping the IC modules to the correct Peripheral Pin Select pins, and setting those pins as inputs by writing to the TRIS register. Next, the IC module was set to capture the timer on every signal edge, interrupt on every capture event, sync with the selected timer, and enabling that module's interrupt flag. On

every interrupt, the value of the selected timer is written to a buffer called ICxBUF. In the interrupt routine, the value of ICxBUF is removed and written to one of two variables called ICxCapture1 and ICxCapture2. If the pin is high when the interrupt is generated, the value is written to the Capture1 variable, and this is considered the start count. If the pin is low when the interrupt occurs, the value is written to Capture 2. By taking the difference of the numbers, the total number of counts that the pulse was high may be obtained. Since the signals from the receiver are cascaded by channel, the timer is set back to 0 when the channel 5 IC module interrupts and the pin is determined to be low, thus signifying that all inputs will remain low until the start of the next period. Timer4 was selected for input capture and configured in the same manner as Timer2. The input capture process is illustrated in Figure 4. The code for implementing the input chapter was written into a function called

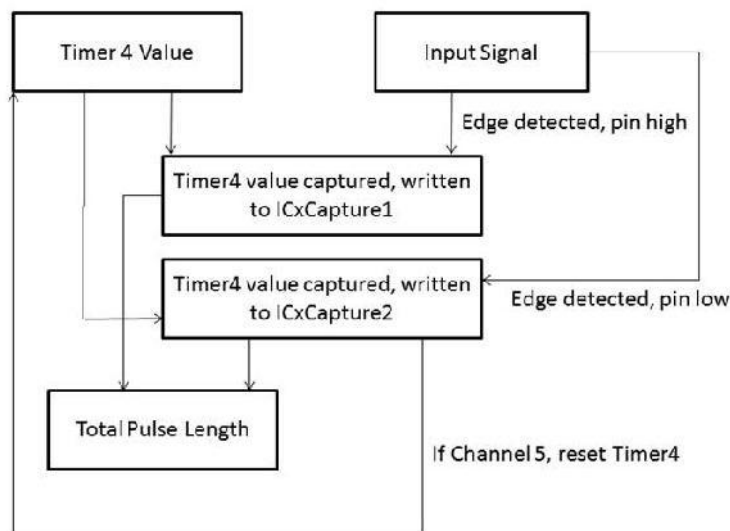


Figure 4: Flow chart of the input capture process

initializePWMcapture() in addition to the interrupt service routines. This code is listed below. Finally, several simple functions were written to make the input and output easier to interact with. These are found in appendix 1.3. A function called readInput() simple checks the integrity of the pulse count and scales it to a percentage from 0 to 100%. For channel 5, which is a switch, a function was written to determine the position of the switch called readCH5(). Lastly, a function called setPower() takes in the four desired throttle positions of each motor in terms of a percentage from 0 to 100%, scales them to a timer count, and writes them to the respective OCxR registers. The input and output PWM were tested to ensure functionality simply by reading in the values of each channel and outputting them to the USBee. The input from the receiver is displayed in Figure 5. The output from the microcontroller is displayed in Figure 6. In this image, it is outputting the four channels from the transmitter with the throttle idle. Therefore, note that channels 1, 2, and 4 are output an 8.2% duty cycle signal (since those those channels were at the neutral position), and channel 3 is outputting 6% duty cycle signal, since the throttle was low.

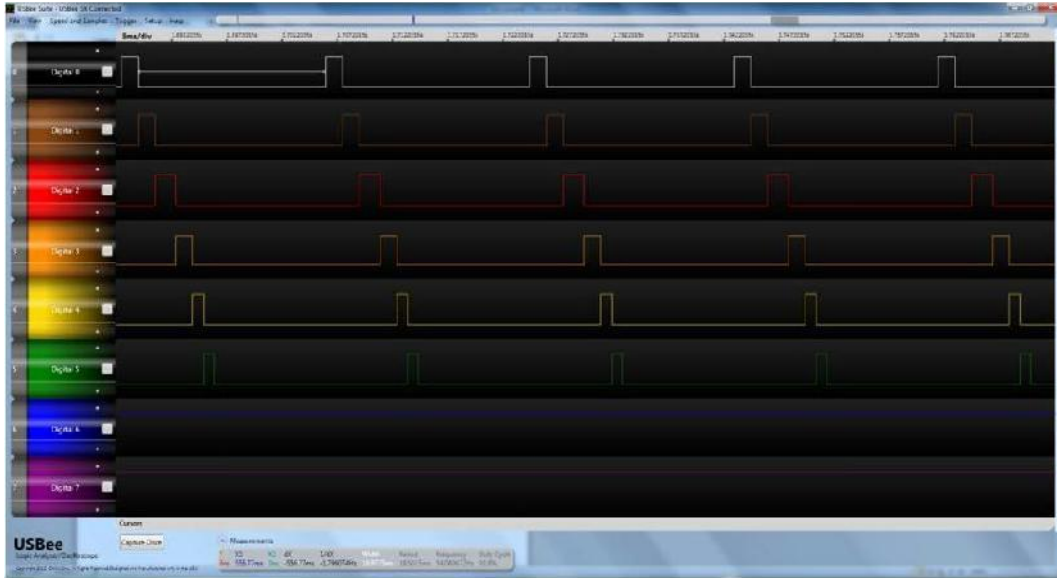


Figure 5: Input from the receiver (screen shot of USBee)

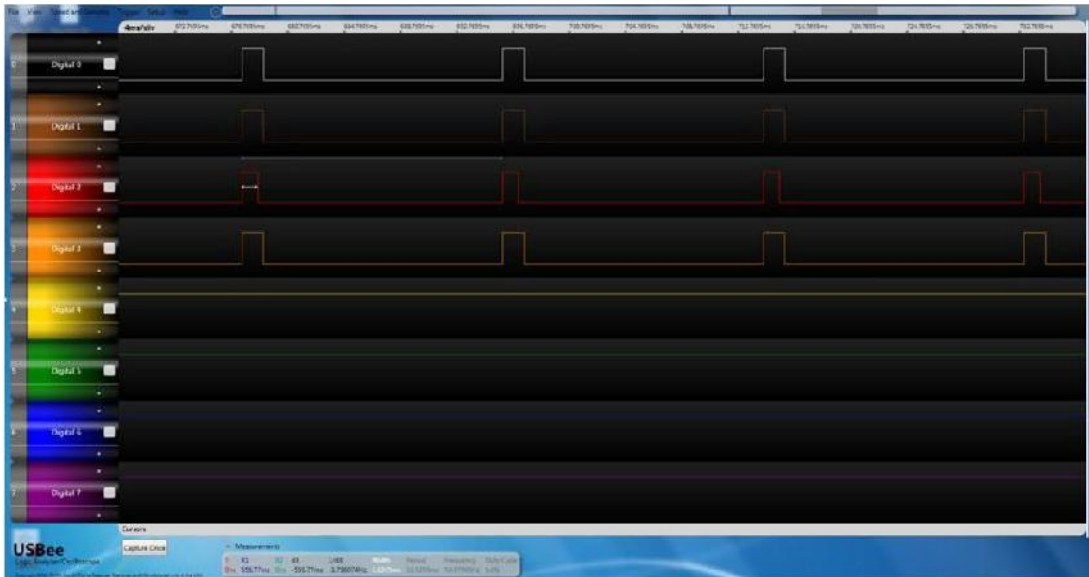


Figure 6: Output from the microcontroller (screen shot of USBee)

3.3.2 Sensors

There are a total of six sensors used to extract data about the QuadRotor and the environment. Those six sensors are GPS, Sonar, accelerometer, magnetometer, gyroscope, and barometer; and they are used to determine the six dimensional orientation vector of the QuadRotor. The two component of this orientation vector are position and angular orientation. The GPS, the barometer, and sonar are used to determine the position vector. The GPS is used to determine the QR's x and y coordinates relative to the zero-frame, the earth. The barometer is used to determine the QR's z coordinate in the zero-frame.

Finally, the sonar is used to detect when the ground is within 10 feet of the bottom of the QuadRotor. Each of these devices uses a different method to talk to the microcontroller. The GPS transmits its data using UART, the barometer uses I2C, and sonar sends an analog signal. The angular orientation of the QR is determined with a combination of the accelerometer, magnetometer, and gyroscope. The gyroscope outputs angular rates in the x, y, z vectors relative to the QuadRotor. The code used to get the Rotational vector is shown in Appendix 1.4. The primary functions are Gyro_init() which initializes the Gyroscope, Gyro_getRotVec() which gets the rotation vector, and Gyro_getOff() which is called in the beginning of the code to zero out the data when there is zero rotation. The gyroscope is particularly noisy; so to compensate for this every time the data is collected, we actually pull the last 16 sets of values for each component. This data is then averaged to produce less noisy information. Even with this averaging, the data still fluctuates pretty wildly. This needs to be accounted for in any control algorithm that uses this data. The accelerometer outputs an acceleration vector in x, y, z coordinates relative to the QR. Assuming there is no actual acceleration of the QR in the x or y coordinates of the zero frame (which we don't want and are driving those accelerations to zero), the acceleration of the QR always points in the z direction relative to the zero frame. This fact is very helpful and is used to help determine the orientation matrix of the QR. The acceleration data collected is very accurate and precise. There is no zeroing needed in the x or y directions. However, a zero factor of roughly 0.2g is used for the z direction. This zero factor is determined in the initializations section of the code which samples the acceleration in the z frame 256 times. This data is then averaged and subtracted from 1 to get the offset. With the offset applied to the data, the z component of the accurately shows 1g in the positive z direction at rest. The code for the accelerometer is shown in Appendix 1.5. The main functions are Acc_init() which initializes the registers in the Accelerometer needed for proper operation, Acc_getAccVec() which gets the acceleration vector, and Acc_getZoff() which gets the z offset. Finally, the magnetometer outputs a magnetic vector in the x, y, z coordinates of the QuadRotor. This vector should always be pointing in the y direction relative to the zero frame assuming that there is no other magnetic field present other than that of the earth. The code for the magnetometer is shown in Appendix 1.6 and the primary functions are Mag_init() which initializes the magnetometer, and Mag_getMagVec() which produces the magnetic vector. The magnetic vector can then be used in conjunction with the acceleration vector to produce the rotation matrix of the QRs coordinate frame. The magnetic vector and the acceleration vector are orthogonal with the magnetic vector in the positive y direction of the zero frame, and the acceleration vector in the positive z direction of the zero frame. The cross product between the magnetic vector and the acceleration vector yield a vector mutually orthogonal in the positive x direction of the zero frame. Using these three vectors, the rotation matrix can be determined using the equation below where V_0 is the vector relative to the zero frame and V_{QR} is the vector relative to the QuadRotor. Plugging in for the six vectors we know—Acceleration, magnetic, and their cross product in both the zero and QR frame—the rotation matrix can easily be solved. This matrix allows us to transform any vector in the zero frame to a vector in the QR frame at a given time.

$$\begin{bmatrix} r_{1,1} & r_{1,2} & r_{1,3} \\ r_{2,1} & r_{2,2} & r_{2,3} \\ r_{3,1} & r_{3,2} & r_{3,3} \end{bmatrix} V_0 = V_{QR}$$

3.3.3 User Interface

There are two ways the user interacts with the QuadRotor. The first is when inputting the desired gps coordinates and to walk through initializing the QuadRotor. During this stage, the user communicates with the microcontroller through a USB interface and a terminal. The USB interface is connected to the microcontroller through a UART connection. This type of connection allows for data to be transmitted to and from a controller without a clock to synchronize the data. The second interface is the RC transmitter the user uses to input commands to the QuadRotor during flight. The transmitter talks directly to a receiver both operating on the 72.97 mHz band range. The transmitter uses five channels to talk to the receiver. Four of the channels are for aircraft manipulation—roll, pitch, yaw, and throttle—and the fifth channel is an on/off switch for the autopilot. The receiver then outputs a PWM signal which is read by the microcontroller.

3.3.4 Motors

The motors are controlled by Electric Speed Controllers, or ESCs. The ESCs accept the control information from the microcontroller via the PWM mentioned previously, and interpret the signal so to vary the FETs accordingly. This signal is then routed to the motors through the power distribution board on the lower level of the quadcopter. Power is also routed to these ESCs through the power distribution board, which is being fed by the Li-Po battery.

3.4 Interfaces

A user interface is required to input the desired GPS coordinates to the microcontroller. A terminal window on a PC is used to communicate with the microcontroller through a USB connection.

4 System Integration Testing

4.1 Description of how the integrated set of subsystems was tested

To test the pulse width modulation code, the duty cycle range of the transmitter had to be determined. This was tested by connecting the USBEE logic analyzer to the radio receiver and varying the throttle on the radio transmitter. The throttle was swept from 0% to 100% and the signal was recorded. From this data, the range of the duty cycle transmitted could be determined. At zero throttle, the duty cycle was 6%. At full throttle, the duty cycle was 10.5%. (See Figure 5). This data was used to govern what the PWM algorithm must output for the desired thrust. The Universal Asynchronous Receiver/Transmitter (UART) is hardware in which data is loaded into a register and shipped out on a single line. The UART

provides a testing platform for the sensors by allowing us to display the data obtained via a terminal window. Using this method, we were able to test readings from the accelerometer to ensure we were collecting sensible data. Tests for the accelerometer consisted of tilting the board and observing the magnitude and sign of the output. This allowed us to ensure that the axis was properly set. In order to determine the output of the motors as a function of the input duty cycle, a test bench was created that measured the thrust and torque of the motors. Figure 7 shows the setup used to measure lift produced by the propellers while Figure 8 shows the setup used to measure the torque produced by the motors. In Figure 7, a WeighMax electronic postal scale measured the force each motor outputted by measuring the upward force created when the throttle was varied. The device, in Figure 8, allows the motor to spin freely about its center. A pull spring scale was attached to one end of the platform and, using the equation for torque, the reaction torque of the motor felt by the frame can be obtained for various throttle inputs. This procedure was repeated for each motor and a best fit line was fitted to the data. This can be seen in Figure 9. This gives an equation for thrust and torque as a function of the input duty cycle.

5 Users Manual

5.1 How to install

After installing the MPLAB X IDE, click New Project in the File menu. Under the Microchip Embedded folder, select Standalone Project and click Next. Choose the Device to be programmed. In our case, our Device was 24FJ256GB110. Select ICD 3 under the Select Tool menu and click on the compiler used. Finally, give the project a name. Once the project is created, header and source files can be created by right clicking the respective group under the project heading. Choose the name of the file and the extension (either .c or .h). When ready to build the project, click on the hammer icon in the top toolbar. If the build is successful, the user can compile the program using melabs Programmer.

5.2 How to setup

For setup of the aircraft itself, see appendix 1. When mounting the board, ensure that it is properly oriented as to coorespond to the controls algorithym you are using.

5.3 How to tell whether the system is working

See appendix 2.

5.4 How to trouble shoot the system

Trouble shooting the system requires self evaluation of the code. One of the first things to check would be declorations. Be sure to understand what variables are global and which are volitile. Other problamatic areas include pin assignments, bit registers, and interupt functions.

6 To-Market Design Changes

As our project did not conclude in a not a completed product, many steps must be taken before it becomes marketable. Through the course of the semester, we were able to complete the hardware side of the project. The board was completed, and the sensors integrated with the microcontroller. this made it so data regarding the quadrotor's orientation and velocity could be collected on board. Within the last week, a simple PID controller was developed demonstrating that this data was in fact usable in flight. From here, the steps needed to be taken are software based. From the data collected, a more robust control algorithm must be developed to make flight possible. The PID controller we designed was too slow in response to changes to maintain it's hover. Once this is completed, more precise control of the power delivered to the motors would make it so the quadrotor could take off and hover by itself. With this step accomplished, it would be possible to begin to integrate the GPS data available and track latitude and longitudinal coordinates, excicuting a flight plan.

7 Conclusions

Tough the project did not culminate in a fully autonomous UAV, great progress was made in this direction. The board was designed and manufactured. Sensors including 3-axis Accelerometer/magnetometer, 3-axis gyroscope, barometer, sonar and GPS were interfaced with a PIC 24 microcontroller collecting data corresponding to position, orientation and velocity of the vehicle. Now the project is in a good position to be handed off to the a future group as mainly a controls project.

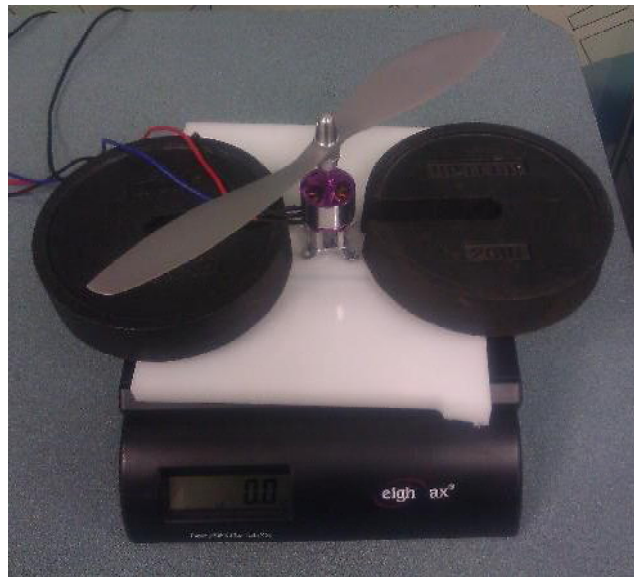


Figure 7: Setup used to measure lift produced by the propellers

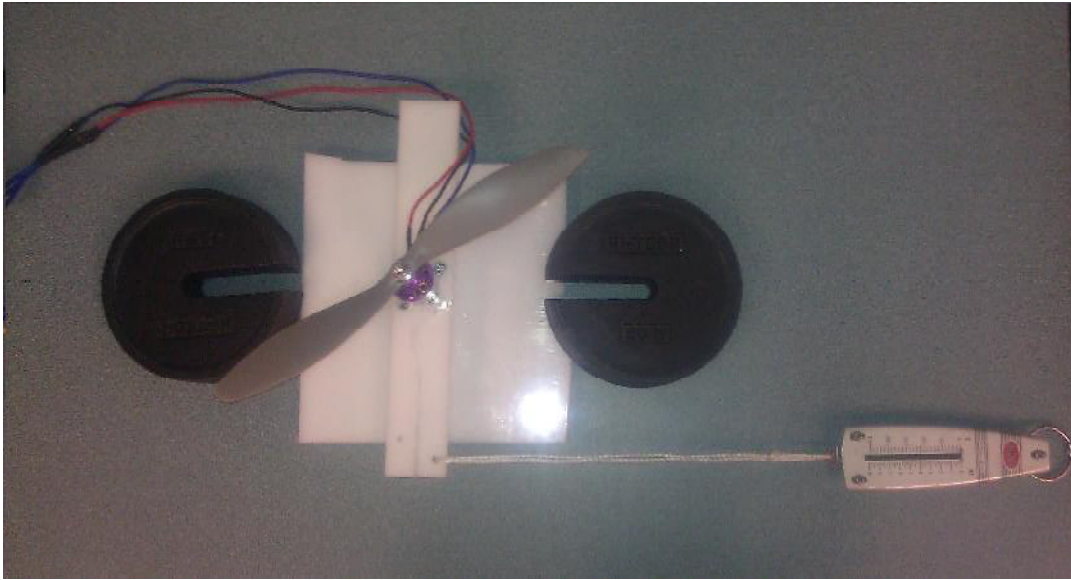


Figure 8: Setup used to measure torque produced by the propellers

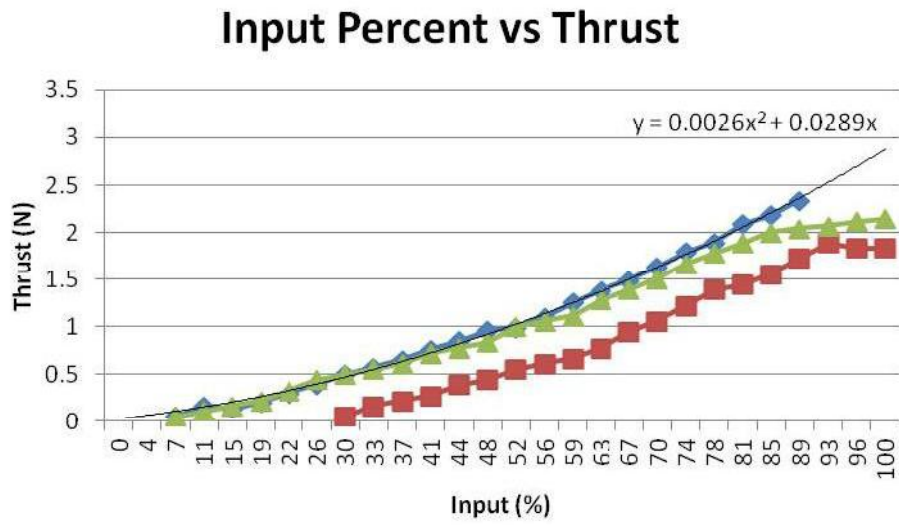


Figure 9: Data obtained from motors

(1)

Appendices

A Setup of the Aircraft

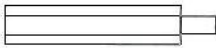
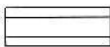










Arducopter 3DR-B

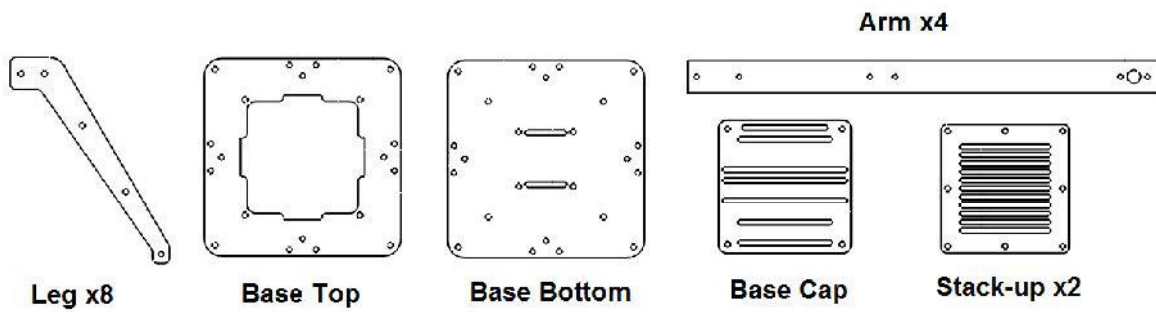


Thank you for purchasing an Arducopter 3DR kit. The Arducopter 3DR is a stable and supported quadrotor frame in the ongoing development of the Arducopter code on DIYDrones. It features a very durable Aluminum and G10 FR4 frame that can withstand hard impacts. The wide legged stand allows for more stable takeoffs and landings and provides an unobstructed view for a bottom mounted camera. The latest revision of this frame (revision B) features a removable base for easy access to the PDB and APM mounting slots. The Arducopter 3DR-B is designed and manufactured at the 3D Robotics headquarters in San Diego, California.



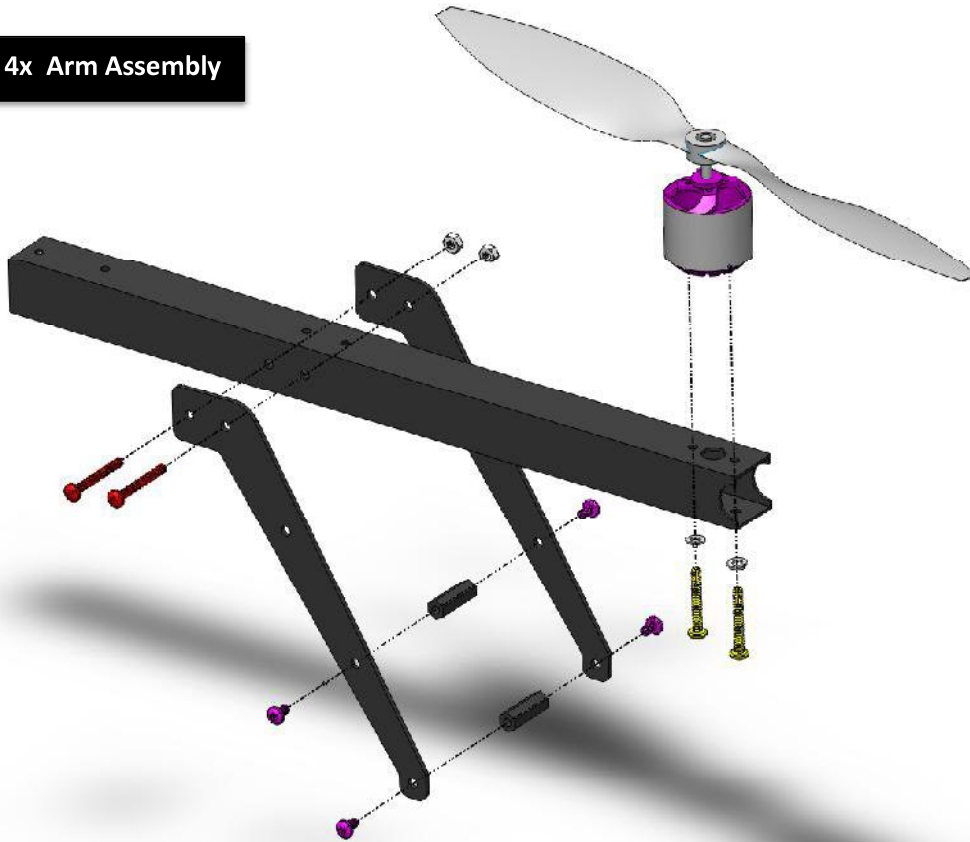
Arducopter 3DR-B Hardware

Name		Qty.
M3x30mm Spacer.....		04
M3x18mm Spacer.....		12
M3x08mm Spacer.....		04
■ M3x30mm SS Screw.....		04
■ M3x25mm SS Screw.....		12
■ M3x22mm Zinc Screw.....		08
■ M3x05mm SS Screw.....		16
■ M3x05mm Nylon Screw.....		08
Rubber Washer.....		04
M3 Metal Hex Nut.....		16
M3 Nylon Hex Nut.....		04
M3 Lock Washer.....		08



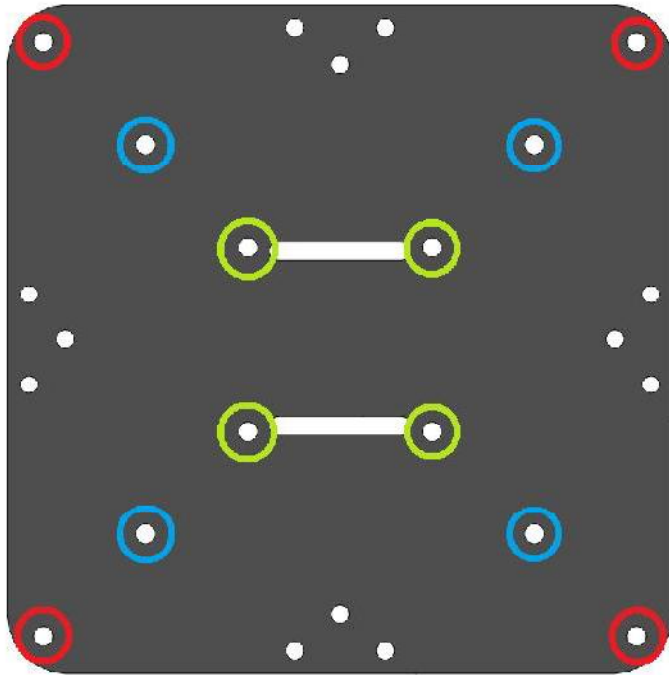
Assembly Guide

4x Arm Assembly



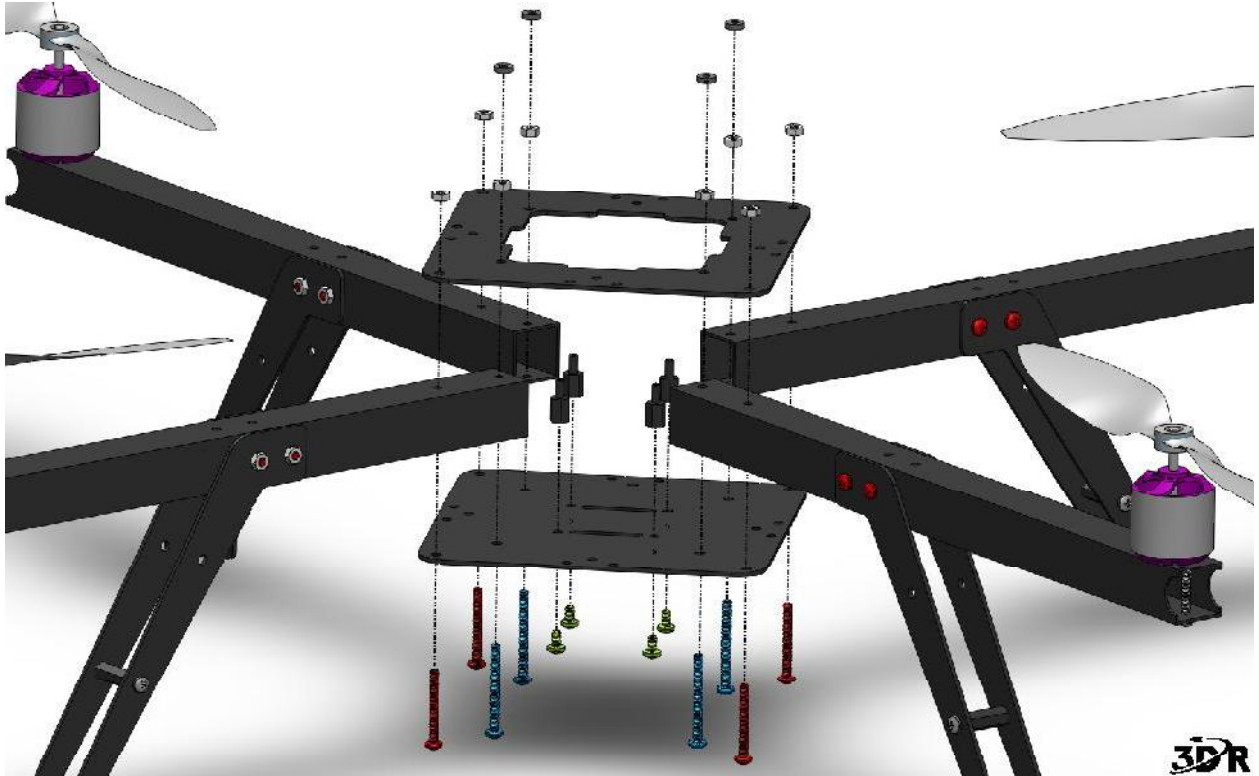
Make sure the motor holes on the arm are facing up. The legs are mounted to the arms using 2x M3x25mm SS Screws (Red) and 2x M3 Metal Hex Nuts. Insert two M3x18mm spacers in between the legs and fasten with 4x M3x5mm screws (Purple) for support. The motors are attached to the arms with 2x M3x22mm zinc plated screws (Yellow) and 2x M3 Lock Washers (Make sure the screws go into the threaded holes in the motors and not the ventilation grooves). Complete the assembly of all four arms.





Main Body Assembly

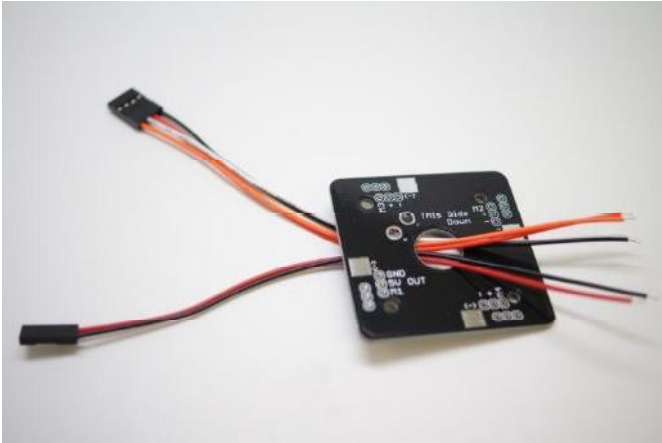
-  M3x25mm SS Screw
-  M3x30mm SS Screw
-  M3x05mm Nylon Screw



Assemble the main body of the Arducopter 3DR-B as shown above. The top and bottom bases are fastened to the four Arm sub-assemblies using the hardware indicated on the previous figures. The outermost screws (shown in Red) are M3x25mm SS Screws fastened to a M3 Metal Hex Nut. The screws shown in Blue are longer (M3x30mm) and will be used to support the stack-up later. These are also fastened with a M3 Metal Hex Nut, but a Rubber Washer is also added on top of the Hex Nut. Finally, install 4x M3x08 Nylon Spacers in the middle using M3x05mm Nylon Screws (Shown in Green). Slide the velcro strap through the grooves in the center. This will be used to hold the battery in place.

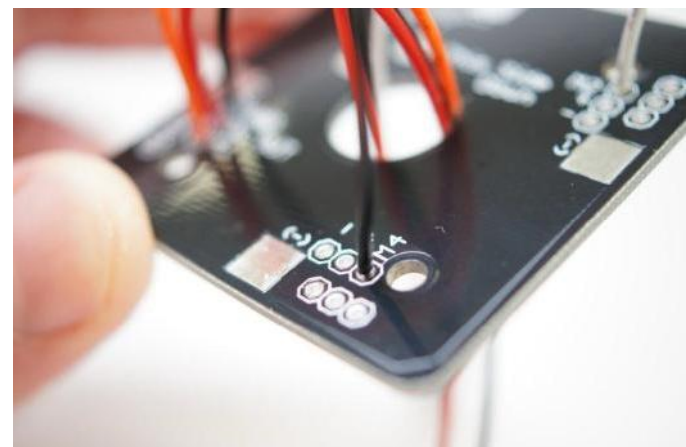
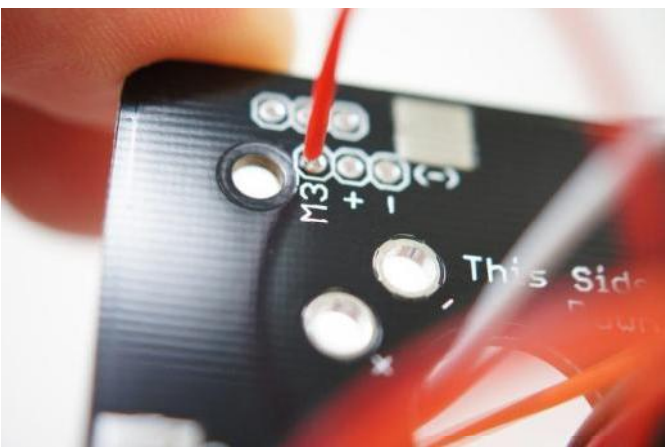
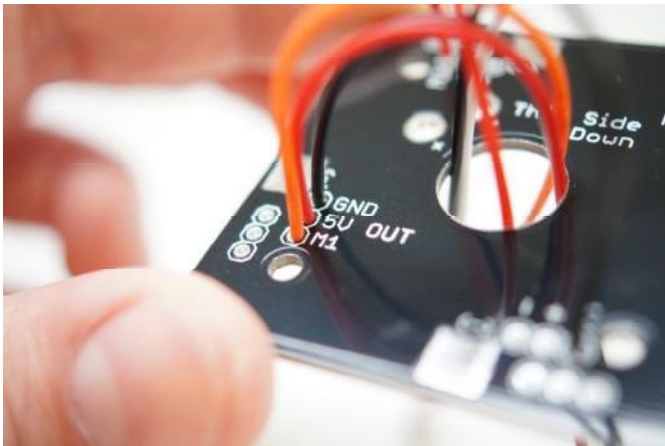


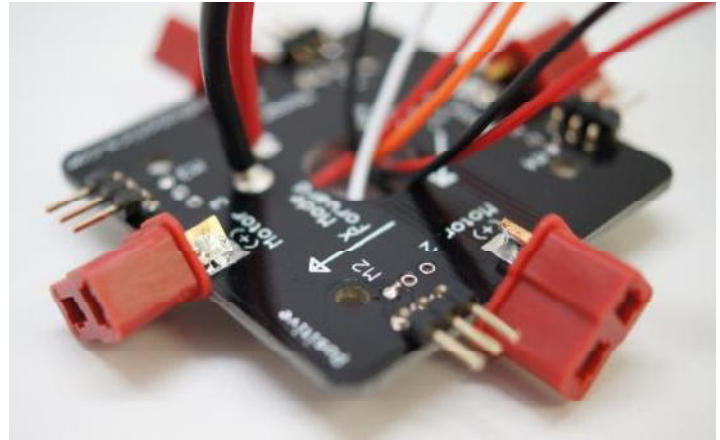
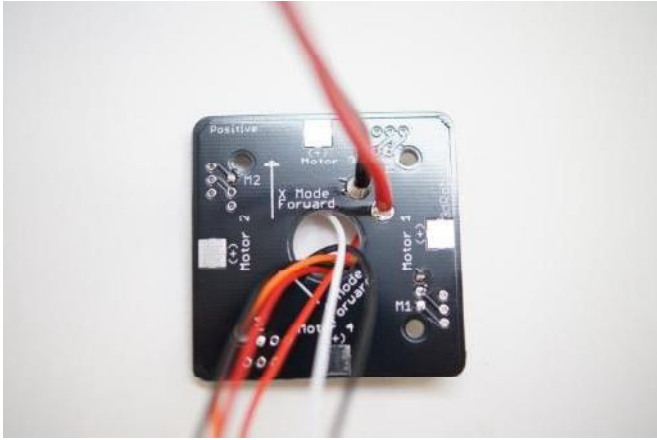
PDB Assembly



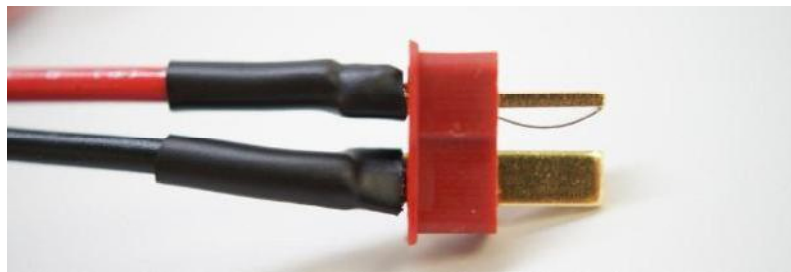
Run the two sets of narrow gauge wire through the central hole. The stripped ends of the wires should all emerge on the bottom side of the PDB, the side that says "This Side Down". Solder the two wire red and black connector to 5V Out and GND respectively.

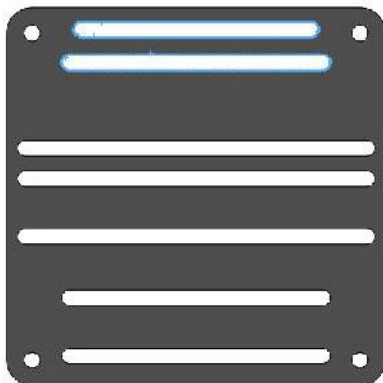
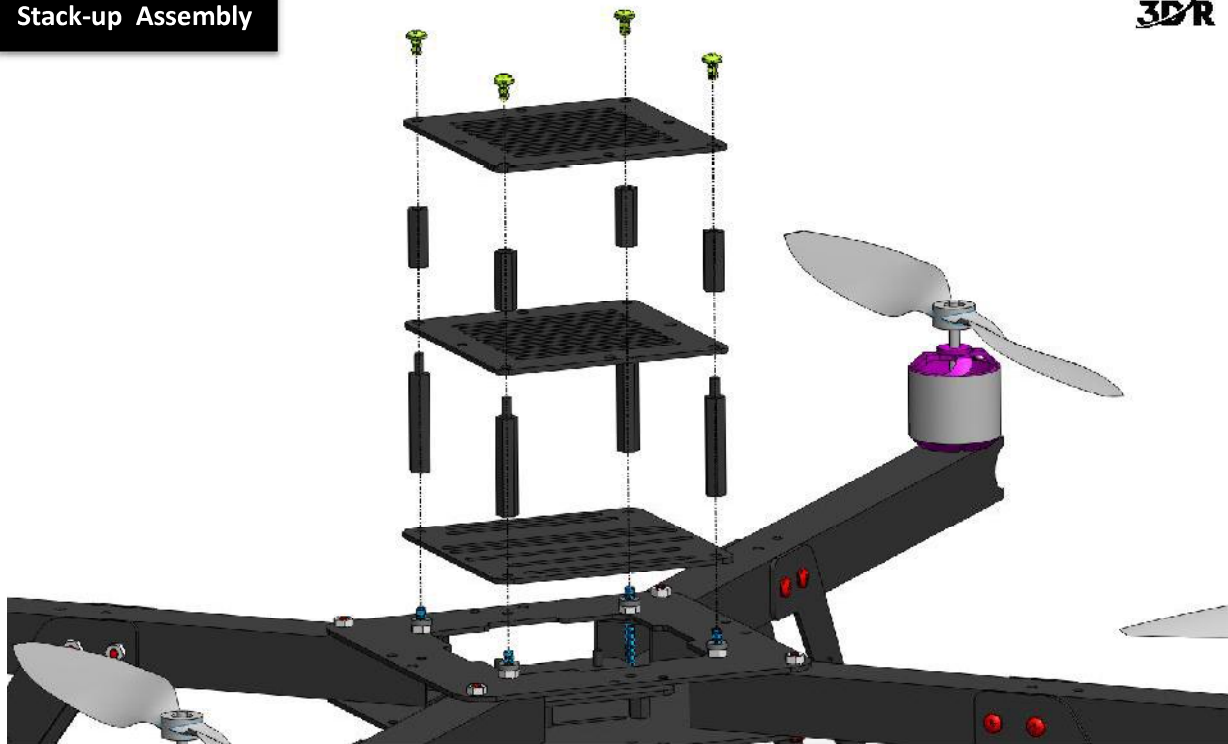
Next solder the four wire connector starting with the orange cable to **M1**, white to **M2**, red to **M3**, and black to **M4**. Use the pictures below for reference.





Next, strip both ends of the thick gauge red and black wires about 4mm. Solder the black wire into the large diameter hole marked “-” and the red wire into the one marked “+” . Slide a piece of shrink tubing into each cable but don’t shrink it yet. Solder a Male Dean’s Plug to the other end of the thick wires matching the red wire to the “+” on the connector and the black wire to the “-” sign. Pull the shrink tubing over the exposed connector leads and shrink it. Finally solder 3 pin headers into the open holes and female Dean’s connectors onto the exposed pads on edge of the PDB board. Make sure to match the “+”, “-” markings on the Dean’s connector.

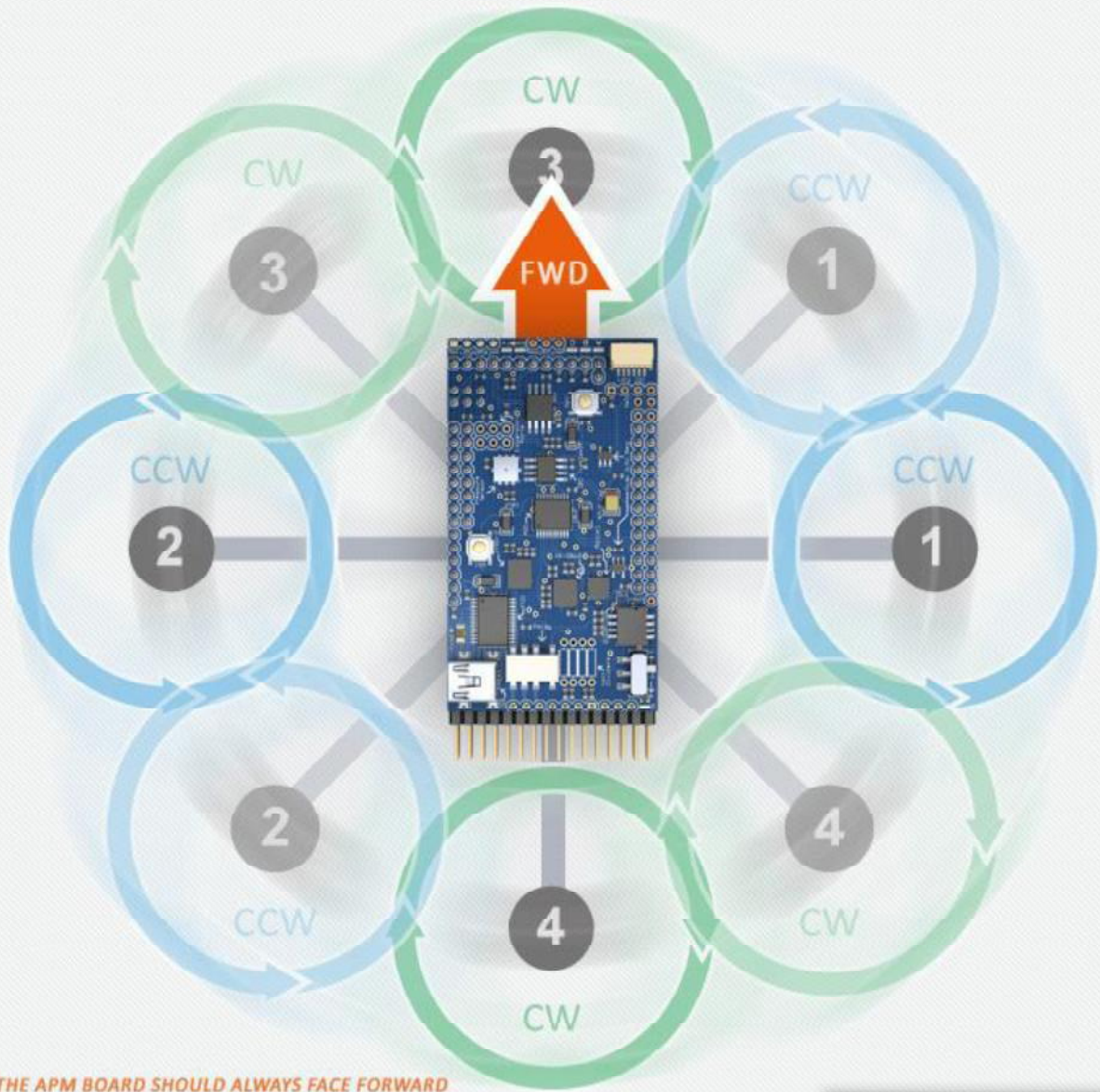




Install the PDB in the center and secure using 4x M3 Nylon Hex nuts. Next install the Base Cap, note that the two slots close together (marked in **Blue**) mark the front side of the quad. Align them with the front arrow on your pdb. For setting correct motor orientation please visit the arducopter wiki (<http://code.google.com/p/arducopter>). The Base Cap allows for easy access to the PDB as well as the motor wires. Screw 4x M3x30mm Spacers to hold the Base Cap in place. The stack-ups fit right on top secured on top by 4x M3x05mm Nylon screws (**Green**).

To attach your APM board to the Base Cap use double sided tape or screws. The Base cap slot pattern allows for your APM1 or APM2 to be mounted in either "X" or "+" configurations. Refer to the figures in the following pages for correct motor numbering and plug in the signal cables from your ESCs to the PDB accordingly. Remember the on the four wire connector, the orange cable is connected to **M1**. Use this as a reference when connecting the four wire connector to the APM outputs. **Orange** should go to output 1.

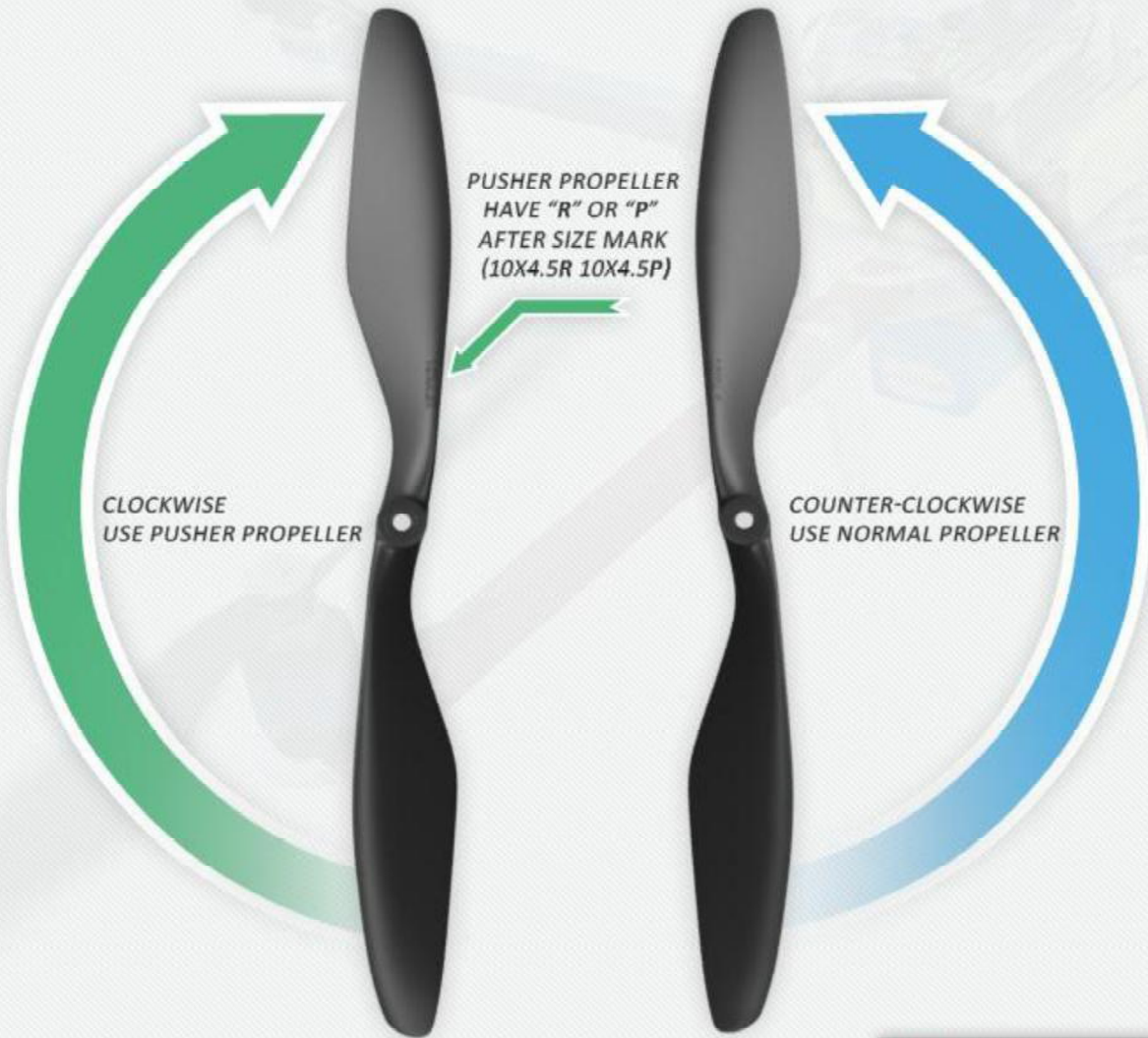
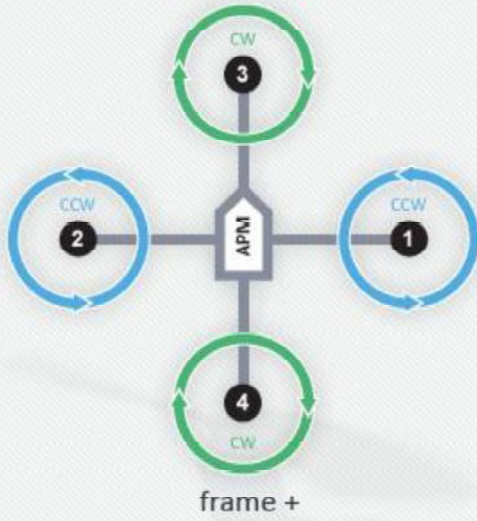
APM BOARD ORIENTATION



THE APM BOARD SHOULD ALWAYS FACE FORWARD
REGARDLESS OF THE FRAME TYPE AND ORIENTATION

DIY DRONES

PROPELLERS ORIENTATION



DIY DRONES



We hope you enjoy your Arducopter 3DR-B. If you have any questions or concerns please feel free to contact us via email at :

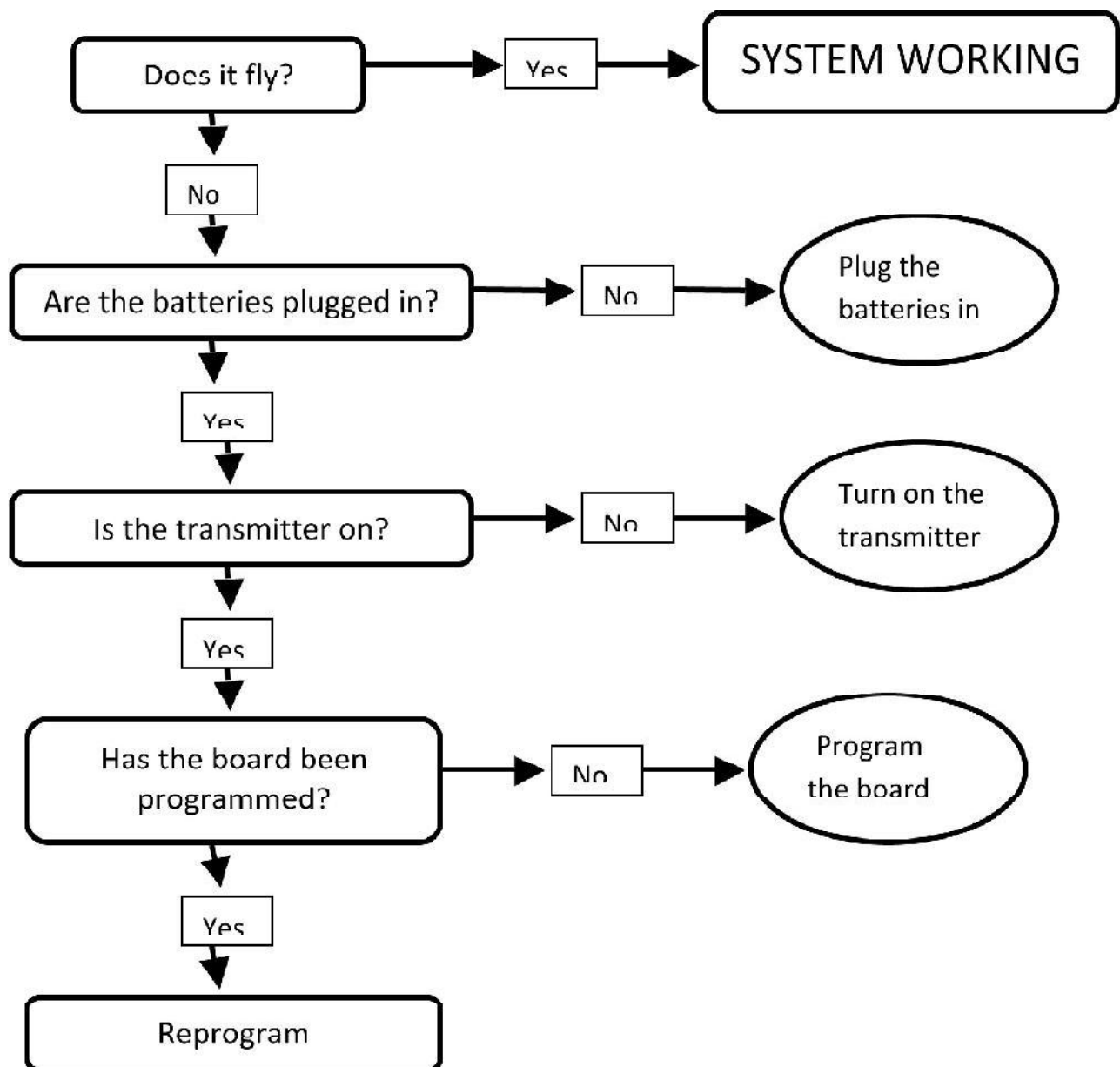
help@3drobotics.com

For additional information on how to set up your Arducopter 3DR and more information on the Arducopter codebase please visit the Arducopter wiki at:

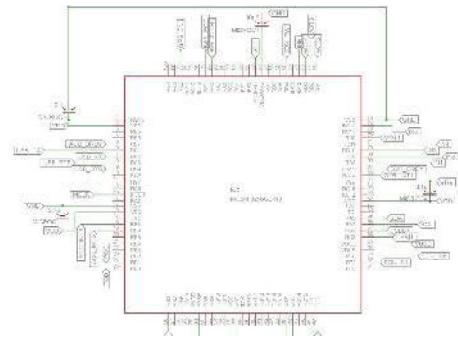
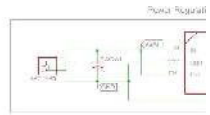
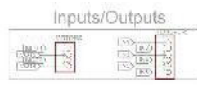
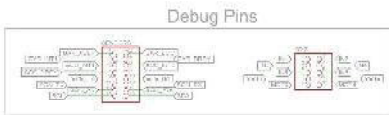
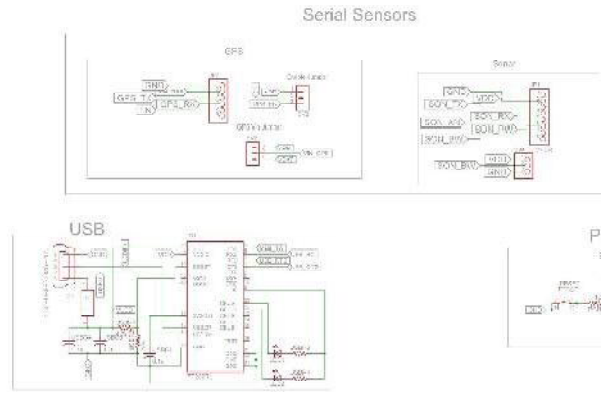
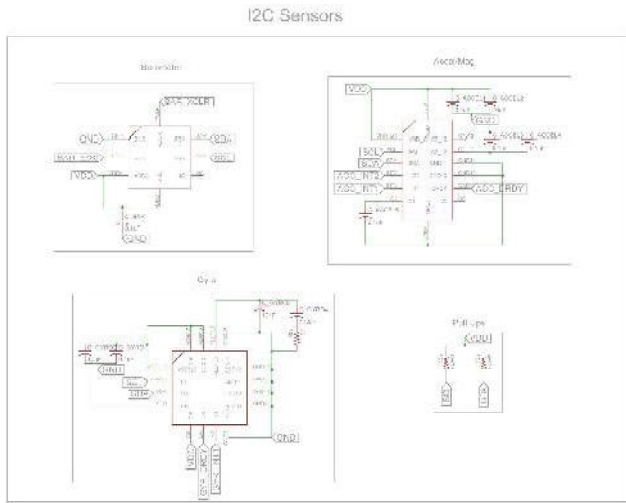
<http://code.google.com/p/arducopter>



B Debugging the system



C Eagle Documentation



Team Name	Part Description	Source/Supplier	Part Number	Quantity
Radio Flyers	MediaTek MT3329 GPS 10Hz	diydrones	MT3329	1
Radio Flyers	Barometric Pressure Sensor - BMP085	Digikey	828-1005-2-ND	1
Radio Flyers	Ultrasonic Range Finder - Maxbotix LV-EZ1	sparkfun electronics	LV-EZ1	1
Radio Flyers	Accelerometer/Magnetometer	Digikey	497-11918-1-ND	1
Radio Flyers	Triple Axis Digital Output Gyroscope	Digikey	497-11071-1-ND	1
Radio Flyers	ZIPPY Flightmax 2200mAh 3S1P 20C	HobbyKing	Z22003S20C	4
Radio Flyers	Polymer Lithium Ion Battery - 110mAh	sparkfun electronics	PRT-00731	2
Radio Flyers	Futaba 6EXP 6-Channel FM Radio System w/R156F Receiver	Tower Hobbies	LXRXF4**	1
Radio Flyers	ArduCopter 3DR Quad KIT, Electronics	diydrones		1
Radio Flyers	Voltage Regulators - Linear (LDO)	Digikey	AP7311-33WG-7DICT-ND	1
Radio Flyers	Mini USB connector	Digikey	A31727CT-ND	1
Radio Flyers	Ceramic capacitor 0.47 uF	Digikey	587-1261-1-ND	1
Radio Flyers	Ceramic capacitor 10000 pF	Digikey	478-1227-1-ND	1
Radio Flyers	Diode	Digikey	641-1003-1-ND	1

Cost/piece	Total Cost	Link
\$29.990	\$29.99	https://store.diydrones.com/MediaTek_MT3329_GPS_10Hz_p/mt3329-01.htm
\$3.814	\$3.81	http://search.digikey.com/us/en/products/BMP085/828-1005-2-ND/1986996
\$25.950	\$25.95	http://www.sparkfun.com/products/639
\$8.430	\$8.43	http://search.digikey.com/us/en/products/L_SM303DLHCTR/497-11918-1-ND/2757636
\$12.950	\$12.95	http://search.digikey.com/us/en/products/L_3G4200DTR/497-11071-1-ND/2587903
\$8.990	\$35.96	http://www.hobbyking.com/hobbyking/store/uh_viewItem.asp?idProduct=6306
\$6.950	\$13.90	http://www.sparkfun.com/products/731
\$129.990	\$129.99	http://www.3.towerhobbies.com/cgi-bin/wti0001p?&l=L_XRXF4**&P=FR
\$589.000	\$589.00	https://store.diydrones.com/ArduCopter_3DR_Quad_KIT_Electronics_p/kt-ac3dr-03.htm
\$0.540	\$0.54	http://search.digikey.com/us/en/products/AP7311-33WG-7/AP7311-33WG-7DICT-ND/2270836
\$1.510	\$1.51	http://search.digikey.com/us/en/products/1734035-2/A31727CT-ND/773789
\$0.180	\$0.18	http://search.digikey.com/us/en/products/FMK107F474ZA-T/587-1261-1-ND/931038
\$0.060	\$0.06	http://search.digikey.com/us/en/products/06035C103KAT2A/478-1227-1-ND/564259
\$0.350	\$0.35	http://search.digikey.com/us/en/products/CDSU400B/641-1003-1-ND/1121125
Total	\$852.62	

Manufacturer	Manufacturer part number	Notes
Mediatek	3329	
Bosch	BMP085	
MaxBotix	LV-EZ1	
STMicroelectronics	LSM303DLHCTR	
STMicroelectronics	L3G4200DTR	
ZIPPY	3S1P	
UNIONFORTUNE	41528	
Futaba	FUTK63**	No Preference for frequency channel
DIY Drones		
Diodes Inc.	AP7311-33WG-7	
TE Connectivity	1734035-2	
Taiyo Yuden	EMK107F474ZA-T	
AVX Corporation	06035C103KAT2A	
Comchip Technology	CDSU400B	

E Code Listing

QuadMain

```
/*  
* File: QuadMain.c  
* Author: John J. Walsh  
*  
* Created on April 3, 2012, 4:31 PM  
*/
```

```
/*  
*****  
* Software License Agreement  
*  
* Copyright © 2012 RadioFlyers and Walsh Inc. All rights reserved.  
* RadioFlyers licenses to you the right to use, modify, copy, distribute, own  
* play with, tinker around, test, modify again, get pissed at, and insult  
* this code only when done at the University of Notre Dame, or in fact whenever  
* you feel like it. It is recommended, however, that should you be just  
* starting this project now, that you immediately stop and pick another  
* project. We would recommend a project that cooks steak. This way you get  
* to eat steak with the University's money. That or a perpetual motion  
* machine. Those are pretty baller and probably much simpler than this  
* project is. If, however, you have finalized the project and are stuck with  
* it, I recommend the following steps. Move the computer to the side, stand  
* up, lean, bend over, pray to whichever god you like, kiss your ass goodbye.  
*  
* You should refer to the individual comments for each section and function  
* for specific concerns and questions. Just about all of this code and  
* comments were written by John J. Walsh (except anything in the PWM.h file.  
* all of that was written by Jay Burns, who coincidentally is a pilot....who  
* knew....). I take great pride in the fact that I didn't comment shit while  
* writing the code, and have gone back now and put in these comments for your  
* damn benefit. That being said, this is being done during finals week,  
* second semester senior year, so my fucks-given meter is at a pretty all time  
* low. So, if comments are not to your liking, and you don't understand what's  
* going on, feel free to get a piece of paper, write down your question/comment,  
* walk to the nearest trash can, and throw the piece of paper on the floor as  
* hard as you can... On a serious note, if you manage to read all this, and  
* figure out a way to contact me, I would be thoroughly impressed and would  
* probably answer any question you had.  
*  
*****/  
*/
```

```
#if defined(__PIC24E__)  
#include <p24Exxxx.h>  
  
#elif defined (__PIC24F__)  
#include <p24Fxxxx.h>  
  
#elif defined(__PIC24H__)  
#include <p24Hxxxx.h>  
  
#elif defined(__dsPIC30F__)  
#include <p30Fxxxx.h>  
  
#elif defined (__dsPIC33E__)  
#include <p33Exxxx.h>  
  
#elif defined(__dsPIC33F__)  
#include <p33Fxxxx.h>  
  
#endif  
  
#define FCY 16000000UL // 16 MHz cycle time for delay routines
```

QuadMain

```
#include<stdio.h>
#include"TerminalUart.h"
#include <libpic30.h>
#include <p24FJ256GB110.h>
#include <stdlib.h>
//#include "Analog.h"
#include <stdbool.h>
#include"I2C.h"
#include"Misc.h"
#include"PWM.h"
#include<math.h>

//_CONFIG3(SOSCSEL_EC)
_CONFIG1 (FWDTEN_OFF & JTAGEN_OFF )
_CONFIG2 (PLLDIV_NODIV & FNOSC_FRCPLL & FCKSM_CSECME & POSCMOD_NONE)

int MotorThrusts[4];

//      Function Prototypes
int main(void);

int main(void) {
    /*****
     * Initializing variables
     *****/
    MotorThrusts[0] = 0;
    MotorThrusts[1] = 0;
    MotorThrusts[2] = 0;
    MotorThrusts[3] = 0;
    char string[20];
    int NewTotalThrust = 0, TotalThrust = 0, DeltaThrust = 0;
    TotalThrust = 0;
    int i;
    bool first = 1;
    bool Switch = 1;
    float targetX = 0;
    float targetY = 0;
    unsigned int pulses;
    float DerivativeX = 0, DerivativeY = 0, IntegralX = 0, IntegralY = 0, errorX =
0, errorY = 0, lastErrorX = 0, lastErrorY = 0, NextIntX = 0, NextIntY = 0, time = 0,
temp, ThetaX, ThetaY;
    short int Px = 0, Py = 0, Ix = 0, Iy = 0, Dx = 0, Dy = 0, Change23 = 0, Change34
= 0;
    int MotMin = 0, TempDel1 = 0, TempDel2 = 0, Delta = 0, MotMax = 0;

    /*****
     * These are the values that you change to manipulate the P.I.D controler
     * The max values are the maximum contribution each of the components can have
     * Kp is for the proportional term
     * Ki is for the integral term
     * Kd is for the derivative term
     *****/
    int Imax = 4, Pmax = 4, Dmax = 3;
    float Kp = 0.6;
    float Ki = 1;
    float Kd = 0.1;
```

QuadMain

```
/*
 * Here starts the initialization routine
 */
init_usart(103);
myputc('\f');
myputs("Initializing.");
init_I2C(0x35);
myputc('.');
Acc_init();
myputc('.');
Mag_init();
myputc('.');
Bar_getParam();
myputc('.');
Gyro_init();
myputc('.');
initializePWMoutput();
myputc('.');
initializePWMcapture();
myputc('.');
init_TMR1();
myputc('.');

/*
 * This is just a simple stoping mechanism to stop the code before all the
 * offsets are determined. This allows one to ensure the QuadRotor is level
 */
myputc('\n');
myputc('\r');
myputs("Ensure switch is off...");
getPWM();
while(Switch == 1)
{
    readCH5();
    Switch = CH5status;
}

myputc('\n');
myputc('\r');
myputs("Ensure QuadRotor is level and stable");
myputc('\n');
myputc('\r');
myputs("Getting offsets.");
Acc_getZoff();
myputc('.');
Gyro_getOff();
PrintOffs();

/*
 * This is put in here to initialize the ESCs should they need it. Otherwise
 * one can take the time here to test out the throttle. If the ESCs need to be
 * calibrated put the throttle in low, then high for 4 sec, then low again.
 */
myputc('\n');
myputc('\r');
myputs("Turn switch on to start initialization...");
while(Switch == 0)
{
    readCH5();
    Switch = CH5status;
```


QuadMain

```
}
myputc('\n');
myputc('\r');
myputs("Turn switch off when finished...");
while(Switch == 1)
{
    getPWM();
    Switch = CH5status;
    setPower(CH3value, CH3value, CH3value, CH3value);
}
myputc('\n');
myputc('\r');
myputs("Turn switch on when ready to fly!");
while(Switch == 0)
{
    readCH5();
    Switch = CH5status;
}

/*****
* Here begins the main loop of the code. The outer while loop is to run
* constantly and to allow us to have a kill switch (currently on ch 5) to
* turn the QuadRotor off and on at any point
*****/
while (1){
    readCH5();
    CH5status = 1;
    Switch = CH5status;

    for (i = 0; i<4; i++)
    {
        MotorThrusts[i] = 0; // set all motors to 0
    }
    TotalThrust = 0;

    setPower(0, 0, 0, 0);

    IntegralX = 0; // clear integral terms of PID
    IntegralY = 0;

    first = 1; // clear derivative terms of PID
    lastErrorX = 0;
    lastErrorY = 0;

    /*****
    * This starts the main loop inside the main loop. This is primarily to
    * run our PID controler
    *****/
    while (Switch == 1)
    {
        //////////////////////////////////////
        // Get Thrust input
        //////////////////////////////////////
        getPWM();
        Switch = CH5status;

        CH3value = 25;
        NewTotalThrust = CH3value*4; // Turn input into
'overall thrust input'
```

QuadMain

```

MotMax = CH3value + 20;
if (MotMax > 170)
{
    MotMax = 170;
}
MotMin = CH3value - 10;
if (MotMin < 0)
{
    MotMin = 0;
}

DeltaThrust = (NewTotalThrust - TotalThrust)>>2;    // Change for each motor
(initially all increased equally)
TotalThrust = NewTotalThrust;
for (i = 0; i<4; i++)
{
    MotorThrusts[i] = MotorThrusts[i]+DeltaThrust; // Increase all the
motors equally
}

////////////////////////////////////
// Get data
////////////////////////////////////
Acc_getAccVec();
Mag_getMagVec();
Gyro_getRotVec();

////////////////////////////////////
// Print Data
////////////////////////////////////
//PrintData();

/*
 * Quick attempt at determining Theta values. Not entirely sure whether
 * it really works or not, gave up on it due to time

ThetaX = atan2(AccVec[1],AccVec[2]);
ThetaY = atan2(AccVec[0],AccVec[2]);

myputc('\n');
myputc('\r');
sprintf(string, "%f",ThetaX);
myputs(string);
myputc(',');
sprintf(string, "%f",ThetaY);
myputs(string);
myputc('\n');
myputc('\r');
*/

////////////////////////////////////
// QuadRotor Orientation & motor numbers + for counter-clockwise - for
clockwise
//          (-)2 \ 1(+)
//                \^/
//                |^|
//                / \
//          (+)3 / 4(-)
//
//          |y
//          |___ x //Motor locations and axis
////////////////////////////////////

```

QuadMain

```
////////////////////////////////////
// P.I.D. Controller
// 3 sets to change, motors 1&4, motors 1&2, and motors 1&3
// correspond to      X accel,      Y accel,      and Z rot
////////////////////////////////////

/*****
* P section of PID controler. This controls proportional response to
* some offset. Our PID controler is simply trying to drive acceleration
* in the X and Y directions relative to the QuadRotor frame to zero. This
* sort of works, but I would advise you to try and develop system to
* determine orientation angles and drive those to given value
*****/

errorX = AccVec[0]-targetX;
errorY = AccVec[1]-targetY;
//errorZ = RotVec[3]-targetZ;           // Never got to implementing it for
rotations about z axis

    if (errorX >=0)
    {
        Px = ceil(Kp * round(errorX*10));           // Nets a range over
which Px is increased                               // 0->0.05 = 0,
    }
0.05->0.3 = 1, 0.3->0.6 = 2, 0.6-> = 3
    else
    {
        Px = floor(Kp * round(errorX*10));
    }

    if (errorY >=0)
    {
        Py = ceil(Kp * round(errorY*10));           // Nets a range over
which Px is increased                               // 0->0.05 = 0,
    }
0.05->0.3 = 1, 0.3->0.6 = 2, 0.6-> = 3
    else
    {
        Py = ceil(Kp * round(errorY*10));
    }

/*
myputc('<');
myputi(Px);
myputc(',');
myputi(Py);
myputc('>');
*/

/*****
* This starts the integral part of the PID. First thing to do is determine
* the amount of time that has passed. Pretty sure the algorithm to do
* that works, but never fully tested it. One might want to look into it.
*****/

if (first)           // In here to disregard data from first loop through
{
```

QuadMain

```

    first = !first;
    time = 0;
}
else
{
pulses = TMR1;
TMR1 = 0;
temp = (float)pulses;
time = temp*64/16000000;
}

if ( errorX>=-0.05 && errorX<=0.05)
{
    NextIntX = 0;
}
//////////////////////////////////////
else // yields increase of 1 if error
is 0.1 for 1 sec
{
    NextIntX = errorX*10*time;
}

if ( errorY>=-0.05 && errorY<=0.05)
{
    NextIntY = 0;
}
else
{
    NextIntY = errorY*10*time;
}

IntegralX = IntegralX + NextIntX;
IntegralY = IntegralY + NextIntY;

if (IntegralX > Imax)
{
    IntegralX = Imax;
}
else if (IntegralX < Imax*-1)
{
    IntegralX = -1*Imax;
}
if (IntegralY > Imax)
{
    IntegralY = Imax;
}
else if (IntegralY < Imax*-1)
{
    IntegralY = -1*Imax;
}
//IntegralZ = IntegralZ + errorZ*time;

if (errorX >=0)
{
    Ix = floor(Ki*IntegralX);
}
else
{
    Ix = ceil(Ki*IntegralX);
}

if (errorY >=0)

```

QuadMain

```

{
    Iy = floor(Ki*IntegralY);
}
else
{
    Iy = ceil(Ki*IntegralY);
}

/*
myputc('<');
myputi(Ix);
myputc(',',');
myputi(Iy);
myputc('>');
*/

/*****
* This begins the derivative section of the PID controler. Never really
* had a good grasp of what I wanted the derivative term to do, so kinda
* just pulled numbers out of thin air to come up with ordinal equation.
* Might want to come up with better way of determining Derivative terms.
* Also, hesitant about its full functionality because Derivative first
* time thought should be infinity (because time is 0), but it doesn't
* seem to be. Should check that out.
*****/

DerivativeX = (errorX-lastErrorX)/time;
DerivativeY = (errorY-lastErrorY)/time;
//DerivativeZ = (errorZ-lastErrorZ)/time;

if (errorX >=0)
{
    Dx = floor(DerivativeX*Kd);           ////////////////
}                                         // Yields change of 1 for change of
0.1g in 0.1 sec
else
{
    Dx = ceil(DerivativeX*Kd);
}

if (errorY >=0)
{
    Dy = floor(DerivativeY*Kd);         ////////////////
}                                         // Yields change of 1 for change of
0.1g in 0.1 sec
else
{
    Dy = ceil(DerivativeY*Kd);
}

/*
myputc('<');
myputi(Dx);
myputc(',',');
myputi(Dy);
myputc('>');
*/

/*****

```

QuadMain

* Ensure all values under max

```

*****/
if (Px > Pmax)
{
    Px = Pmax;
}
else if (Px < -1 * Pmax)
{
    Px = -1 * Pmax;
}
if (Py > Pmax)
{
    Py = Pmax;
}
else if (Py < -1 * Pmax)
{
    Py = -1 * Pmax;
}
if (Ix > Imax)
{
    Ix = Imax;
}
else if (Ix < -1 * Imax)
{
    Ix = -1 * Imax;
}
if (Iy > Imax)
{
    Iy = Imax;
}
else if (Iy < -1 * Imax)
{
    Iy = -1 * Imax;
}
if (Dx > Dmax)
{
    Dx = Dmax;
}
else if (Dx < -1 * Dmax)
{
    Dx = -1 * Dmax;
}
if (Dy > Dmax)
{
    Dy = Dmax;
}
else if (Dy < -1 * Dmax)
{
    Dy = -1 * Dmax;
}

/*****
* Put all the terms together. Might want to look into pulling the K
* values outside of the functions so they can just be multiplied
* directly to the P, I, and D terms respectively
*****/
*****/
Change23 = Px + Ix + Dx;
Change34 = Py + Iy + Dy;
// Change24 = Kp*errorZ + Ki*IntegralZ + Kd*DerivativeZ;

```

QuadMain

```
////////////////////////////////////
// Change individual motor thrusts
////////////////////////////////////

/*      Over all form. Broken up to ensure total thrust remains 'constant'
MotorThrusts[1-1] = MotorThrusts[1-1] - Change23 - Change34; //- Change24;
MotorThrusts[2-1] = MotorThrusts[2-1] + Change23 - Change34; //+ Change24;
MotorThrusts[3-1] = MotorThrusts[3-1] + Change23 + Change34; //- Change24;
MotorThrusts[4-1] = MotorThrusts[4-1] - Change23 + Change34; //+ Change24;
*/

/*****
* This following is just a whole lot of coding to ensure that the
* 'total thrust' of all 4 motors remains a constant. This is done by
* first adding the change in one direction, then seeing if that change
* resulted in a value over the allowed max or under allowed min, then
* correcting accordingly. Pretty sure this works as intended
*****/

// x dimension changes (rotation about y (yaw))
MotorThrusts[1-1] = MotorThrusts[1-1] - Change23;
MotorThrusts[2-1] = MotorThrusts[2-1] + Change23;
MotorThrusts[3-1] = MotorThrusts[3-1] + Change23;
MotorThrusts[4-1] = MotorThrusts[4-1] - Change23;

// Check if change23 sends a motor below 0
if (MotorThrusts[2-1] < MotMin || MotorThrusts[3-1] < MotMin)
{
    TempDel1 = MotMin - MotorThrusts[2-1];
    TempDel2 = MotMin - MotorThrusts[3-1];
    if (TempDel1 > TempDel2)
    {
        Delta = TempDel1;
    }
    else
    {
        Delta = TempDel2;
    }
}
else if (MotorThrusts[1-1] < MotMin || MotorThrusts[4-1] < MotMin)
{
    TempDel1 = MotMin - MotorThrusts[1-1];
    TempDel2 = MotMin - MotorThrusts[4-1];
    if (TempDel1 > TempDel2)
    {
        Delta = -1*TempDel1;
    }
    else
    {
        Delta = -1*TempDel2;
    }
}
else
{
    Delta = 0;
}
MotorThrusts[1-1] = MotorThrusts[1-1] - Delta;
MotorThrusts[2-1] = MotorThrusts[2-1] + Delta;
```

```

                                QuadMain
MotorThrusts[3-1] = MotorThrusts[3-1] + Delta;
MotorThrusts[4-1] = MotorThrusts[4-1] - Delta;

// Check if change sends motors above max
if (MotorThrusts[2-1] > MotMax || MotorThrusts[3-1] > MotMax)
{
    TempDel1 = MotMax - MotorThrusts[2-1];
    TempDel2 = MotMax - MotorThrusts[3-1];
    if (TempDel1 > TempDel2)
    {
        Delta = TempDel1;
    }
    else
    {
        Delta = TempDel2;
    }
}
else if (MotorThrusts[1-1] > MotMax || MotorThrusts[4-1] > MotMax)
{
    TempDel1 = MotMax - MotorThrusts[1-1];
    TempDel2 = MotMax - MotorThrusts[4-1];
    if (TempDel1 > TempDel2)
    {
        Delta = -1*TempDel1;
    }
    else
    {
        Delta = -1*TempDel2;
    }
}
else
{
    Delta = 0;
}
MotorThrusts[1-1] = MotorThrusts[1-1] - Delta;
MotorThrusts[2-1] = MotorThrusts[2-1] + Delta;
MotorThrusts[3-1] = MotorThrusts[3-1] + Delta;
MotorThrusts[4-1] = MotorThrusts[4-1] - Delta;

// y dimension changes (rotation about x (pitch))
MotorThrusts[1-1] = MotorThrusts[1-1] - Change34;
MotorThrusts[2-1] = MotorThrusts[2-1] - Change34;
MotorThrusts[3-1] = MotorThrusts[3-1] + Change34;
MotorThrusts[4-1] = MotorThrusts[4-1] + Change34;

// Check if change34 sends a motor below 0
if (MotorThrusts[3-1] < MotMin || MotorThrusts[4-1] < MotMin)
{
    TempDel1 = MotMin - MotorThrusts[3-1];
    TempDel2 = MotMin - MotorThrusts[4-1];
    if (TempDel1 > TempDel2)
    {
        Delta = TempDel1;
    }
    else
    {
        Delta = TempDel2;
    }
}
}

```



```

                                QuadMain
else if (MotorThrusts[1-1] < MotMin || MotorThrusts[2-1] < MotMin)
{
    TempDel1 = MotMin - MotorThrusts[1-1];
    TempDel2 = MotMin - MotorThrusts[2-1];
    if (TempDel1 > TempDel2)
    {
        Delta = -1*TempDel1;
    }
    else
    {
        Delta = -1*TempDel2;
    }
}
else
{
    Delta = 0;
}
MotorThrusts[1-1] = MotorThrusts[1-1] - Delta;
MotorThrusts[2-1] = MotorThrusts[2-1] - Delta;
MotorThrusts[3-1] = MotorThrusts[3-1] + Delta;
MotorThrusts[4-1] = MotorThrusts[4-1] + Delta;

// check if change sends motors above max
if (MotorThrusts[3-1] > MotMax || MotorThrusts[4-1] > MotMax)
{
    TempDel1 = MotMax - MotorThrusts[3-1];
    TempDel2 = MotMax - MotorThrusts[4-1];
    if (TempDel1 > TempDel2)
    {
        Delta = TempDel1;
    }
    else
    {
        Delta = TempDel2;
    }
}
else if (MotorThrusts[1-1] > MotMax || MotorThrusts[2-1] > MotMax)
{
    TempDel1 = MotMax - MotorThrusts[1-1];
    TempDel2 = MotMax - MotorThrusts[2-1];
    if (TempDel1 > TempDel2)
    {
        Delta = -1*TempDel1;
    }
    else
    {
        Delta = -1*TempDel2;
    }
}
else
{
    Delta = 0;
}
MotorThrusts[1-1] = MotorThrusts[1-1] - Delta;
MotorThrusts[2-1] = MotorThrusts[2-1] - Delta;
MotorThrusts[3-1] = MotorThrusts[3-1] + Delta;
MotorThrusts[4-1] = MotorThrusts[4-1] + Delta;

// Making sure it never goes above 100 or below 0
for (i = 0; i<4; i++)

```


I2C

```
/*
 * This header file holds all the functions necessary for I2C to work as well
 * as communicating and getting data from the I2C sensors (accel, magnetometer,
 * gyro, and barometer). As of the end of the semester, all of the sensors can
 * be written to and read from, but only the accel seems to be producing usable,
 * actual data. Not sure what is wrong with the others, but the data they give
 * is just not right. Probably in the functions, but could be an error in
 * hardware as well.
 */

#include<math.h>

/*
 * I2C addresses of the sensors
 */
char GyroAddw = 0b11010000;
char GyroAddr = 0b11010001;
char AccAddw = 0b00110010;
char AccAddr = 0b00110011;
char MagAddw = 0b00111100;
char MagAddr = 0b00111101;

/*
 * Global variables for all the data vectors and any off sets
 */
float AccVec[3]; // <x,y,z> current acceleration vector
float MagVec[3]; // <x,y,z> current Magnetometer vector
int RotVec[3]; // <x,y,z> current Angular velocity vector
int RotOff[3]; // <x,y,z> 0 offset for angular rotation
float OrientVec[3]; // <x,y,z> approximate vector of absolute angles
float AccZoff = 0;
float Temperature;

/*
 * This function initializes I2C. It takes in a char value for the baud rate
 * and sets the I2C to communicate at that speed. Note, the equation given in
 * the data sheet to determine what the rate char needs to be is incorrect. Not
 * really sure where the proper equation is found, but just kinda guessed at the
 * values.
 */
void init_I2C(char rate)
{
    TRISAbits.TRISA2 = 0;
    PORTAbits.RA2 = 0;
    int i;
    for (i = 0; i < 255; i++) // Put in to make sure sensors don't
    { //think they're supposed to be outputing
data
        PORTAbits.RA2 = !PORTAbits.RA2;
        __delay_us(1.25);
    }
    TRISAbits.TRISA2 = 1;
    I2C2BRG = rate;
    I2C2CONbits.I2CEN = 1;
    I2C2STATbits.IWCOL = 0;
    I2C2STATbits.I2COV = 0;
    IFS3bits.MI2C2IF = 0;
    IFS3bits.SI2C2IF = 0;
    return;
}

/*
 * Function to call whenever you want to issue a stop command in I2C
 */
```

I2C

```

*****
void I2Cstop(void)
{
    IFS3bits.MI2C2IF = 0;
    I2C2STATbits.BCL = 0;
    I2C2STATbits.IWCOL = 0;
    I2C2CONbits.PEN = 1;
}

/*****
* Function to call whenever you want to issue a start command in I2C
*****/
void I2Cstart(void)
{
    IFS3bits.MI2C2IF = 0;
    I2C2STATbits.BCL = 0;
    I2C2STATbits.IWCOL = 0;
    I2C2CONbits.SEN = 1;
    return;
}

/*****
* Function to call whenever you want to issue a repeated start command in I2C
*****/
void I2Crestart(void)
{
    IFS3bits.MI2C2IF = 0;
    I2C2STATbits.BCL = 0;
    I2C2STATbits.IWCOL = 0;
    I2C2CONbits.RSEN = 1;
    return;
}

/*****
* Function to get what was received by the microcontroller. Returns 8 bits in
* the Receive register.
*****/
short int I2Crx(bool ack)
{
    short int data;
    data = I2C2RCV;
    I2C2CONbits.ACKDT = ack;
    I2C2CONbits.ACKEN = 1;
    return data;
}

/*****
* Function to put the microcontroller in Rx mode
*****/
void I2Csetrx(void)
{
    IFS3bits.MI2C2IF = 0;
    I2C2STATbits.BCL = 0;
    I2C2STATbits.IWCOL = 0;
    I2C2CONbits.RCEN = 1;
    return;
}

/*****
* Function to transmit data out to the SDA bus. Takes in 8 bits of data to be
* sent out, returns whether previous command was acknowledged.
*****/
bool I2Ctx(char data)

```

I2C

```

{
    //bit ack;
    I2C2TRN = data;
    return I2C2STATbits.ACKSTAT;
    //return ack;
}

/*****
 * Funciton that returns whether something was acknoledged.
 *****/
bool I2Cack(void)
{
    //bit ack;
    return I2C2STATbits.ACKSTAT;
    //return ack;
}

/*****
 * Wait funciton used to have the microcontroller wait until I2C process has
 * been completed. This has to be called after EVERY I2C command given. This
 * was never straight up put in all the functions because a beter design practice
 * would be to establish a state machine and have the code run through states
 * instead of having the microcontroller kill time by simply waiting. Never got
 * around to making the state machine though.
 *****/
void I2Cwait(void)
{
    while (I2C2CONbits.SEN || I2C2CONbits.RSEN || I2C2CONbits.PEN ||
I2C2CONbits.RCEN || I2C2CONbits.ACKEN);
    {}
    while (!IFS3bits.MI2C2IF);
    {}
    IFS3bits.MI2C2IF = 0;           //IF flag set back to 0
}

////////////////////////////////////
// All of the Gyro Functions
// Gyro_init
// Gyro_getRotVec
// Gyro_getOff
// DeterminePos    <- doesn't work
////////////////////////////////////

/*****
 * Function to initialize the gyro. No inputs or outputs needed. For specifics
 * on what I am setting the gyro to do consult the datasheet
 *****/
void Gyro_init(void)
{
    bool ack;

    I2Cstart();
    I2Cwait();

    ack = I2Ctx(0b11010000);    //Device address
    I2Cwait();

    ack = I2Ctx(0x20);         //Mem address
    I2Cwait();

    ack = I2Ctx(0b11001111);    //Stuff written out
    I2Cwait();
    ack = I2Ctx(0x00);
}

```

I2C

```

I2Cwait();
ack = I2Ctx(0b00000000);
I2Cwait();
ack = I2Ctx(0x00);
I2Cwait();
ack = I2Ctx(0b01001010);
I2Cwait();

I2Cstop();
I2Cwait();

I2Cstart();
I2Cwait();

ack = I2Ctx(0b11010000);    //Device address
I2Cwait();

ack = I2Ctx(0x2E);         //Mem address
I2Cwait();

ack = I2Ctx(0b01011111);
I2Cwait();

I2Cstop();
I2Cwait();
}

/*****
* Function to get the 'Rotation Vector.' This is a vector of the angular
* velocity of the QuadRotor. This takes in no input, and changes the global
* variable.
* notes: Giving values about an order of magnitude off. Might have to do with
*        the way I'm getting 32 values at once and averaging. Not sure if
*        that is working properly.
*****/
void Gyro_getRotVec(void)
{
    bool ack;
    int i;
    short int Xlsb, Xmsb, Ylsb, Ymsb, Zlsb, Zmsb;
    int TempX=0, TempY=0, TempZ=0;
    I2Cstart();
    I2Cwait();
    ack = I2Ctx(GyroAddw);
    I2Cwait();
    ack = I2Ctx(0b10101000);
    I2Cwait();
    I2Crestart();
    I2Cwait();
    ack = I2Ctx(GyroAddr);
    I2Cwait();

    for (i = 0; i<16; i++)
    {
        I2Csetrx();
        I2Cwait();
        Xlsb = I2Ctx(0);
        I2Cwait();
        I2Csetrx();
        I2Cwait();
        Xmsb = I2Ctx(0);
        I2Cwait();
    }
}

```

I2C

```

I2Csetrx();
I2Cwait();
Ylsb = I2Crx(0);
I2Cwait();
I2Csetrx();
I2Cwait();
Ymsb = I2Crx(0);
I2Cwait();
I2Csetrx();
I2Cwait();
Zlsb = I2Crx(0);
I2Cwait();
I2Csetrx();
I2Cwait();
Zmsb = I2Crx(1);
I2Cwait();

TempX = TempX + ((Xmsb << 8) | xlsb );           ////////////////
TempY = TempY + ((Ymsb << 8) | ylsb );           // Create running average
TempZ = TempZ + ((Zmsb << 8) | zlsb );           ////////////////

}

I2Cstop();
I2Cwait();

TempX >>= 4;           ////////////////
TempY >>= 4;           // Divide by 32
TempZ >>= 4;           ////////////////
RotVec[0] = (TempX*7.629)-RotOff[0];
RotVec[1] = (TempY*7.629)-RotOff[1];
RotVec[2] = (TempZ*7.629)-RotOff[2];
}

/*****
* This function gets the offsets for the gyro by sampling 256 times, averaging,
* and then setting that value to the offset vector.
*****/
void Gyro_getOff(void)
{
    int DataX = 0;
    int DataY = 0;
    int DataZ = 0;
    int i;
    for (i=0;i<256;i++)
    {
        Gyro_getRotVec();
        DataX = DataX+RotVec[0];
        DataY = DataY+RotVec[1];
        DataZ = DataZ+RotVec[2];
        __delay_ms(80);
        myputc(' ');
    }
    RotOff[0] = (DataX>>8);
    RotOff[1] = (DataY>>8);
    RotOff[2] = (DataZ>>8);
}

/*****
* This function is supposed to be a simple integration. Doesn't work. Not
* sure if didn't work because gyro values sucked, or didn't work because some
* part of algorithmt sucked. Look into this heavily before using
*****/

```

```

void DeterminePos(void)
{
    int pulses;
    float time;
    T1CONbits.TON = 1;
    pulses = TMR1;
    TMR1 = 0;
    time = (float)pulses*8/16000000;
    OrientVec[0] = (float)RotVec[0]*time/1000 + OrientVec[0];
    OrientVec[1] = (float)RotVec[1]*time/1000 + OrientVec[1];
    OrientVec[2] = (float)RotVec[2]*time/1000 + OrientVec[2];
}

////////////////////////////////////
// Accel functions
// Acc_init
// Acc_getAccVec
// Acc_getZoff
////////////////////////////////////

/*****
 * Function to initialize the accelerometer. For specifics on what is being set
 * look at the datasheet.
 *****/
void Acc_init(void)
{
    bool ack;

    I2Cstart();
    I2Cwait();

    ack = I2Ctx(AccAddw);
    I2Cwait();
    ack = I2Ctx(0x20);
    I2Cwait();
    ack = I2Ctx(0b01000111);
    I2Cwait();
    I2Cstop();
    I2Cwait();

    return;
}

/*****
 * Function to get the acceleration vector. This gives acceleration in x, y,
 * and z. No inputs needed, and changes the global variables.
 * notes: This function definitely works, and was working really well. This is
 * why the entire PID controller is based off accelerations.
 */
void Acc_getAccVec(void)
{
    bool ack;
    short int xlsb, xmsb, ylsb, ymsb, zlsb, zmsb, Temp;
    float Tempf;
    I2Cstart();
    I2Cwait();
    ack = I2Ctx(AccAddw);
    I2Cwait();
    ack = I2Ctx(0b10101000);
    I2Cwait();
    I2Crestart();
    I2Cwait();
}

```


I2C

```

ack = I2Ctx(AccAddr);
I2Cwait();
I2Csetrx();
I2Cwait();
Xlsb = I2Crx(0);
I2Cwait();
I2Csetrx();
I2Cwait();
Xmsb = I2Crx(0);
I2Cwait();
I2Csetrx();
I2Cwait();
Ylsb = I2Crx(0);
I2Cwait();
I2Csetrx();
I2Cwait();
Ymsb = I2Crx(0);
I2Cwait();
I2Csetrx();
I2Cwait();
Zlsb = I2Crx(0);
I2Cwait();
I2Csetrx();
I2Cwait();
Zmsb = I2Crx(1);
I2Cwait();
I2Cstop();
I2Cwait();
Temp = ((Xmsb << 8) | Xlsb );
Tempf = (float)Temp;
Tempf = ((Tempf*2)/32768);
if (Tempf - AccVec[1] > 0.3 || AccVec[1] - Tempf > 0.3)
{
    Tempf = AccVec[1];
}
AccVec[1] = Tempf;
Temp = ((Ymsb << 8) | Ylsb );
Tempf = (float)Temp;
Tempf = ((Tempf*2)/32768);
if (Tempf - AccVec[0] > 0.3 || AccVec[0] - Tempf > 0.3)
{
    Tempf = AccVec[0];
}
AccVec[0] = Tempf;
Temp = ((Zmsb << 8) | Zlsb );
AccVec[2] = (float)Temp;
AccVec[2] = ((AccVec[2]*2)/32768) + AccZoff;
}

/*****
* Funcion to get Z offset of the accelerometer to ensure 1 g when stationary.
* Runs 256 times, takes an averages, subtracts that from 1 g, then always adds
* that number to Acceleration Vector.
*****/
void Acc_getZoff(void)
{
    float Data, Temp;
    Data = 0;
    int i;
    for (i=0;i<256;i++)
    {
        Acc_getAccVec();
    }
}

```

I2C

```

    Temp = AccVec[2];
    Data = Data+Temp;
  }
  AccZoff = 1-(Data/256);
}

```

```

////////////////////////////////////
// Magnetometer function
// Mag_init
// Mag_getMagVec
// Mag_getTemp
////////////////////////////////////

```

```

/*****
* Initialized magnetometer. For specifics on what is initialized, look at
* values sent, then look it up in the damn datasheet. Magnetometer never
* worked properly. Not sure if that's because of a hardware issue (had a wire
* running right over sensor which could produce magnetic field) or if its a
* software related thing.
*****/

```

```
void Mag_init(void)
```

```

{
    bool ack;

    I2Cstart();
    I2Cwait();

    ack = I2Ctx(MagAddw);
    I2Cwait();
    ack = I2Ctx(0x00);
    I2Cwait();
    ack = I2Ctx(0b10011000);
    I2Cwait();
    ack = I2Ctx(0b00100000);
    I2Cwait();
    ack = I2Ctx(0x00);
    I2Cwait();
    I2Cstop();
    I2Cwait();

    return;
}

```

```

/*****
* This function returns the magnetometer vector in x, y, and z. Again, this
* never really returned reasonable values. Possibly software related. The
* stupid fucking datasheet for this part says almost nothing about the data
* received, or how precise (how many bits long) it actually is. Therefore,
* not entirely sure what prescaling needs to be....just to re-iterate-fuck the
* datasheet.
*****/

```

```
void Mag_getMagVec(void)
```

```

{
    bool ack;
    short int Xlsb, Xmsb, Ylsb, Ymsb, Zlsb, Zmsb, Temp;
    I2Cstart();
    I2Cwait();
    ack = I2Ctx(MagAddw);
    I2Cwait();
    ack = I2Ctx(0x83);
    I2Cwait();
    I2Crestart();
    I2Cwait();

```

I2C

```

ack = I2Ctx(MagAddr);
I2Cwait();
I2Csetrx();
I2Cwait();
Xlsb = I2Crx(0);
I2Cwait();
I2Csetrx();
I2Cwait();
Xmsb = I2Crx(0);
I2Cwait();
I2Csetrx();
I2Cwait();
Ylsb = I2Crx(0);
I2Cwait();
I2Csetrx();
I2Cwait();
Ymsb = I2Crx(0);
I2Cwait();
I2Csetrx();
I2Cwait();
Zlsb = I2Crx(0);
I2Cwait();
I2Csetrx();
I2Cwait();
Zmsb = I2Crx(1);
I2Cwait();
I2Cstop();
I2Cwait();
Temp = ((Xmsb << 8) | Xlsb );
//Temp >>= 4;
MagVec[1] = (float)Temp;
MagVec[1] = (MagVec[1]*1.3/2048);
Temp = ((Ymsb << 8) | Ylsb );
//Temp >>= 4;
MagVec[0] = (float)Temp;
MagVec[0] = -1*(MagVec[0]*1.3/2048);
Temp = ((Zmsb << 8) | Zlsb );
//Temp >>= 4;
MagVec[2] = (float)Temp;
MagVec[2] = MagVec[2]*1.3/2048;
}

/*****
* This function was written to test the magnetometer by getting the temperature
* value from it. Never returned a valid temperature. Again, not sure if that
* is because of the function, or if the magnetometer just didn't want to
* cooperate. Again, fuck this datasheet, thought the little documentation it
* had on the temperature was more than for the actual magnetometer...
*****/
float Mag_getTemp(void)
{
    bool ack;
    short int xlsb, xmsb, Temp;
    I2Cstart();
    I2Cwait();
    ack = I2Ctx(MagAddr);
    I2Cwait();
    ack = I2Ctx(0xB1);
    I2Cwait();
    I2Crestart();
    I2Cwait();
    ack = I2Ctx(GyroAddr);

```

I2C

```

I2Cwait();
I2Csetrx();
I2Cwait();
Xmsb = I2CrX(0);
I2Cwait();
I2Csetrx();
I2Cwait();
Xlsb = I2CrX(1);
I2Cwait();
I2Cstop();
I2Cwait();

Temp = (Xmsb<<8) | Xlsb;
Temp = Temp>>4;
Temperature = (float)Temp;
Temperature = Temperature/8;

}

////////////////////////////////////
// Barometer functions
// Bar_getParam
// Bar_getTemp
////////////////////////////////////

short int AC1, AC2, AC3, B1, B2, MB, MC, MD;
unsigned short int AC4, AC5, AC6;

/*****
* Function to get all the parameters in order to calculate temperature (and
* later pressure and altitude). Not sure if working right or not. Never got
* accurate temperature reading, and kinda gave up on it to focus on more
* pressing matters. Also, if you thought the Mag datasheet sucked, wait till
* you look at this one. Has absolutely nothing on it. Does tell what to do
* to get temp and pressure values, but does so by assuming you have their code
* and are using their functions (aka saying stuff along the lines of get temp
* by using Bar.workPerfectly&GetTemp() function, or something equally useless
*****/
void Bar_getParam(void){
    bool ack;
    short int h,l;

    //////////////////////////////////////
    //Code to get stuff from Barometer
    //////////////////////////////////////
    I2Cstart();
    I2Cwait();
    ack = I2Ctx(0xEE);
    I2Cwait();
    ack = I2Ctx(0xAA);
    I2Cwait();
    I2Crestart();
    I2Cwait();
    ack = I2Ctx(0xEF);
    I2Cwait();
    //////////////////////////////////////
    //Get lots of data
    //////////////////////////////////////
    I2Csetrx();
    I2Cwait();
    h = I2CrX(0);
    I2Cwait();

```

I2C

```
I2Csetrx();  
I2Cwait();  
l = I2Crx(0);  
I2Cwait();  
AC1 = (h<<8) | l;
```

```
I2Csetrx();  
I2Cwait();  
h = I2Crx(0);  
I2Cwait();  
I2Csetrx();  
I2Cwait();  
l = I2Crx(0);  
I2Cwait();  
AC2 = (h<<8) | l;
```

```
I2Csetrx();  
I2Cwait();  
h = I2Crx(0);  
I2Cwait();  
I2Csetrx();  
I2Cwait();  
l = I2Crx(0);  
I2Cwait();  
AC3 = (h<<8) | l;
```

```
I2Csetrx();  
I2Cwait();  
h = I2Crx(0);  
I2Cwait();  
I2Csetrx();  
I2Cwait();  
l = I2Crx(0);  
I2Cwait();  
AC4 = (h<<8) | l;
```

```
I2Csetrx();  
I2Cwait();  
h = I2Crx(0);  
I2Cwait();  
I2Csetrx();  
I2Cwait();  
l = I2Crx(0);  
I2Cwait();  
AC5 = (h<<8) | l;
```

```
I2Csetrx();  
I2Cwait();  
h = I2Crx(0);  
I2Cwait();  
I2Csetrx();  
I2Cwait();  
l = I2Crx(0);  
I2Cwait();  
AC6 = (h<<8) | l;
```

```
I2Csetrx();  
I2Cwait();  
h = I2Crx(0);  
I2Cwait();  
I2Csetrx();  
I2Cwait();
```

I2C

```

l = I2Crx(0);
I2Cwait();
B1 = (h<<8) | l;

I2Csetrx();
I2Cwait();
h = I2Crx(0);
I2Cwait();
I2Csetrx();
I2Cwait();
l = I2Crx(0);
I2Cwait();
B2 = (h<<8) | l;

I2Csetrx();
I2Cwait();
h = I2Crx(0);
I2Cwait();
I2Csetrx();
I2Cwait();
l = I2Crx(0);
I2Cwait();
MB = (h<<8) | l;

I2Csetrx();
I2Cwait();
h = I2Crx(0);
I2Cwait();
I2Csetrx();
I2Cwait();
l = I2Crx(0);
I2Cwait();
MC = (h<<8) | l;

I2Csetrx();
I2Cwait();
h = I2Crx(0);
I2Cwait();
I2Csetrx();
I2Cwait();
l = I2Crx(1);
I2Cwait();
MD = (h<<8) | l;

////////////////////////////////////
//Finish up protocol
////////////////////////////////////
I2Cstop();
I2Cwait();
}

/*****
* Function to determine temperature from Barometer data. Not sure if it is
* correct or not. Consult the stupid fucking datasheet. It should be noted
* that if this were ever to be actually implemented, it should be split up into
* 2 functions. One to tell the barometer to measure temp. And the other to get
* the data once it was finished.
*****/
int Bar_getTemp(void){
    bool ack;
    short int UT;
    short int MSBdata, LSBdata;

```

I2C

```
long int x1, x2, B5, T;

////////////////////////////////////
//Code to get Barometer to collect data
////////////////////////////////////
I2Cstart();
I2Cwait();
ack = I2Ctx(0xEE);
I2Cwait();
ack = I2Ctx(0xF4);
I2Cwait();
ack = I2Ctx(0x2E);
I2Cwait();
I2Cstop();
I2Cwait();
__delay_us(4500);

////////////////////////////////////
//Code to get Temp data
////////////////////////////////////
I2Cstart();
I2Cwait();
ack = I2Ctx(0xEE);
I2Cwait();
ack = I2Ctx(0xF6);
I2Cwait();
I2Crestart();
I2Cwait();
ack = I2Ctx(0xEF);
I2Cwait();
I2Csetrx();
I2Cwait();
MSBdata = I2Crx(0);
I2Cwait();
I2Csetrx();
I2Cwait();
LSBdata = I2Crx(1);
I2Cwait();
I2Cstop();
I2Cwait();
UT =(MSBdata<<8) | LSBdata;

////////////////////////////////////
// Calculate true temp
////////////////////////////////////
X1 = (UT - AC6)*AC5/pow(2,15);
X2 = MC*pow(2,11)/(X1+MD);
B5 = X1+X2;
T = (B5+8)/pow(2,4);
return (int)T;
}
```

PWM

```

/*
 * File:   PWMfunctions.c
 * Author: Jay Burns
 *
 * Created on April 3, 2012, 9:47 PM
 */

/*
 * README
 *
 * There are four functions you need to call to effectively use the PWM code.
 * First, call initializePWMcapture() and initializePWMoutput(). This will
 * setup the control registers and timers so input and output is ready to go. Then,
 * in your while loop, call getPWM(); This will read the PWM values. They are stored in:
 *
 * CH1value
 * CH2value
 * CH3value
 * CH4value
 * These are all values of 0 to 100.
 *
 * CH5status
 * CH5status is either 1 or 0. I suggest your main looking something like this:
 *
 * int main(void)
 {
     initializePWMcapture();
     initializePWMoutput();

     while(1)
     {
         getPWM();
         if (CH5status == 1)
         {
             setPower(75, 74, 76, 72); //This is just an example output to the
motors. Remember, it's //setPower(motor1, motor2, motor3, motor4)
         }
         else if (CH5status == 0)
         {
             setPower(0, 0, 0, 0);
         }
     }
 }
 */

//      Function Prototypes
void getPWM(void);
void initializePWMcapture(void);
void initializePWMoutput(void);
void setPower(int, int, int, int);
float readCH1(int);
float readCH2(int);
float readCH3(int);
float readCH4(int);
void readCH5(void);

//Declare Global Variables

int IC1Capture1, IC1Capture2, IC2Capture1, IC2Capture2, IC3Capture1, IC3Capture2,
IC4Capture1, IC4Capture2, IC5Capture1, IC5Capture2;
int CH1count, CH2count, CH3count, CH4count, CH5count, CH1value, CH2value, CH3value,
CH4value, CH5value, CH5status;

```


PWM

```

void initializePWMoutput(void)
{
    int pulseperiod = 4629;
    int pulsemin = 280;

    // Configure Output Functions (Table 10-4)
    RPOR5bits.RP11R = 18; // Assign OC1 to Pin RP11 (RD0)
    RPOR12bits.RP24R = 19; // Assign OC2 to Pin RP24 (RD1)
    RPOR11bits.RP23R = 20; // Assign OC3 to Pin RP23 (RD2)
    RPOR11bits.RP22R = 21; // Assign OC4 to Pin RP22 (RD3)

    //Set up OC1
    OC1CON1 = 0x0000; //Clear control registers
    OC1CON2 = 0x0000; //Clear control registers
    OC1RS = pulsemin; //Load secondary register with initial pulse
value
    OC1R = pulsemin; //Load primary register with initial pulse
value
    OC1CON1bits.OCTSEL = 0b000; //Select Timer 2 as source
    OC1CON2bits.SYNCSEL = 0b01100; //Select Timer 2 as sync
    OC1CON1bits.OCM = 0b110; //Select Edge aligned PWM
    OC1CON2bits.OCTRIG = 0;

    //Set up OC2
    OC2CON1 = 0x0000; //Clear control registers
    OC2CON2 = 0x0000; //Clear control registers
    OC2RS = pulsemin; //Load secondary register with initial pulse
value
    OC2R = pulsemin; //Load primary register with initial pulse
value
    OC2CON1bits.OCTSEL = 0b000; //Select Timer 2 as source
    OC2CON2bits.SYNCSEL = 0b01100; //Select Timer 2 as sync
    OC2CON1bits.OCM = 0b110; //Select Edge aligned PWM
    OC2CON2bits.OCTRIG = 0;

    //Set up OC3
    OC3CON1 = 0x0000; //Clear control registers
    OC3CON2 = 0x0000; //Clear control registers
    OC3RS = pulsemin; //Load secondary register with initial pulse
value
    OC3R = pulsemin; //Load primary register with initial pulse
value
    OC3CON1bits.OCTSEL = 0b000; //Select Timer 2 as source
    OC3CON2bits.SYNCSEL = 0b01100; //Select Timer 2 as sync
    OC3CON1bits.OCM = 0b110; //Select Edge aligned PWM
    OC3CON2bits.OCTRIG = 0;

    //Set up OC4
    OC4CON1 = 0x0000; //Clear control registers
    OC4CON2 = 0x0000; //Clear control registers
    OC4RS = pulsemin; //Load secondary register with initial pulse
value
    OC4R = pulsemin; //Load primary register with initial pulse
value
    OC4CON1bits.OCTSEL = 0b000; //Select Timer 2 as source
    OC4CON2bits.SYNCSEL = 0b01100; //Select Timer 2 as sync
    OC4CON1bits.OCM = 0b110; //Select Edge aligned PWM
    OC4CON2bits.OCTRIG = 0;

    //Set up Timer2, our timer for output PWM

```

```

    TMR2          = 0;          //Set Timer2 to 0
    PR2          = pulseperiod; //Load number of increments in period
    T2CONbits.TCKPS = 0b10;    //Prescale value is 64
    IFS0bits.T2IF  = 0;        //Clear Timer2 interrupt flag
    IEC0bits.T2IE  = 1;        //Enable Timer2 interrupts
    T2CONbits.TON  = 1;        //Turn Timer2 on
}

```

```
void initializePWMcapture(void)
```

```

{
    // Configure Output Functions (Table 10-4)

    RPINR7bits.IC1R = 2;      //Assign Input Capture 1 to RP2 (RD8)
    RPINR7bits.IC2R = 4;      //Assign Input Capture 2 to RP4 (RD9)
    RPINR8bits.IC3R = 3;      //Assign Input Capture 3 to RP3 (RD10)
    RPINR8bits.IC4R = 12;     //Assign Input Capture 4 to RP12 (RD11)
    RPINR9bits.IC5R = 42;     //Assign Input Capture 5 to RPI42 (RD12)

    //Initialize IC1
    TRISDbits.TRISD8 = 1;     //Set RD8 to input
    IC1CON1bits.ICM = 0b000;  //This should reset the overflow condition flag,
reset the FIFO to empty, and reset the prescale count
    IC1CON1bits.ICTSEL = 0b010; //Select Timer4 as timer
    IC1CON1bits.ICM = 0b01;   //Capture timer value on every edge
    IC1CON1bits.ICI = 0b00;   //Interrupt on every capture event
    IC1CON2bits.SYNCSEL = 0b01110; //Sync with Timer4
    IC1CON2bits.ICTRIG = 0;   //Clear trigger bit for synchronous mode.
    IEC0bits.IC1IE = 1;      //Enable IC1 interrupt
    IFS0bits.IC1IF = 0;      //Set interrupt flag status to 0

    //Initialize IC2
    TRISDbits.TRISD9 = 1;     //Set RD9 to input
    IC2CON1bits.ICM = 0b000;  //This should reset the overflow condition flag,
reset the FIFO to empty, and reset the prescale count
    IC2CON1bits.ICTSEL = 0b010; //Select Timer4 as timer
    IC2CON1bits.ICM = 0b01;   //Capture timer value on every edge
    IC2CON1bits.ICI = 0b00;   //Interrupt on every capture event
    IC2CON2bits.SYNCSEL = 0b01110; //Sync with Timer4
    IC2CON2bits.ICTRIG = 0;   //Clear trigger bit for synchronous mode.
    IEC0bits.IC2IE = 1;      //Enable IC2 interrupt
    IFS0bits.IC2IF = 0;      //Set interrupt flag status to 0

    //Initialize IC3
    TRISDbits.TRISD10 = 1;    //Set RD10 to input
    IC3CON1bits.ICM = 0b000;  //This should reset the overflow condition flag,
reset the FIFO to empty, and reset the prescale count
    IC3CON1bits.ICTSEL = 0b010; //Select Timer4 as timer
    IC3CON1bits.ICM = 0b01;   //Capture timer value on every edge
    IC3CON1bits.ICI = 0b00;   //Interrupt on every capture event
    IC3CON2bits.SYNCSEL = 0b01110; //Sync with Timer4
    IC3CON2bits.ICTRIG = 0;   //Clear trigger bit for synchronous mode.
    IEC2bits.IC3IE = 1;      //Enable IC3 interrupt
    IFS2bits.IC3IF = 0;      //Set interrupt flag status to 0

    //Initialize IC4
    TRISDbits.TRISD11 = 1;    //Set RD11 to input
    IC4CON1bits.ICM = 0b000;  //This should reset the overflow condition flag,
reset the FIFO to empty, and reset the prescale count
    IC4CON1bits.ICTSEL = 0b010; //Select Timer4 as timer
    IC4CON1bits.ICM = 0b01;   //Capture timer value on every edge
    IC4CON1bits.ICI = 0b00;   //Interrupt on every capture event
    IC4CON2bits.SYNCSEL = 0b01110; //Sync with Timer4
    IC4CON2bits.ICTRIG = 0;   //Clear trigger bit for synchronous mode.
}

```

```

                                PWM
IEC2bits.IC4IE      = 1;      //Enable IC5 interrupt
IFS2bits.IC4IF      = 0;      //Set interrupt flag status to 0

//Initialize IC5
TRISDbits.TRISD12   = 1;      //Set RD12 to input
IC5CON1bits.ICM     = 0b000;   //This should reset the overflow condition flag,
reset the FIFO to empty, and reset the prescaler count
IC5CON1bits.ICTSEL  = 0b010;   //Select Timer4 as timer
IC5CON1bits.ICM     = 0b01;    //Capture timer value on every edge
IC5CON1bits.ICI     = 0b00;    //Interrupt on every capture event
IC5CON2bits.SYNCSEL = 0b01110; //Sync with Timer4
IC5CON2bits.ICTRIG  = 0;      //Clear trigger bit for synchronous mode.
IEC2bits.IC5IE      = 1;      //Enable IC5 interrupt
IFS2bits.IC5IF      = 0;      //Set interrupt flag status to 0

//Set up Timer4, our clock for PWM
//IFS1bits.T4IF      = 0;      // Commented out because I couldn't think of a
reason to need the interrupt
//IEC1bits.T4IE      = 1;      // Enable Timer4 Compare interrupts
T4CONbits.TON       = 1;      // Start Timer2 with assumed settings
T4CONbits.TCKPS     = 0b10;   // Timer4 Prescale value = 64
TMR4                = 0;      // Set value of Timer4 to 0
}

```

```

void __attribute__((interrupt, shadow, no_auto_psv)) _IC1Interrupt()
{
    IFS0bits.IC1IF = 0; //Reset respective interrupt flag

    if (PORTDbits.RD8 == 1) //If the input is high, start count
    {
        IC1Capture1 = IC1BUF; //Load the start value from the capture buffer
    }

    else if (PORTDbits.RD8 == 0) //If the input is low, end count
    {
        IC1Capture2 = IC1BUF; //Load the end value from the capture buffer
    }

    CH1count = (IC1Capture2 - IC1Capture1); //Running time is ent time minus start
time

    if (CH1count < 0) //Verify integrity of count
    {CH1count = CH1count + 4619;}
}

```

```

void __attribute__((interrupt, shadow, no_auto_psv)) _IC2Interrupt()
{
    IFS0bits.IC2IF = 0; //Reset respective interrupt flag

    if (PORTDbits.RD9 == 1)
    {
        IC2Capture1 = IC2BUF;
    }

    else if (PORTDbits.RD9 == 0)
    {
        IC2Capture2 = IC2BUF;
    }

    CH2count = (IC2Capture2 - IC2Capture1);

    if (CH2count < 0) //Verify integrity of count
    CH2count = CH2count;
}

```

PWM

```

}

void __attribute__((interrupt, shadow, no_auto_psv)) _IC3Interrupt()
{
    IFS2bits.IC3IF = 0; //Reset respective interrupt flag

    if (PORTDbits.RD10 == 1)
    {
        IC3Capture1 = IC3BUF;
    }

    else if (PORTDbits.RD10 == 0)
    {
        IC3Capture2 = IC3BUF;
    }

    CH3count = (IC3Capture2 - IC3Capture1);

    if (CH3count < 0) //Verify integrity of count
        CH3count = CH3count + 4619;
}

void __attribute__((interrupt, shadow, no_auto_psv)) _IC4Interrupt()
{
    IFS2bits.IC4IF = 0; //Reset respective interrupt flag

    if (PORTDbits.RD11 == 1)
    {
        IC4Capture1 = IC4BUF;
    }

    else if (PORTDbits.RD11 == 0)
    {
        IC4Capture2 = IC4BUF;
    }

    CH4count = (IC4Capture2 - IC4Capture1);

    if (CH4count < 0) //Verify integrity of count
        CH4count = CH4count + 4619;
}

void __attribute__((interrupt, shadow, no_auto_psv)) _IC5Interrupt()
{
    IFS2bits.IC5IF = 0; //Reset respective interrupt flag

    if (PORTDbits.RD12 == 1)
    {
        IC5Capture1 = IC5BUF;
    }

    else if (PORTDbits.RD12 == 0)
    {
        IC5Capture2 = IC5BUF;
    }

    CH5count = (IC5Capture2 - IC5Capture1);

    if (CH5count < 0) //Verify integrity of count
        CH5count = CH5count + 4619;
}

void __attribute__((interrupt, shadow, no_auto_psv)) _T2Interrupt()

```

PWM

```

{
  IFS0bits.T2IF = 0;
}

void __attribute__((interrupt, shadow, no_auto_psv)) _T4Interrupt()
{
  IFS1bits.T4IF = 0;
}

void setPower(int one, int two, int three, int four) //Function takes motor thrust
commands from 0 to 100% and outputs OCxR register value
{
  OC1R = (one) + 280;      //Scale up the percentage to register value
  OC2R = (two) + 280;
  OC3R = (three) + 280;
  OC4R = (four) + 280;
}

float readCH1(int count)          //Takes captured value and translates it to
percentage 0 to 100
{
  float value = 0;              //Placeholder variable
  if (count > 500 || count < 250) //Check integrity of value, if out of 6-10.5%
range, just write it as low value, 0%
  {
    value = 0;
  }
  else                          //If it's good, compute its percentage range
  {
    value = 0.510126*count - 145.38591;
  }
  return value;                 //Return that the percentage
}

float readCH2(int count)          //Takes captured value and translates it to
percentage 0 to 100
{
  float value = 0;              //Placeholder variable
  if (count > 500 || count < 250) //Check integrity of value, if out of 6-10.5%
range, just write it as low value, 0%
  {
    value = 0;
  }
  else                          //If it's good, compute its percentage range
  {
    value = 0.50505051*count - 139.39394;
  }
  return value;                 //Return that the percentage
}

float readCH3(int count)          //Takes captured value and translates it to
percentage 0 to 100
{
  float value = 0;              //Placeholder variable
  if (count > 500 || count < 250) //Check integrity of value, if out of 6-10.5%
range, just write it as low value, 0%
  {
    value = 0;
  }
  else                          //If it's good, compute its percentage range
  {
    value = 0.50761421*count - 141.62437;
  }
}

```

```

    return value;                PWM //Return that the percentage
}

float readCH4(int count)        //Takes captured value and translates it to
percentage 0 to 100
{
    float value = 0;            //Placeholder variable
    if (count > 500 || count < 250) //Check intergrity of value, if out of 6-10.5%
range, just write it as low value, 0%
    {
        value = 0;
    }
    else                        //If it's good, compute its percentage range
    {
        value = 0.51282051*count - 144.61538;
    }
    return value;              //Return that the percentage
}

void readCH5(void)
{
    if (CH5count > 500 && CH5count < 530)
    {
        CH5status = 1;
    }
    else if (CH5count < 400 && CH5count > 230)
    {
        CH5status = 0;
    }
}

void getPWM(void)
{
    CH1value = readCH1(CH1count); //Read in the input channels
    CH2value = readCH1(CH2count);
    CH3value = readCH1(CH3count);
    CH4value = readCH1(CH4count);
    readCH5();
}

```

TerminalUart

```
/*This header file contains the functions
   necessary to output a string, char, and
   an integer to a terminal*/

#include <stdlib.h>
#include <p24FJ256GB110.h>
#include <libpic30.h>
#include <stdbool.h>

// Define output port #s
#define U1Tx 3;
#define U1RTS 4;

// Initialize USART1 to USB input
void init_usart(unsigned short rate)
{
    RPINR18bits.U1RXR = 38;    //Assign U1Rx to pin RPI38
    RPINR18bits.U1CTS = 40;    //Assign U1CTS to pin RPI40
    RPOR7bits.RP15R = U1Tx;    //Assign U1Tx to pin RP15
    RPOR15bits.RP30R = U1RTS;  //Assign U1RTS to pin rp21
    U1MODEbits.UARTEN = 1;     //Enable Uart1
    U1MODEbits.UEN = 0b10;     //Tx, Rx, CTS, and RTS enabled & used
    U1MODEbits.PDSEL = 0b00;   //8-bits no parity
    U1STAbits.UTXEN = 1;      //Enable transmit

    U1BRG = rate;              //set the BRG register to rate
}

/*****
 * Function to put a char to the terminal
 *****/
void myputc(char value)
{
    // volatile bit TSReg = U1STAbits.TRMT;
    while(!U1STAbits.TRMT);
    U1TXREG = value;
    return;
}

/*****
 * Function to put a string to the terminal
 *****/
void myputs(char *strn)
{
    int i = 0;
    while(strn[i] != NULL)
    {
        myputc(strn[i]);
        i++;
    }
}

/*****
 * Function to put an integer to the terminal
 *****/
void myputi(int value)
{
    char cValue[10];
    itoa(cValue, value, 10);
    myputs(cValue);
}
}
```

TerminalUart

Misc

```
/* *****  
* This header file is primarily a place where all the miscellaneous functions  
* I wrote ended up, hence the name.  
* ***** */
```

```
/* *****  
* Function to print the offsets to the screen. Got tired of this giant block  
* of text in the main, so put it in a function.  
* ***** */
```

```
void PrintOffs(void)  
{  
    char string[20];  
    myputc('\n');  
    myputc('\r');  
    myputs("Start:");  
    myputc('\n');  
    myputc('\r');  
    sprintf(string, "Zoff: %f", AccZoff);  
    myputs(string);  
    myputc('\n');  
    myputc('\r');  
    myputs("Rotational Offsets: ");  
    myputi(RotOff[0]);  
    myputc(',');  
    myputi(RotOff[1]);  
    myputc(',');  
    myputi(RotOff[2]);  
    myputc('.');  
    myputc('\n');  
    myputc('\r');  
}
```

```
/* *****  
* Function to print out most, if not all, of the data to the screen. Again  
* wrote the function because I was tired of the big block sitting in the main.  
* ***** */
```

```
void PrintData(void)  
{  
    char string[20];  
    myputc('<');  
    sprintf(string, "%f", AccVec[0]);  
    myputs(string);  
    myputc(',');  
    sprintf(string, "%f", AccVec[1]);  
    myputs(string);  
    myputc(',');  
    sprintf(string, "%f", AccVec[2]);  
    myputs(string);  
    myputc('>');  
    myputc('\t');  
    myputc('<');  
    sprintf(string, "%f", MagVec[0]);  
    myputs(string);  
    myputc(',');  
    sprintf(string, "%f", MagVec[1]);  
    myputs(string);  
    myputc(',');  
    sprintf(string, "%f", MagVec[2]);  
    myputs(string);  
    myputc('>');  
    myputc('\t');  
    myputc('<');  
}
```

Misc

```

    myputi(RotVec[0]);
    myputc(',');
    myputi(RotVec[1]);
    myputc(',');
    myputi(RotVec[2]);
    myputc('>');
    /*
    myputc('\t');
    myputc('<');
    sprintf(string, "%f",OrientVec[0]);
    myputs(string);
    myputc(',');
    sprintf(string, "%f",OrientVec[1]);
    myputs(string);
    myputc(',');
    sprintf(string, "%f",OrientVec[2]);
    myputs(string);
    myputc('>');
    */
    myputc('\n');
    myputc('\r');
}

/*****
 * Function I put together to round a number. Pretty sure it works. Can't
 * believe the math.h header didn't have this function in it.
 *****/
float round(float num)
{
    int numi;
    numi = num*10;
    numi = numi%10;

    if (numi >= 5)
    {
        if (num >=0)
        {
            return (float)ceil(num);
        }
        else
        {
            return (float)floor(num);
        }
    }
    else
    {
        if (num >= 0)
        {
            return (float)floor(num);
        }
        else
        {
            return (float)ceil(num);
        }
    }
}

/*****
 * Function written to initialize timer 1. This is used to calculate time it
 * takes for the loop to run to calculate integral and derivative portions of
 * the PID controller
 *****/
void init_TMR1(void)

```

```

{
    TMR1          = 0;          //Set Timer2 to 0
    T1CONbits.TCKPS = 0b10;    //Prescale value is 8
    T1CONbits.TON  = 1;        //Turn Timer2 off
}
Misc
```