GPS Guided Autonomous Drone

Cameron Roberts, Electrical Engineering

Project Advisor: Dr. Tony Richardson

April 25, 2016

University of Evansville

College of Engineering and Computer Science

Evansville, Indiana 47714

Table of Contents

List of Figures

List of Tables

I. Introduction

The project is to design an autonomous flying drone, specifically a quadcopter. The drone is fitted with a GPS tracking system and programmed to be able to autonomously fly from one location to another using GPS coordinates. Significant consideration is given to safety and ruggedness due to the possibility of collision with a variety of objects. In addition to collisions, the drone is also rugged enough to operate during moderately windy conditions. The goal of the project is to act as a proof of concept for small scale autonomous aerial delivery similar to that nearing deployment by Amazon.

II. Problem Definition

Creating a GPS guided drone holds numerous challenges, chief of which is flight. In order to fly the drone uses four motors at the end of long arms that stick out from a central hub. The hub is where the flight controller and battery are located so that their weight doesn't throw of the drone's delicate balance. Furthermore the motors cannot adjust the pitch of the blades attached to them leaving their rotational speed the main control mechanism. Orientation information for a flying drone is usually discussed using just three key elements pitch, roll and yaw as is shown in figures 1 and 2. This methodology of control wouldn't work however if it weren't for the fact that there are two motors spinning counter clockwise and two spinning clockwise as can be seen in figure 3. A GPS receiver on the central hub tells the drone where it currently is by communicating with several GPS satellites. This information is used in conjunction with a user selected destination to get a path of travel. To simplify the problem many drones fly at an altitude free of obstacles, as the quadcopter for this project is meant to, so sensors for detecting nearby objects are unnecessary and thus are not included.

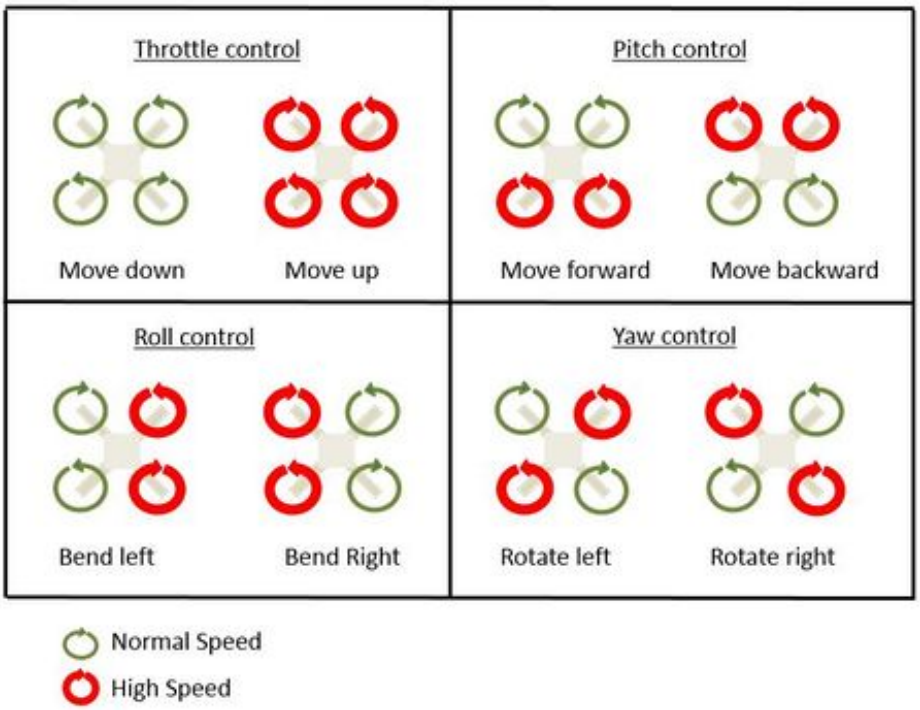
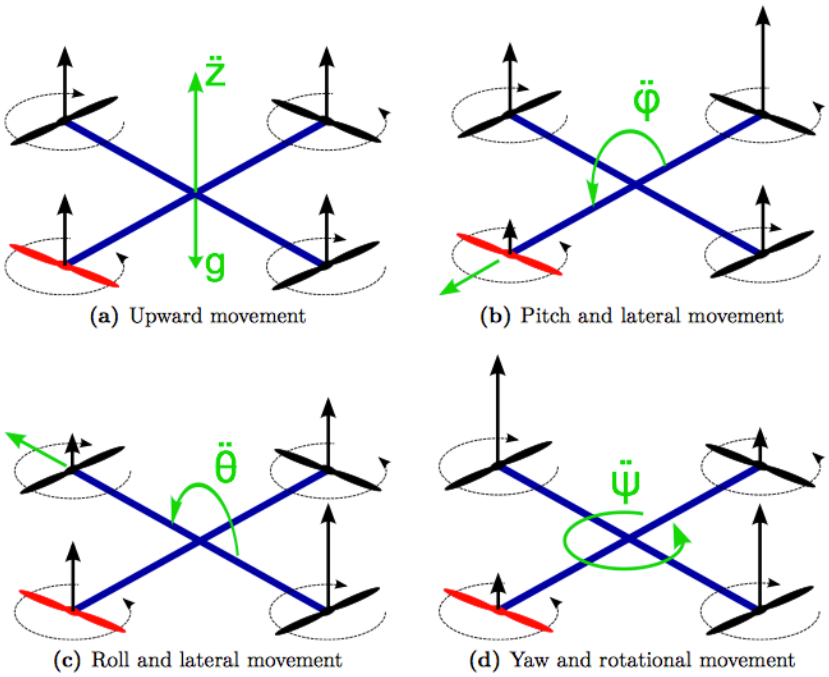


Figure 1: Quadcopter control related to individual motor speeds [1]



(a) Upward movement (b) Pitch and lateral movement

(c) Roll and lateral movement (d) Yaw and rotational movement

Figure 2: Quadcopter orientation terminology diagram [2]

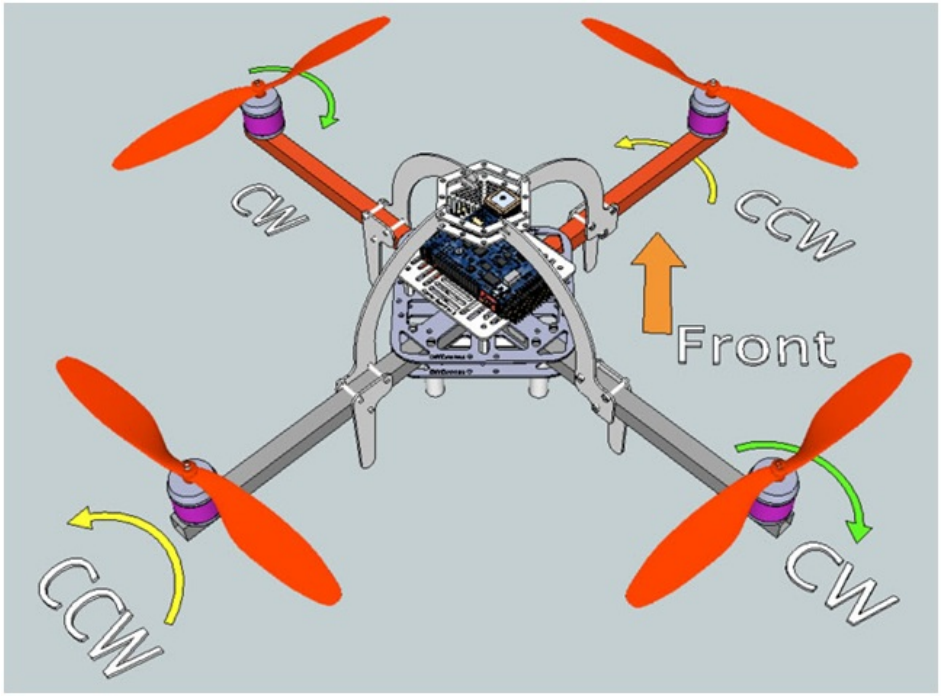


Figure 3: 3-D quadcopter model with motor rotation directions [3]

Client Requirements:

- Design and construct a multicopter
- Program the multicopter to be able to fly without outside assistance
- Use GPS coordinates for the drone's navigation method

III. Project Design

The design of the drone is broken down into two large subcategories of hardware and software. The hardware involved assembling the drone and testing it to verify that its components were working properly once connected together. The software portion of the project is to create the code necessary for the quadcopter to control itself during flight using the Raspberry Pi B+ [4].

The hardware subsection consisted of assembling the individual components that were necessary for flight and navigation as well as using the Navio+ [5] for testing, which will be explained in greater detail later. No kit or predetermined list is employed, instead each part of the quadcopter is selected based on in-depth research and the advice of more experienced multicopter pilots. Lift is achieved using four Cobra C-2213/22 brushless DC motors [6] attached to a rigid Flip Sport carbon composite frame [7]. The motors are driven by an electronic speed controller (ESC) because the current and voltage necessary to operate them cannot be supplied by the Navio+ or Raspberry PI B+. To change the speed of the motors the duty ratio of the control signal is varied changing the voltage the motors see. For this particular set up duty ratio varies between .04 (1 ms pulses) and .07 (1.75 ms pulses). Unfortunately, the large amount of power used by the four motors means that a lithium ion battery (LiPo) can't offer any more than about ten minutes of flight time.

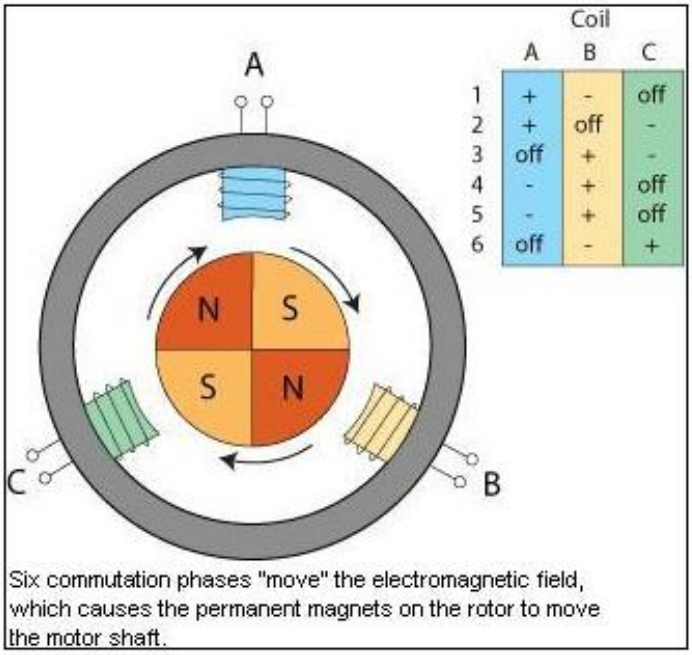


Figure 4: Cross section and operation of simple four pole brushless DC motor [8]

Figure 5: Photograph of the Flip Sport frame being used for the project [7]

Putting together each of the drone's core components took several steps and careful planning to make sure everything could fit correctly. All large and heavy parts were kept as close to the middle of the central hub as possible to reduce their effects on stability. With a solid framework for positioning assembly along the guidelines of figure 6 took place early in the semester to make way for the testing and software phases of the project.

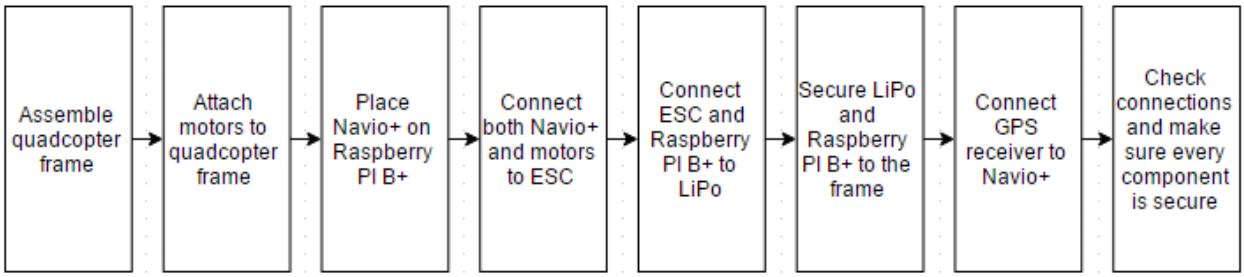


Figure 6: Block diagram of drone hardware assembly

The below schematic shows the various connections between the motors, flight controller (Raspberry PI B+/Navio+) and ESC. A voltage regulator is necessary between the LiPo battery and flight controller because the Raspberry PI B+ could be severely damaged or destroyed by voltages any higher than 5.5 volts. Alternatively the 11.1 volts supplied by the three cell LiPo

battery is sent directly to the four motors. It is impossible to see here, but by switching any of the three wires going to each motor (signal, ground and power) the rotation could be reversed. This made it simple to pair motors 1 and 3 to be clockwise as well as 2 and 4 to be counterclockwise.

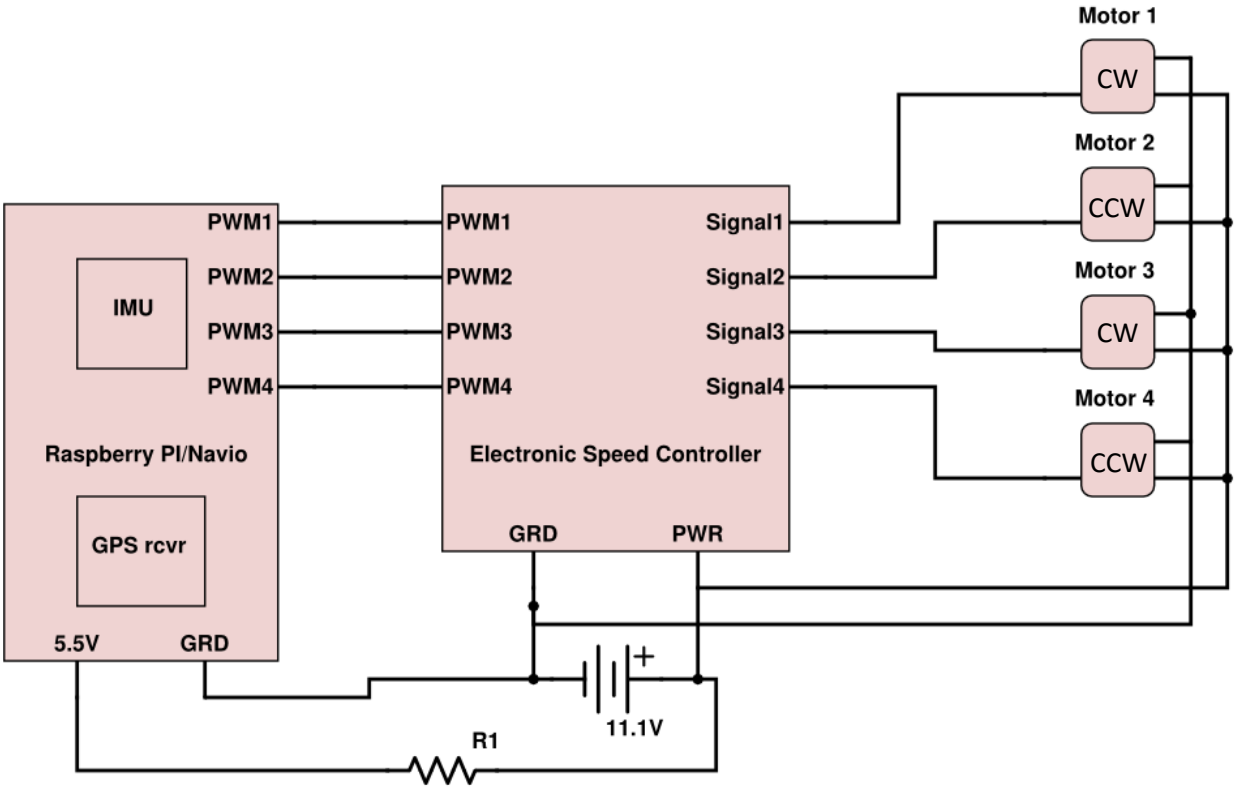


Figure 7: Drone electrical schematic

Hardware also went beyond construction to include testing of the drone's sensor equipment. The goal here was to determine the sensor data corresponding to a given orientation. Testing is possible thanks to the Navio+'s available software, allowing for receiving and manipulating of GPS and IMU sensor data prior to my own code being completed. Magnetometer values were used to determine North, South, East, West orientation whereas the accelerometers were used to test for being level. This was done by comparing the magnetometer values in real

time to a compass as the quadcopter was turned in a circle. The magnetometer values in all three axes were recorded for each of the cardinal directions. To simplify movement the drone will only attempt to move in a single cardinal direction at a time so knowing combinations of them for this testing is unnecessary.

Accelerometer values were tested in a similar way but instead of spinning the drone it was instead tilted in the eight main ways it could be while in flight. Those tilting directions are forward, right, forward-right, left, forward-left, back, back-right and back-left. For each of these different scenarios the three axes accelerometers change in value in a specific way that can be easily tested for. This method of testing for a tilt using the accelerometers is how the drone determines when to call the PID controller discussed below.

Programming of the drone's software is fundamentally crucial to allow for autonomous flight. The main challenge here is to use the Navio+ to control the motors individually through programming the Raspberry PI B+ employing a closed loop controller. A PID controller was chosen because of its three levels of control each of which has a particular situation it is best at. PID is an acronym for these three control methods the P standing for proportional, I for integral and D for derivative.

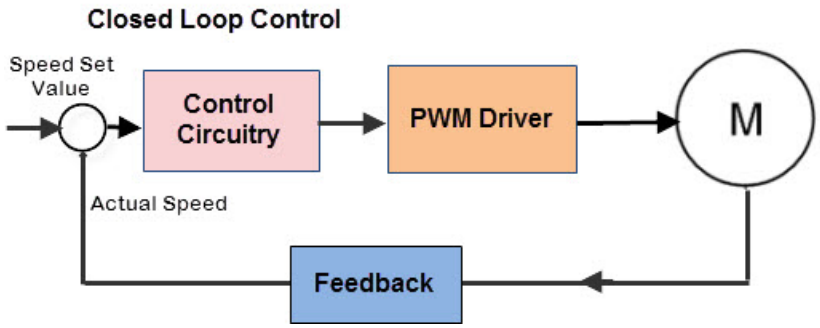


Figure 7: PID controller block diagram

The Navio+ contained the IMU that supplied the necessary rotational and orientation data to the Raspberry PI B+ for the PID controller. The P term is what has the fastest and most drastic impact on the output to the motors. The I term is for more fine control of the drones stability and the D term is only implemented to fix sudden drastic changes in stability caused by things like a gust of wind. This meant the Raspberry PI B+ needed to be programed to receive information from the Navio+, process it, and send back the necessary signals to maintain stability. In terms of the above diagram, the Raspberry PI B+ served as the control circuitry and the Navio+ acted as the PWM driver and feedback, with the motors being represented by the "M". Testing of the software was simulated by creating artificial input numbers that caused the PID controller to react as if it were in flight. Setting of the desired destination coordinate is done using a terminal on a personal computer. This is possible thanks to an SSH (secure shell) connection to the Raspberry PI B+ over Wi-Fi to my windows laptop. The full code that is used to control the quadcopter drone is contained in appendix A of the appendices section.

Flying height constraints for the project are set by the FAA, but all other constraints are set by the team members and project advisor based on social considerations and the IEEE code of ethics [9]. Current FAA guidelines allow small autonomous drones, such as the one for this project, to be operated at under 1200 feet [10]. Pending legislation could restrict operation to daylight hours away from helipads. Laws passed in December require registration for any autonomous drone weighing more than half a pound, therefore the drone was registered with the US government. It is at this point that the IEEE code of ethics comes into play and places constraints on the operation and construction of the drone. The first rule of the code, considering the wellbeing of the public, is the most applicable to the project and was used to develop a list of reasonable rules to abide by [9]. The first rule is that due to the noise created by the drone while in operation, it won't be used

near any residential structures. The second guideline is that the drone won't be used near people, animals, or damageable property until tests confirm safe operation. The final guideline is dispose of any damaged or broken parts properly to keep the drone's environmental impact to a minimum.

The only given specification for completion of the project is a drone that is capable of moving from one GPS coordinate to another coordinate on its own. Therefore, complete success of the project would have been demonstrated by giving the drone a coordinate roughly 50 to 100 feet way and allowing it to fly there without help.

Figures 9 and 10 below show a top and side view of the fully constructed quadcopter respectively. The black circuit board in the middle of the quadcopter is the Navio+ and upon careful inspection a green board is visible below it, this is the Raspberry PI B+. The four motors with attached nine inch propellers are clearly visible at the end of the frames four arms. The red wire sticking out from the hub in both figures is the power between the battery and ESC, it is disconnected so the motors don't accidentally turn on and cause harm to anyone or anything. The ESC and battery aren't visible in either figure because they are below the Raspberry PI/Navio+ on a second and third level of the frame. This was done because there was insufficient lateral room to fit in everything without severely shifting the center of mass from the middle of the central hub.

Figure 8: Top view of final drone

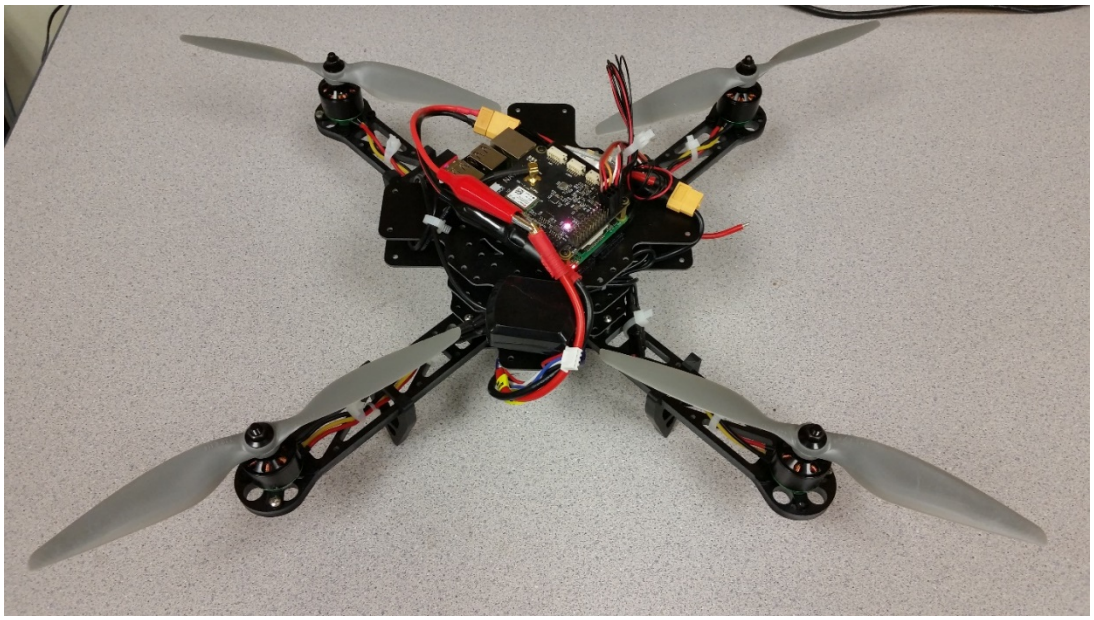


Figure 9: Side view of fully constructed drone

IV. Costs

The only monetary costs for the entire project were the initial purchasing of the components to construct the drone. The price of the parts necessitated the acquisition of funds from the College of Engineering and Computer Science. A proposal was given to Dr. Gerhart, the Dean of the College of Engineering, and the project received $500 of funding as a result. Beyond that, only time on the part of the team was necessary to complete the software section of the project.

Components	**Cost**
Flip Sport (Black)	$65
Flip four tab center plate kit (Black)	$20
Mounting hardware (8 spacers with screws)	$4
Cobra C-2213/22 1100Kv Brushless Motor x4	$116
APC 9x4.5 Reverse Rotation x2	$3.30
APC 9x4.5 Normal Rotation x2	$3.30
Turnigy nano-tech 4000mah 3S 25~50C Lipo Pack	$27
Navio+ Autopilot Kit for Raspberry Pi A+/B+	$213
Turnigy Accucel-6 50W 6A Balancer/Charger w/ Accessories	$23
4-IN-1 Speed Control Quad Hobbywing Skywalker Quattro 25A X 4	$40
Total + shipping	$550

Table 1: Cost layout for components

V. Results

A quadcopter that is capable of being given a GPS coordinate using a PC and attempts to fly to that coordinate by running through the constructed program. Due to the final weight being incalculable before construction and limited funding the quadcopter was unfortunately too heavy to lift off the ground. With larger DC motors this would not have been an issue, but brushless DC motors are very costly. Contrarily the software section was completed to as great an extent as possible given the lack of ability for real world testing.

Execution of the drone's software begins with the user being prompted to enter a set of destination latitude and longitude values in decimal form, negative numbers are used for southern and western coordinates. These values are then converted into feet and compared to the GPS receiver's values also converted to feet. This difference determines the destinations direction depending on if the latitude difference is positive or negative and if the longitude difference is positive or negative. Once direction is known ground level is set as the current altitude for landing purposes. The drone would then lift off and maneuver to its destination, maintaining stability along the way using the PID controller. Once at its destination, with a margin of error of two feet in either direction, it would slowly spin down its motors until it returned to its preset ground level.

```
pi@navio-rpi ~/navio/C++/Examples/AccelGyroMag $ sudo ./AccelGyroMag
To begin please enter in a set of GPS coordinates with the form XX.XXXXXX
Placing a negative sign in front of latitude value causes the computer to treat it as a West coordinate
Placing a negative sign in front of the longitude value causes the computer to treat it as a South coordinate
Enter a valid Latitude value between -90 and 90: 37.971283
Enter a valid Longitude value between -180 and 180: -87.530717
You have entered a GPS coordinate of Latitude: 37.971283
With a Longitude of: -87.530716
Latitude in feet: 13826271
Longitude in feet: -31872068
Acc:  -0.035  +0.048  +0.988  Gyr:   -0.244   +0.366   -0.244  Mag: -18.871 +25.388  +0.000
Temperature(C): 34.173409 Pressure(millibar): 991.623352
Ublox test OK
Message not captured
Acc:  -0.031  +0.045  +0.986  Gyr:   -0.305   +0.305   +0.000  Mag: -20.811 +72.436 +24.212
Temperature(C): 34.190201 Pressure(millibar): 991.653259
Ublox test OK
Longitude: 0.000000
Latitude: 0.000000
Height above mean sea level: -17.000 m
```

Figure 10: Beginning of program execution

VI. Conclusion

An autonomous drone such as the one constructed in this project could have many uses in practical applications. Among other things a GPS guided drone could deliver packages to residences and businesses drastically lowering delivery times and costs. Other uses could be gathering weather and atmospheric data or taking pictures of large areas at once. At a cost of roughly $550 the constructed drone was also substantially cheaper than any commercial drone of

a similar size. With the project complete, with larger motors all that is necessary to achieve flight, replication is now relatively simply assuming identical components the code would require no alteration. Hardware assembly and testing is all that would be necessary to create as many autonomous drones as desired.

VII. References

[1] Harsha, Sree. (2014, August). Quadcopter. San Jose State University. San Jose, California. [Online]. Available: http://www.socialledge.com/sjsu/index.php?title=S14:_Quadcopter#Acknowledgement

[2] Saleh Abo Elmakarem. (2014, June). Quadcopter Mechanics. UAV Society. Egypt. [Online]. Available: http://uav-society.blogspot.com/2014/06/quadcopter-mechanics.html

[3] Jespersen, Thomas. (2012, March). Quadcopters – How to get started. TKJ Electronics. Denmark. [Online]. Available: http://blog.tkjelectronics.dk/2012/03/quadcopters-how-to-get-started/

[4] Raspberry PI B+ is a single board computer fully equipped with I/O ports a CPU/GPU as well as onboard RAM and flash memory https://www.raspberrypi.org/products/model-b-plus/

[5] Navio+ is an autopilot shield for the Raspberry PI B+ that has onboard sensors including gyroscopes/ accelerometers and a barometer http://www.emlid.com/shop/navio-plus/

[6] Cobra C-2213/22 is a small brushless DC motor with 14 magnet poles and 12 stator slots that spins at a ratio of 1100 rmp/volt http://www.cobramotorsusa.com/motors-2213-22.html

[7] Hoverthings Inc. (2015). Flip Sport (Black). Hoverthings Inc. Saint Petersburg, Florida. [Online]. Available: http://www.hoverthings.com/the-flip-black

[8] Agarwal, Tarun. (August, 2014). How to Control a Brushless DC Motor in Electrical Field. Edgefx. Hyderabad, India. [Online]. Available: http://www.efxkits.co.uk/speed-control-of-brushless-dc-motor/

[9] IEEE Code of Ethics (2015). 7.8 IEEE Code of Ethics. Institute of Electrical and Electronics Engineers. Piscataway, New Jersey. [Online]. Available: http://www.ieee.org/about/corporate/governance/p7-8.html

[10] Laksman, Martin. (September, 2013). This is How the FAA Regulates American Airspace. Popular Mechanics. New York City, New York. [Online] Available: http://www.popularmechanics.com/military/a9397/this-is-how-the-faa-regulates-american-airspace-15894142/

VIII. Appendices

Appendix A Main Code

```
// USB front
#define NAVIO_RCOUTPUT_1 12 // Brown (Left)
#define NAVIO_RCOUTPUT_2 13 // Red    (Back)
#define NAVIO_RCOUTPUT_3 14 // Orange (Right)
#define NAVIO_RCOUTPUT_4 15 // White  (Front)
#define SERVO_MIN 1 /*mS*/
#define SERVO_MAX 1.75 /*mS*/

#include <iomanip>
#include <iostream>
#include "Navio/Ublox.h"
#include "Navio/MPU9250.h"
#include "Navio/PCA9685.h"
#include "Navio/MS5611.h"
#include <Navio/gpio.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

// PID controller
#include "PID_v1.h"
#include <iostream>
#include <cstdio>
#include <ctime>

using namespace std;
using namespace Navio;

long double millis();
void takeoff();
void forward_motion();
void hover(double ax, double ay, double az);
void rotate_CCW();
void land();

int main(int argc, char *argv[])
{
        float latitude_user = 0;
        float longitude_user = 0;
        float latfeet_user;
```

```
        float longfeet_user;
        float latfeet_gps;
        float longfeet_gps;
        float lat_diff;
        float long_diff;
        cout << setprecision(8);
        int dest_dir;
        double ground_level;

        printf("To begin please enter in a set of GPS coordinates with the form
XX.XXXXXX\n");
        printf("Placing a negative sign in front of latitude value causes the computer to treat it as
a West coordinate\n");
        printf("Placing a negative sign in front of the longitude value causes the computer to treat
it as a South coordinate\n");
        cout << "Enter a valid Latitude value between -90 and 90: ";
        cin >> latitude_user;
        while(latitude_user > 90 || latitude_user < -90)
        {
                latitude_user = 0;
                cout << "Invalid Latitude. Please enter a longitude value in the correct range: ";
                cin >> latitude_user;
        }
        cout << "Enter a valid Longitude value between -180 and 180: ";
        cin >> longitude_user;
        while(longitude_user > 180 || longitude_user < -180)
        {
                longitude_user = 0;
                cout << "Invalid Longitude. Please enter a longitude value in the correct range: ";
                cin >> longitude_user;
        }
        cout << "You have entered a GPS coordinate of Latitude: " << latitude_user << endl;
        cout << "With a Longitude of: " << longitude_user << endl;

        // Accuracy down to roughly 4 inches per .000001 change
        // .000139 = 50 feet
        latfeet_user = (latitude_user * (10000/90)) * 3280.4;
        longfeet_user = (longitude_user * (10000/90)) * 3280.4;
        cout << "Latitude in feet: " << latfeet_user << endl;
        cout << "Longitude in feet: " << longfeet_user << endl;

    // This vector is used to store location data, decoded from ubx messages.
    // After you decode at least one message successfully, the information is stored in vector
    // in a way described in function decodeMessage(vector<double>& data) of class
UBXParser(see ublox.h)
```

```
std::vector<double> pos_data;
        MS5611 barometer;
        barometer.initialize();

// create ublox class instance
Ublox gps;

        MPU9250 imu;
        imu.initialize();

        float ax, ay, az, gx, gy, gz, mx, my, mz;

//----------------------------------------------------------------------

while(1) {
   imu.getMotion9(&ax, &ay, &az, &gx, &gy, &gz, &mx, &my, &mz);
                printf("Acc: %+7.3f %+7.3f %+7.3f  ", ax, ay, az);
                printf("Gyr: %+8.3f %+8.3f %+8.3f  ", gx, gy, gz);
                printf("Mag: %+7.3f %+7.3f %+7.3f\n", mx, my, mz);

                /*
                barometer.refreshPressure();
   usleep(10000); // Waiting for pressure data ready
   barometer.readPressure();

   barometer.refreshTemperature();
   usleep(10000); // Waiting for temperature data ready
   barometer.readTemperature();

   barometer.calculatePressureAndTemperature();

   printf("Temperature(C): %f Pressure(millibar): %f\n",
   barometer.getTemperature(), barometer.getPressure());

   sleep(1);
   */

      if(gps.testConnection())
{
   printf("Ublox test OK\n");

   // gps.decodeMessages();

      if (gps.decodeSingleMessage(Ublox::NAV_POSLLH, pos_data) == 1)
      {
         //printf("GPS Millisecond Time of Week: %.0lf s\n", pos_data[0]/1000);
```

```
   printf("Longitude: %lf\n", pos_data[1]/10000000);
   printf("Latitude: %lf\n", pos_data[2]/10000000);
   //printf("Height above Ellipsoid: %.3lf m\n", pos_data[3]/1000);
   printf("Height above mean sea level: %.3lf m\n", pos_data[4]/1000);
   //printf("Horizontal Accuracy Estateimate: %.3lf m\n", pos_data[5]/1000);
   //printf("Vertical Accuracy Estateimate: %.3lf m\n", pos_data[6]/1000);

   latfeet_gps = ((pos_data[2]/10000000) * (10000/90)) * 3280.4;
                        longfeet_gps = ((pos_data[1]/10000000) * (10000/90)) * 3280.4;
                        cout << "Latitude in feet: " << latfeet_gps << endl;
                        cout << "Longitude in feet: " << longfeet_gps << endl;

                        lat_diff = latfeet_gps - latfeet_user;
                        long_diff = longfeet_gps - longfeet_user;

                        cout << "Latitude difference in feet: " << lat_diff << endl;
                        cout << "Longitude difference in feet: " << long_diff << endl;

   // Setting of dest_dir variable
   // N = 1, NE = 2, E = 3, SE = 4 ....
   if(lat_diff > 2) // South
   {
      if(long_diff > 2) // South East
         dest_dir = 4;
      else if(long_diff < -2) // South West
         dest_dir = 6;
      else
         dest_dir = 5;
   }
   else if(lat_diff < -2) //North
   {
      if(long_diff > 2) // North East
         dest_dir = 2;
      else if(long_diff < -2) // North West
         dest_dir = 8;
      else
         dest_dir = 1;
   }
   else if(long_diff > 2) // West
      dest_dir = 7;
   else if(long_diff < -2) // East
      dest_dir = 3;

} else {
   printf("Message not captured\n");
}
```

```
    if (gps.decodeSingleMessage(Ublox::NAV_STATUS, pos_data) == 1)
    {
       printf("Current GPS status:\n");
       printf("gpsFixOk: %d\n", ((int)pos_data[1] & 0x01));

       printf("gps Fix status: ");
       switch((int)pos_data[0]){
          case 0x00:
             printf("no fix\n");
             break;

          case 0x01:
             printf("dead reckoning only\n");
             break;

          case 0x02:
             printf("2D-fix\n");
             break;

          case 0x03:
             printf("3D-fix\n");
             break;

          case 0x04:
             printf("GPS + dead reckoning combined\n");
             break;

          case 0x05:
             printf("Time only fix\n");
             break;

          default:
             printf("Reserved value. Current state unknown\n");
             break;

       }

       printf("\n");

    } else {
       // printf("Status Message not captured\n");
    }   static const uint8_t outputEnablePin = RPI_GPIO_27;

Pin pin(outputEnablePin);
```

```
    if (pin.init()) {
        pin.setMode(Pin::GpioModeOutput);
        pin.write(0); /* drive Output Enable low */
    } else {
        fprintf(stderr, "Output Enable not set. Are you root?");
    }
    }

    // Sets up motors for use
    PCA9685 pwm;

    pwm.initialize();
    pwm.setFrequency(50);

    cout << millis() << endl;
    cout << dest_dir << endl;

    // Set ground level to altitude before lift off
    ground_level = pos_data[4]/1000;

            // Set of if statements to check for current drone status
    if(millis() < 300000)
    {
        takeoff();
    }
    else if(az < .93)
    {
        hover(ax, ay, az);
    }
    else if((lat_diff < 2 && lat_diff > -2) && (long_diff < 2 && long_diff > -2) &&
(pos_data[4]/1000 > (ground_level + 1)))
    {
        land();
    }
    else if((lat_diff < 2 && lat_diff > -2) && (long_diff < 2 && long_diff > -2) &&
(pos_data[4]/1000 < (ground_level + 1)))
    {
        pwm.setPWMmS(NAVIO_RCOUTPUT_1, SERVO_MIN);
        pwm.setPWMmS(NAVIO_RCOUTPUT_2, SERVO_MIN);
        pwm.setPWMmS(NAVIO_RCOUTPUT_3, SERVO_MIN);
        pwm.setPWMmS(NAVIO_RCOUTPUT_4, SERVO_MIN);
        cout << "Done" << endl;
    }


    if(dest_dir == 2 || dest_dir == 8 || dest_dir == 1)
```

```
{
   // turn drone north
   if((mx < 23 || mx > 28) && (my < 51 || my > 57) && az > .93)
   {
      rotate_CCW();
   }
   if(my >= 51 && my <= 57 && mx > 0)
   {
      forward_motion();
   }
}
else if(dest_dir == 4 || dest_dir == 5 || dest_dir == 6)
{
   // turn drone south
   if((mx < -23 || mx > -17) && (my < 52 || my > 59) && az > .93)
   {
      rotate_CCW();
   }
   if(my >= 52 && my <= 59 && mx < 0)
   {
      forward_motion();
   }
}
else if(dest_dir == 3)
{
   // turn drone east
   if((mx < -2 || mx > 8) && (my < 27 || my > 32) && az > .93)
   {
      rotate_CCW();
   }
   if(my >= 27 && my <= 32)
   {
      forward_motion();
   }
}
else if(dest_dir == 7)
{
   // turn drone west
   if((mx < 2 || mx > 7) && (my < 74 || my > 80) && az > .93)
   {
      rotate_CCW();
   }
   if(my >= 74 && my <= 80)
   {
      forward_motion();
   }
```

```
   }
}
}

// Called just after a destination is selected to lift off
void takeoff()
{
        PCA9685 pwm;

        cout << "Taking off" << endl;

   sleep(10);
   pwm.setPWMmS(NAVIO_RCOUTPUT_1, SERVO_MIN);
   pwm.setPWMmS(NAVIO_RCOUTPUT_2, SERVO_MIN);
   pwm.setPWMmS(NAVIO_RCOUTPUT_3, SERVO_MIN);
   pwm.setPWMmS(NAVIO_RCOUTPUT_4, SERVO_MIN);
   sleep(5);
   pwm.setPWMmS(NAVIO_RCOUTPUT_1, SERVO_MAX);
   pwm.setPWMmS(NAVIO_RCOUTPUT_2, SERVO_MAX);
   pwm.setPWMmS(NAVIO_RCOUTPUT_3, SERVO_MAX);
   pwm.setPWMmS(NAVIO_RCOUTPUT_4, SERVO_MAX);
}

// Moves drone forward
void forward_motion()
{
        PCA9685 pwm;

   //Set forward facing motor to minimum throttle
   pwm.setPWMmS(NAVIO_RCOUTPUT_4, 1.1);
   //Set side motors to mid throttle
   pwm.setPWMmS(NAVIO_RCOUTPUT_1, 1.4);
   pwm.setPWMmS(NAVIO_RCOUTPUT_3, 1.4);
   //Set rear motor to max throttle
   pwm.setPWMmS(NAVIO_RCOUTPUT_2, 1.75);

        cout << "-----------------------Moving-----------------------" << endl;
}

// Keeps drone level using PID controller
// Called whenever accelerometer value of az drops below .93
void hover(double ax, double ay, double az)
{
   double Out_x, Out_y, Out_z;

   double Set, ControllerDirection = 0;
```

```
double Kp; double Ki; double Kd; double Input; double Output; double Setpoint;

P PID_x;
P PID_y;
P PID_z;

PCA9685 pwm;

cout << "------------------------Hovering-------------------------" << endl;

// Repeatedly call PID controller until drone is level
Set = 1;
Input = az;        // Input is sensor data
Setpoint = Set;    // Setpoint are values around 0 for stability
PID_z.PID(Input, Output, Setpoint, Kp, Ki, Kd, ControllerDirection);
Output = ((Output/4096) * .35) + 1.4;
Out_z = Output;

Set = 0;
Input = ax;        // Input is sensor data
Setpoint = Set;    // Setpoint are values around 0 for stability
PID_x.PID(Input, Output, Setpoint, Kp, Ki, Kd, ControllerDirection);
Output = ((Output/4096) * .35) + 1.4;
Out_x = Output;

Set = 0;
Input = ay;        // Input is sensor data
Setpoint = Set;    // Setpoint are values around 0 for stability
PID_y.PID(Input, Output, Setpoint, Kp, Ki, Kd, ControllerDirection);
Output = ((Output/4096) * .35) + 1.4;
Out_y = Output;

if(ay > .08)
   {
      // tilt forward
      pwm.setPWMmS(NAVIO_RCOUTPUT_4, Out_y);
      pwm.setPWMmS(NAVIO_RCOUTPUT_1, 1.4);
      pwm.setPWMmS(NAVIO_RCOUTPUT_3, 1.4);
      pwm.setPWMmS(NAVIO_RCOUTPUT_2, 1.4);
   }
else if(ay < -.08)
   {
      // tilt backward
      pwm.setPWMmS(NAVIO_RCOUTPUT_4, 1.4);
      pwm.setPWMmS(NAVIO_RCOUTPUT_1, 1.4);
      pwm.setPWMmS(NAVIO_RCOUTPUT_3, 1.4);
```

```
        pwm.setPWMmS(NAVIO_RCOUTPUT_2, Out_y);
    }
else if(ax > .08)
    {
        // tilt right
        pwm.setPWMmS(NAVIO_RCOUTPUT_4, 1.4);
        pwm.setPWMmS(NAVIO_RCOUTPUT_1, 1.4);
        pwm.setPWMmS(NAVIO_RCOUTPUT_3, Out_x);
        pwm.setPWMmS(NAVIO_RCOUTPUT_2, 1.4);
    }
else if(ax < -.08)
    {
        // tilt left
        pwm.setPWMmS(NAVIO_RCOUTPUT_4, 1.4);
        pwm.setPWMmS(NAVIO_RCOUTPUT_1, Out_x);
        pwm.setPWMmS(NAVIO_RCOUTPUT_3, 1.4);
        pwm.setPWMmS(NAVIO_RCOUTPUT_2, 1.4);
    }
else if(ax < -.08 && ay > .08)
    {
        // tilt forward-left
        pwm.setPWMmS(NAVIO_RCOUTPUT_4, Out_y);
        pwm.setPWMmS(NAVIO_RCOUTPUT_1, Out_x);
        pwm.setPWMmS(NAVIO_RCOUTPUT_3, 1.4);
        pwm.setPWMmS(NAVIO_RCOUTPUT_2, 1.4);
    }
else if(ax > .08 && ay > .08)
    {
        // tilt forward-right
        pwm.setPWMmS(NAVIO_RCOUTPUT_4, Out_y);
        pwm.setPWMmS(NAVIO_RCOUTPUT_1, 1.4);
        pwm.setPWMmS(NAVIO_RCOUTPUT_3, Out_x);
        pwm.setPWMmS(NAVIO_RCOUTPUT_2, 1.4);
    }
else if(ax < -0.8 && ay < -.08)
    {
        // tilt backward-left
        pwm.setPWMmS(NAVIO_RCOUTPUT_4, 1.4);
        pwm.setPWMmS(NAVIO_RCOUTPUT_1, Out_x);
        pwm.setPWMmS(NAVIO_RCOUTPUT_3, 1.4);
        pwm.setPWMmS(NAVIO_RCOUTPUT_2, Out_y);
    }
else if(ax > .08 && ay < -.08)
    {
        // tilt backward-right
        pwm.setPWMmS(NAVIO_RCOUTPUT_4, 1.4);
```

```
            pwm.setPWMmS(NAVIO_RCOUTPUT_1, 1.4);
            pwm.setPWMmS(NAVIO_RCOUTPUT_3, Out_x);
            pwm.setPWMmS(NAVIO_RCOUTPUT_2, Out_y);
        }
}

// Used to point drone in the right direction after takeoff
void rotate_CCW()
{
        PCA9685 pwm;

   // Set forward facing motor to minimum throttle
   pwm.setPWMmS(NAVIO_RCOUTPUT_4, 1.1);
   // Set side motors to max throttle
   pwm.setPWMmS(NAVIO_RCOUTPUT_1, 1.75);
   pwm.setPWMmS(NAVIO_RCOUTPUT_3, 1.75);
   // Set rear motor to max throttle
   pwm.setPWMmS(NAVIO_RCOUTPUT_2, 1.1);
   cout << "------------------------------Rotating-----------------------------------" << endl;
}

// Called when GPS coordinates indicate the drone is at it's destination
void land()
{
        PCA9685 pwm;

   for(float i = 1.4; i > 1.0; i = i - .0001)
   {
      pwm.setPWMmS(NAVIO_RCOUTPUT_4, i);
      pwm.setPWMmS(NAVIO_RCOUTPUT_1, i);
      pwm.setPWMmS(NAVIO_RCOUTPUT_3, i);
      pwm.setPWMmS(NAVIO_RCOUTPUT_2, i);
      usleep(1000);
      cout << "Landing" << endl;
   }

}

// PID Functions ------------------------------------------------------------

// Constructor (executes PID related functions)
void P::PID(double Input, double Output, double Setpoint,
      double Kp, double Ki, double Kd, int ControllerDirection)
{
   myOutput = Output;                // Output is the PWM signal sent to the motors
   myInput = Input;                  // Input is sensor data
```

```
   mySetpoint = Setpoint;              // Setpoint are values for the sensors that they should not
cross to maintain stability
          inAuto = true;

          P::SetOutputLimits(0, 4096, myOutput);  // 1.4ms = 0 and 1.75ms = 4096 (2^12)

   SampleTime = 1;                                                    //default Controller Sample
Time is 0.001 seconds
          lastTime = millis() - SampleTime;

   P::SetControllerDirection(ControllerDirection);

   kp = 2;                          // Proportional tuning value
   ki = .5;                          // Integral tuning value
   kd = .05;                         // Derivative tuning value

   P::Compute(myOutput, myInput, mySetpoint);
}

// Returns true when the output is computed, false when nothing has been done.
bool P::Compute(double myOutput, double myInput, double mySetpoint)
{
  unsigned long now;

  now = millis();

  unsigned long timeChange = (now - lastTime);
  if(!inAuto) return false;
  if(timeChange >= SampleTime)
  {
    /*Compute all the working error variables*/
    double input = myInput;
    double error = mySetpoint - input;
    ITerm += (ki * error);
    if(ITerm > outMax) ITerm = outMax;
    else if(ITerm < outMin) ITerm = outMin;
    double dInput = (input - lastInput);

    /*Compute PID Output*/
    double output = kp * error + ITerm - kd * dInput;

          if(output > outMax) output = outMax;
    else if(output < outMin) output = outMin;
          myOutput = output;

    /*Remember some variables for next time*/
```

```
      lastInput = input;
      lastTime = now;
            return true;
    }
    else return false;
}

// Clamps output to between 0 and 4096
void P::SetOutputLimits(double Min, double Max, double myOutput)
{
    if(Min >= Max) return;
    outMin = Min;
    outMax = Max;

    if(inAuto)
    {
            if(myOutput > outMax) myOutput = outMax;
            else if(myOutput < outMin) myOutput = outMin;

            if(ITerm > outMax) ITerm = outMax;
            else if(ITerm < outMin) ITerm = outMin;
    }
}

/* SetControllerDirection
 * The PID will either be connected to a DIRECT acting process (+Output leads
 * to +Input) or a REVERSE acting process(+Output leads to -Input.) */
void P::SetControllerDirection(int Direction)
{
    if(inAuto && Direction != controllerDirection)
    {
          kp = (0 - kp);
      ki = (0 - ki);
      kd = (0 - kd);
    }
    controllerDirection = Direction;
}

long double millis()
{
    long double millisecond;

    millisecond = std::clock();

    return millisecond;
}
```

Appendix B Makefile

```
CC = g++
CFLAGS =-c -Wall
NAVIO = ../../Navio
INCLUDES = -I ../..

all: AccelGyroMag

AccelGyroMag: AccelGyroMag.o Ublox.o PCA9685.o gpio.o MS5611.o I2Cdev.o
MPU9250.o

	$(CC) AccelGyroMag.o Ublox.o PCA9685.o gpio.o MS5611.o I2Cdev.o
MPU9250.o -o AccelGyroMag
	rm -rf *.o

AccelGyroMag.o: AccelGyroMag.cpp
	$(CC) $(INCLUDES) $(CFLAGS) AccelGyroMag.cpp

PCA9685.o: $(NAVIO)/PCA9685.cpp
	$(CC) $(INCLUDES) $(CFLAGS) $(NAVIO)/PCA9685.cpp

gpio.o: $(NAVIO)/gpio.cpp
	$(CC) $(INCLUDES) $(CFLAGS) $(NAVIO)/gpio.cpp

MS5611.o: $(NAVIO)/MS5611.cpp
	$(CC) $(INCLUDES) $(CFLAGS) $(NAVIO)/MS5611.cpp

I2Cdev.o: $(NAVIO)/I2Cdev.cpp
	$(CC) $(INCLUDES) $(CFLAGS) $(NAVIO)/I2Cdev.cpp

MPU9250.o: $(NAVIO)/MPU9250.cpp
	$(CC) $(INCLUDES) $(CFLAGS) $(NAVIO)/MPU9250.cpp

Ublox.o: $(NAVIO)/Ublox.cpp
	$(CC) $(INCLUDES) $(CFLAGS) $(NAVIO)/Ublox.cpp

clean:
	rm -rf *.o gps
```