ESE350

April 30th, 2010

Team Members:
William Etter
Paul Martin

[WYVERN QUADROTOR]

ESE350 – Embedded Systems
Electrical and Systems Engineering
School of Engineering and Applied Science
University of Pennsylvania

TABLE OF CONTENTS

TABLE OF FIGURES

WYVERN QUADROTOR

WILLIAM ETTER AND PAUL MARTIN

1. INTRODUCTION

The term 'Quadrotor' is short for quadrotor helicopter and characterizes a helicopter constructed from four distinct rotors. The quadrotor platform is interesting in both design and concept. Similar to a normal helicopter, the rotational forces of the blades induce moments in the quadrotor body, causing it to yaw and become unstable. Likewise, in order to stabilize a quadrotor, sophisticated controls must be implemented so that the body does not pitch or roll out of stability. With proper controls in place, a quadrotor system offers advantages over conventional aircraft. Quadrotors are more maneuverable than both fixed-wing and traditional rotorcraft. Since quadrotors use fixed-pitch blades, mechanical design and maintenance is simplified. In addition, both smaller motors and props can be used since a set of four are utilized to generate lift. These benefits make quadrotors desirable platforms for UAV systems and research with countless possibilities for expansion.

Most quadrotor systems are built using RC transmission units, but this requires the unit to be under human supervision at all times. The Wyvern Quadrotor is not yet autonomous, but it does lend itself well to expansion in that direction; our unit is both controlled by and runs on ATMega32u4 microcontrollers. This allows the possibility of running localization and position control outside of the main control and stabilization loop.

The majority of quadrotors are very expensive. The possibility of creating a budget quadrotor robot opens the door to new and interesting experiments such as group communication and flocking patterns. Within the scope of this course, building a quadrotor robot offers an interesting application of wireless communication, high frequency sensor and ADC measurements, and a combination of mechanical design, electrical design, and controls theory.

2. PROJECT OBJECTIVES

The main goal of the Wyvern project is to build a working quadrotor from the ground up. We have developed a list of goals that we intend to meet in designing our system. These are listed below.

- Design a light mechanical body for a quadrotor helicopter
- Design PID controls for stabilization in yaw, pitch, roll, and altitude using an IMU
- Command the unit wirelessly with RF
- Design an inexpensive system to be used as a platform for future projects

3. QUADROTOR CONTROL THEORY

3.1 FLIGHT CONTROL

The algorithm for correcting for instability in a quadrotor helicopter is quite simple. There are straightforward logic paths for correcting for yaw, pitch, roll, and altitude. Correcting for absolute position requires a more complex method of localization. As of now the Wyvern system does not implement any form of localization (as in a GPS) and therefore has no position control.

The following controls assume the configuration shown in the figure below.

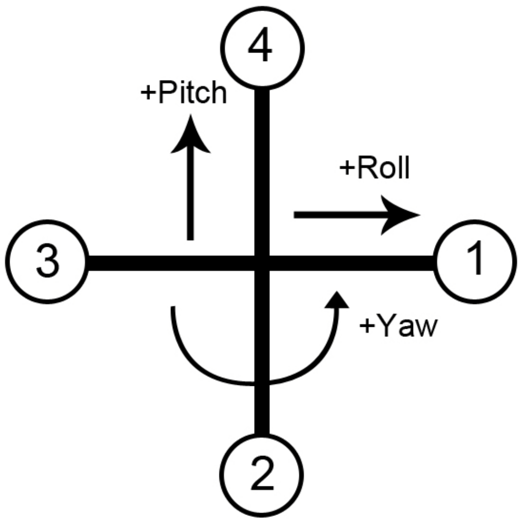

FIGURE 1: MOTOR LAYOUT

With this orientation in mind, we adopt the following controls:

- A positive pitch is effected by decreasing motor 4 and increasing motor 2.
- A positive roll is effected by decreasing motor 1 and increasing motor 3.
- A positive yaw is effected by decreasing one pair of counter rotating rotors (3 and 1 or 4 and 2) and increasing the other. This will depend on which pairs are installed on which motors.

3.2 DIRECTION COSINE MATRIX (DCM)

In order to implement controls to counter any movement in yaw, pitch, or roll, we need very accurate data from our inertial measurement unit (IMU). Our IMU consists of a 3-axis accelerometer, a 3-axis gyroscope, and a 3-axis magnetometer (used only for drift correction). In order to convert the inertial axes from the IMU into Euler angles referenced from the ground plane, we used a direction cosine matrix. In doing so, we take a vector of angles with respect to the quadrotor (local), Q_L and a vector with respect to the ground, Q_G. The ground Euler angles can be found through linear manipulation of the local vector Q_L by multiplying by a rotational matrix R, where R is given by multiplying by a series of three rotational matrices about yaw, pitch, and roll. The resulting rotational matrix is below.

$$\mathbf{R} = \begin{bmatrix} \cos\theta\cos\psi & \sin\phi\sin\theta\cos\psi - \cos\phi\sin\psi & \cos\phi\sin\theta\cos\psi + \sin\phi\sin\psi \\ \cos\theta\sin\psi & \sin\phi\sin\theta\sin\psi + \cos\phi\cos\psi & \cos\phi\sin\theta\sin\psi - \sin\phi\cos\psi \\ -\sin\theta & \sin\phi\cos\theta & \cos\phi\cos\theta \end{bmatrix}$$

FIGURE 2 : DIRECTION COSINE MATRIX

3.3 PROPORTIONAL, INTEGRAL, DERIVATIVE (PID) CONTROLLER

PID, or proportional-integral-derivative controllers are loop feedback controls that aim to correct for a given error. Often, the effect of a PID is modeled as a step response. One PID control is needed for each parameter to be corrected. In our case, we need a PID loop for yaw, pitch, and roll. Future designs might implement PID for altitude and x/y position as well. In the embedded environment, the proportional, integral, and derivative terms are calculated as follow:

$$P = \textit{desired} - \textit{actual}$$

$$I = I + \textit{actual}$$

$$D = \textit{current\_error} - \textit{past\_error}$$

Note that the derivative term can also be determined by accessing the direct ADC values from the rate gyroscopes for a given axis of rotation.

Each of the three terms has a gain K_P, K_I, or K_D that is tuned to minimize rise time to the step input and to minimize oscillation and overshoot. For each of the three Euler angles, we compute a sum comprised of the gains and their corresponding terms, e.g.

$$\textit{pitch\_sum} = Kp*P + Ki*I + Kd*D$$

At the end of the PID loop, the appropriate summations were added and subtracted from each of the four motors as seen in the code snippet below. The *thrustFloor* term corresponds to the thrust that the user commands on the joystick.

```
PWM_MOT1 = thrustFloor - roll_sum + yaw_sum;
PWM_MOT2 = thrustFloor + pitch_sum - yaw_sum;
PWM_MOT3 = thrustFloor + roll_sum + yaw_sum;
PWM_MOT4 = thrustFloor - pitch_sum - yaw_sum;
```

The entire PID loop must be completed at a relatively high frequency to ensure fast motor response. Without a fast PID response the quadrotor will not properly stabilize.

4. QUADROTOR DESIGN

4.1 FRAME

We believe that the current frame will prove sufficient for our purposes. It is light, sturdy, easy to construct, and capable of housing the components we require securely. However, given more time several improvements are possible. When adding the electrical components and wire to our frame, we noticed that adding specific wire routing might prove beneficial during the modeling process. Moreover, our nodes seem more robust than necessary and we could probably stay within our design goal of robustness while reducing mass of material used. In the future use of carbon fiber tubing and sheets, and perhaps even thermoplastics, might allow for an equally functional structure that is lighter and more aesthetically pleasing. Though the aluminum support rods seem to hold our motors well enough for our purposes, some of the arms had the tendency to torque to offset angles and required readjusted before each flight. Also, screwing the ABS to the aluminum rods (for ease of assembly/disassembly) proved detrimental to the structural integrity as the rods dented with the added strain. We hope that future iterations will not have these issues if the arms are constructed out of carbon fiber.

4.1.1 DESIGN

We went into the design process with the goal of creating a simple, low-cost, robust, and manufacturable quadrotor platform. The general design consists of three major components: the motor nodes, the support rods, and the central node. It was essential that the design be reconfigurable. Thus a multiple-node schematic was followed that allowed for a degree of modularity that enables us to adapt the quadrotor to multiple roles, propellers, motors, etc. With this in mind, no part of the quadrotor is glued together. However, as stated before this led to slight deformation of the aluminum rods from the use of locking screws. Also, it is possible that the high frequency vibrations induced by the motor/rotor might reveal a need for gluing in subsequent versions. Additionally, we have attempted to utilize electrical connectors over soldering to protect connections with the battery, ESCs and motors at a slight weight cost.

Below are images of our SolidWorks model and of the design process. In the SolidWorks model the rotors are represented as clear cylinders.

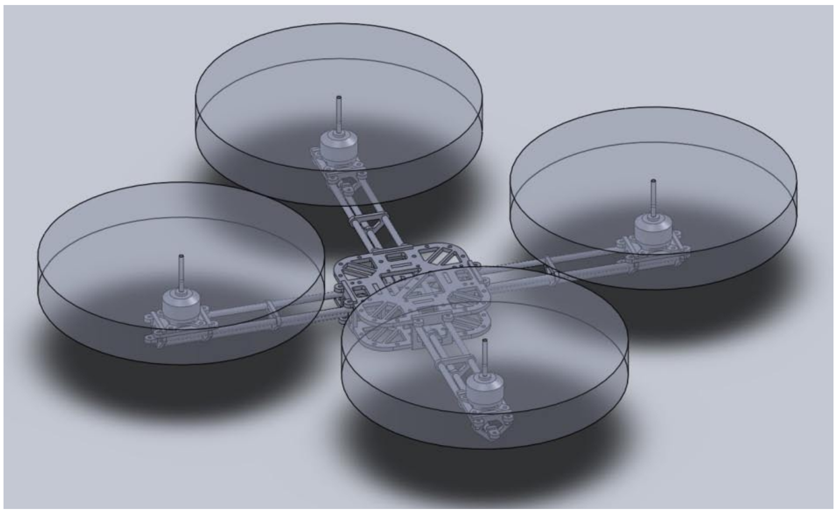

FIGURE 3: FRAME SOLDIWORKS ISOMETRIC VIEW

4.1.2 MATERIALS

With respect to materials, the motor nodes and central node are cut out of 1/8 inch Acrylonitrile Butadiene Styrene (ABS), while the support rods are 3/8 inch hollow aluminum. Additionally, a number of 3 mm steel screws are used for fixing the motor nodes and central nodes to the rods as well as for fixing the motors and boards to the nodes. We felt that the aforementioned materials allowed for rapid assembly while maintaining a reasonable overall weight. This was confirmed by our overall frame mass which was about 130 grams. Improvements in the use of materials will be noted in the conclusion section.

4.1.3 ASSEMBLY

Assembly proved to be quite rapid and easy. A few iterations for finding the optimal sizing for the node/rod interface were necessary, but meticulous modeling ensured that the majority of our parts fit well. It was apparent that the rods would have to overlap in some way, which we solved by crimping the rods at intersection points. Holding the nodes in place was accomplished through the use of screws that pressed down on the rods, providing a surprisingly robust fix. Careful observation of some of our thrust test videos will reveal a nut that falls off of a screw. While this was not an issue during the test since each screw had an additional redundant nut, we decided to eliminate such uncertainty in our quadrotor by tapping the ABS. The resulting screw friction combined with the dampening qualities of the ABS plastic should be enough to prevent the screws from rotating during operation and coming free.

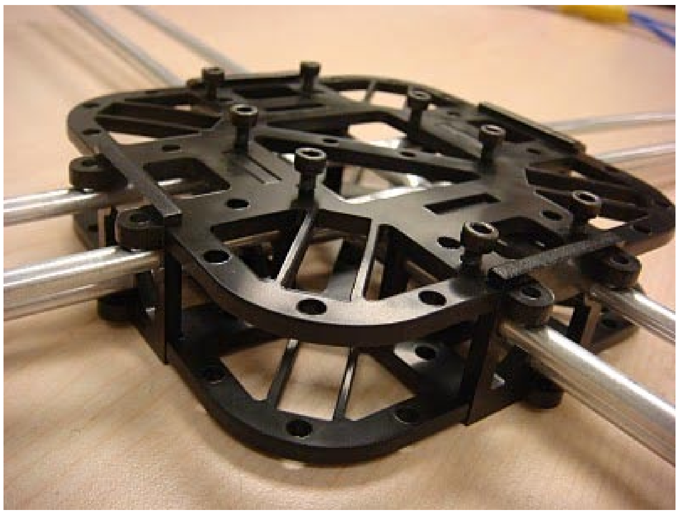
FIGURE 4: WYVERN CENTRAL NODE ASSEMBLY

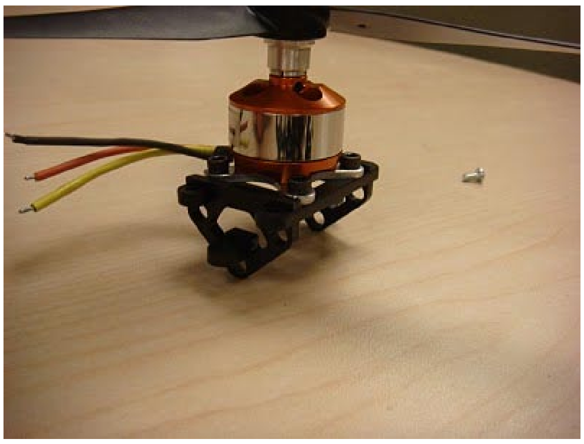
FIGURE 5: MOTOR NODE ASSEMBLY

4.1.4 CENTRAL NODE CONFIGURATION

The slots in the central node house the ESCs for our brushless motors (Mystery A2208-14). Wires from the ESCs are routed through the openings at the top of the node through those between the rods out to the motors. The battery is strapped on the bottom of the central node, and any future altimeter will be placed under/inside the node as well (to limit air turbulence from affecting readings). Our control, IMU, and wireless boards are configured to be placed between the ESCs on the top of the central node and nylon spacers.

4.2 POWER SUPPLY

After some research to determine which characteristics (weight, Amp-hours, output current) would affect us the most, we decided to use 3-cell (11.1V) Lithium Polymer (LiPo) batteries rated for 2200 mAh and 25C. This means that the battery can provide 25x2.2 Amps, or a little over 50 Amps, of continuous current as well as three second peaks of up to 100 Amps.

To charge the batteries, we chose a high-end LiPo battery charger with on-board balancer. Balancing the LiPo cells equalizes the voltage and results in both longer run-time and battery life. The Thunder AC6 battery charger is able to handle several types of batteries (LiPo, LiIon, LiFe, NiCd, and NiMH) as well as 1 to 6 cells. The charger can charge from 0.1 to 5 Amps and discharge (to give a fresh charge to a battery) from 0.1 to 1 Amp.

Deans connectors were used to connect the battery to the charger and to the ESCs of the quadrotor itself. These connectors can handle a large amount of current and do not easily come disconnected. However, when plugging in the battery there is an initial spark due to the charging current of the large capacitors on each ESC. This will eventually create buildup on the connector that will increase connection resistance. On future iterations we intend to use one of two possible solutions. The first option is to change to 4mm bullet style connectors. These will still spark, but the increased surface area in contact will reduce the effect that the sparks have on the lifetime of the connection. The second option is to wire a large resistor in parallel to a switch that will allow us to slowly charge the capacitors and then fully connect the batteries to the ESCs.

FIGURE 6: BLUE LIPO LITHIUM POLYMER BATTERY[1]

Battery Specifications	
Voltage	11.4V
Cells	3
Capacity	2200 mAh
Max Continuous Discharge	25C (50 A)
Max Burst Discharge	50C (100 A)
Weight	152 g
Dimensions	24.4 x 35 x 103 mm

TABLE 1: LIPO BATTERY SPECIFICATIONS

FIGURE 7: DEANS CONNECTORS[2]

[1] http://www.hobbypartz.com/83p-2200mah-3s1p-111-25c.html
[2] http://www.purehobby.com/Connectors/DeansMain.htm

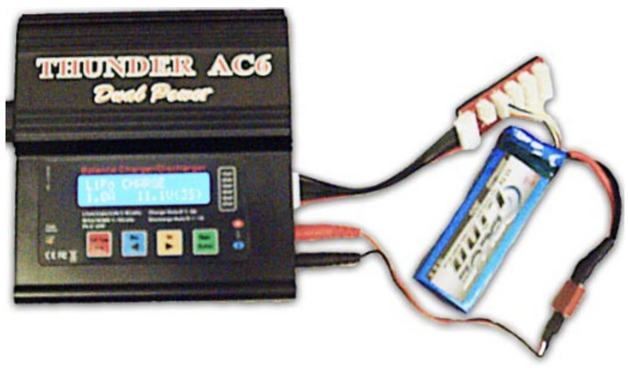

FIGURE 8: LIPO CHARGER

4.3 QUADROTOR POWERPLANT

Below is a simple block diagram of the Wyvern, the wireless control unit, and a PC that illustrates the current configuration of our various code and hardware blocks.

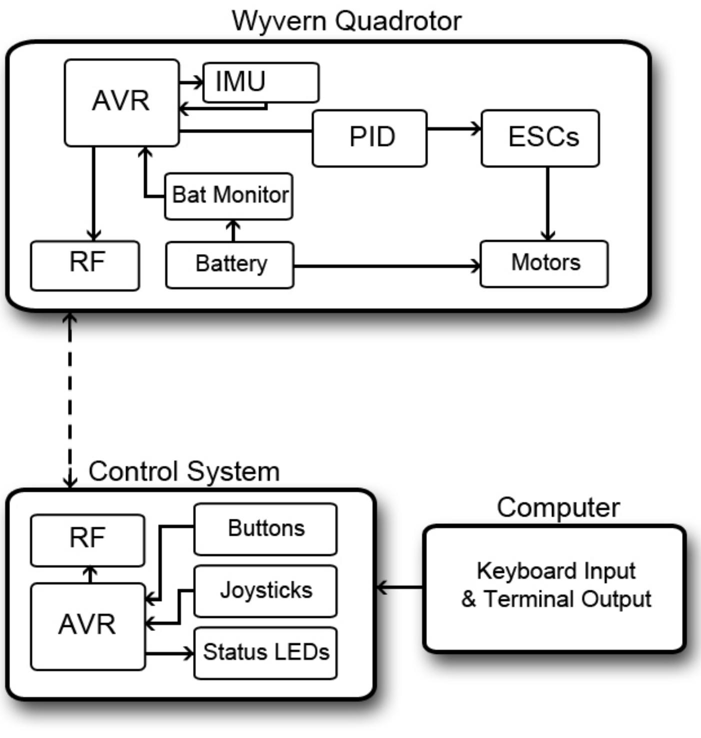

FIGURE 9: SYSTEM FLOW

4.3.1 MOTORS

Brushless DC motors are more efficient and longer lasting than brushed motors, offering lower cost in the long run and a greater thrust to power ratio. A downside to using this type of motor is that it requires a particular form of dedicated motor controller and is often more expensive than a brushed motor. The proposed quadrotor will utilize the Mystery (or Suppo) A2208-14 motor which, for the approximated weight, will draw around 2.5 Amps at 11 volts for stable flight with an overall lifting capability of 600 grams at higher currents. This will provide the entire unit an overall lifting capability of over 2 kilograms. Moreover, the A2208 motor has a rise of 1450 K/volt. The aforementioned properties should be sufficient for stability and most maneuvering. Variation of rotors will allow for differing lifting capabilities at greater current consumptions.

FIGURE 10: BRUSHLESS MOTOR[3]

Motor Specifications	
Voltage	7.4V – 11.1V
RPM/V	1450 RPM/V
Max Efficiency	80%
Max Current	12 A for 60 seconds
Max Watts	130 W
Resistance	0.140 Ω
Weight	39.3 g
Dimensions	27.8 x 27 x 33 mm

TABLE 2: MOTOR SPECIFICATIONS

4.3.2 ESCS

Brushless ESCs are used to allow motor control using a convenient PWM output from the MaEvArM. Each ESC produces the necessary sinusoids to drive the brushless motors at the necessary current. Using back-EMF the motor controller is able to determine the current phase of the motor and apply the correct voltage in sequence and thereby commutate the motor. A schematic of how the ESCs were connected to the motors can be found in Appendix B.

FIGURE 11: BRUSHLESS ESC[4]

[3] http://www.dinodirect.com/mystery-a2208-14-1450kv-outrunner-brushless-motor-for-rc-helicopter.html

[4] http://www.dinodirect.com/Mystery-30A-Brushless-IN-BEC-Electronic-Speed-Motor-Controller-ESC-Yellow.html

ESC Specifications	
Voltage	5.6V – 16.8V
BEC Output	5V, 1 A
Max Current	30 A
Weight	25 g

TABLE 3: ESC SPECIFICATIONS

4.3.3 PROPS

We decided to try several propeller designs in an attempt to find quiet, fast, and high thrust combinations for our motors. The first set consists of two-blade, 8x3.8 props. (8 inch diameter, 3.8 blade pitch).

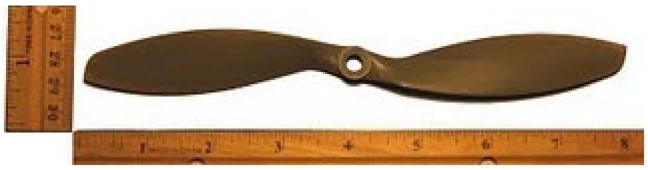

FIGURE 12: 8X3.8 DUAL-BLADE PROP

These are fairly large rotors and draw fairly low current for our hover estimate. The second set consists of tri-blade, 7x3.5 props (7 inch diameter, 3.5 blade pitch).

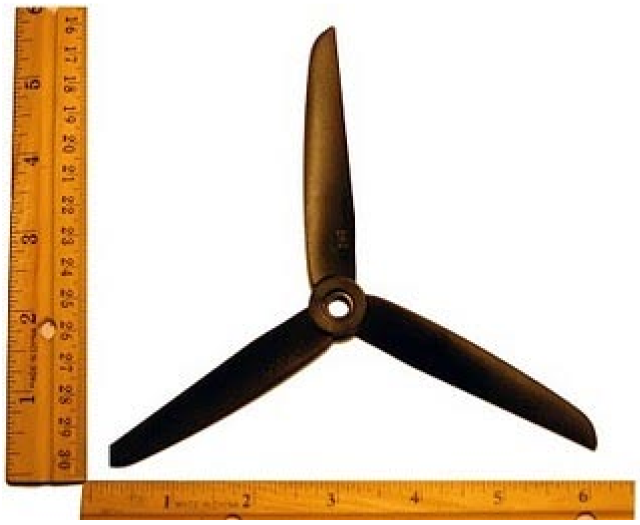

Figure 13: 7x3.5 Tri-Blade Prop

After some research, it was found that tri-blade props can be quieter than dual-blade when run at the same speed, though they draw a couple hundred milliamps more at hover. The test results for both prop sets can be seen in the figures below (Figure 14: Two-Blade 8x3.8 vs. Tri-Blade 7x3.5 and Figure 15: Two-blade vs. Tri-Blade Zoomed).

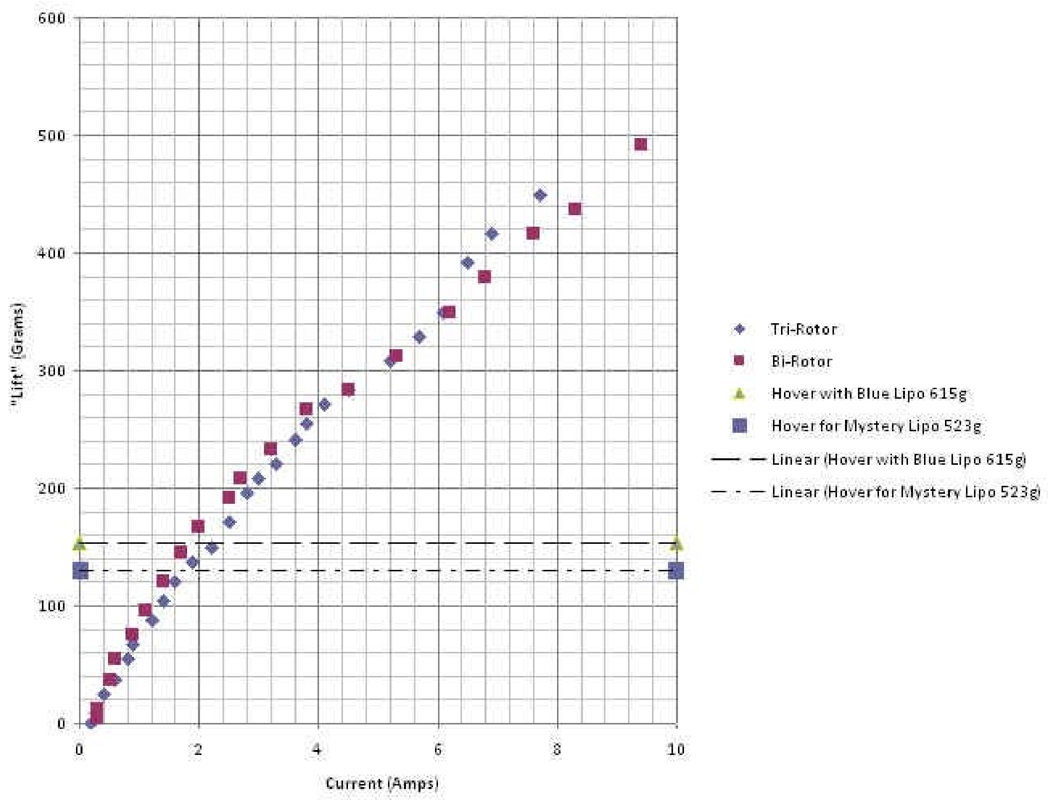

FIGURE 14: TWO-BLADE 8X3.8 VS. TRI-BLADE 7X3.5

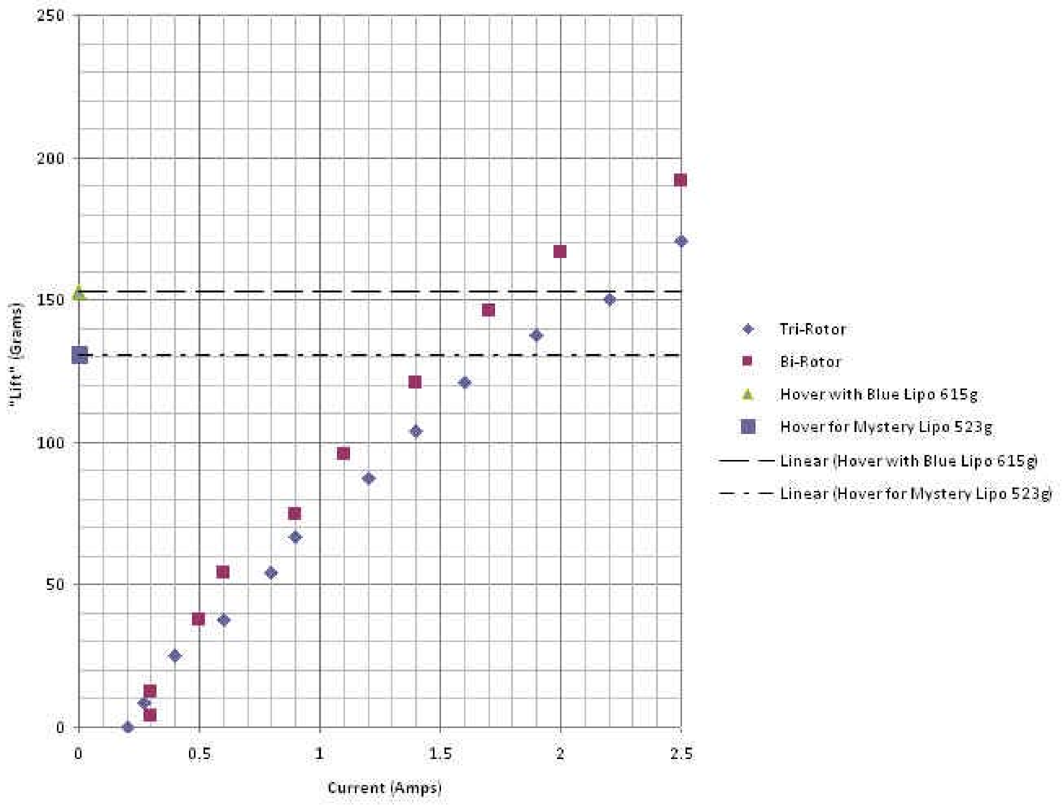

FIGURE 15: TWO-BLADE VS. TRI-BLADE ZOOMED

The thrust test platform used to measure the data presented by the plots above consisted of a lever made from ABS and a metric scale, such that prop thrust generated a proportional weight on the scale. By scaling this thrust by the ratio of the lever lengths we were able to get an approximate measurement of lift (in grams) versus current for our two props.

Our findings indicate that the larger diameter propellers have the lowest hover currents while the smaller diameters have better high thrust properties. The difference in efficiency at high and low current output is particularly evident when plotting RPM versus current as in the figure below (Figure 16: RPM vs. Current for Two-Blade and Tri-Blade props). Note that at higher currents the dual-blade prop has a lower RPM gain per Amp than the 7x3.8 tri-blade, and the tri-blade has a higher RPM gain per Amp than the dual-blade at lower amps.

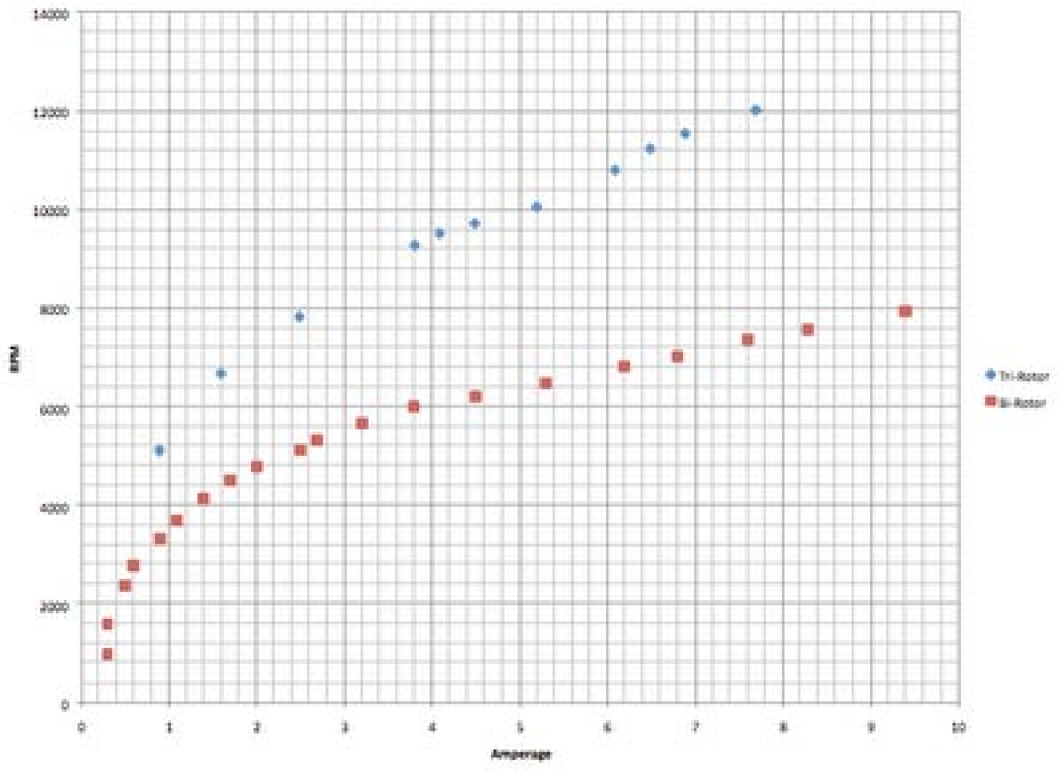

FIGURE 16: RPM VS. CURRENT FOR TWO-BLADE AND TRI-BLADE PROPS

To determine the efficiency of our system, we measured the entire weight of our body (conservatively adding additional weight to account for future wires, sensors, and connections) with a particular battery. Then, the hover time was calculated based on battery capacity as well as the current required to maintain the quadrotor hovering. For example, using Blue Lipo lithium polymer batteries with 2200 mAh (at about 185g including connector and wires) we have a total body weight of 615g. In order to hover we need each motor to generate 615/4 or about 154 grams of "thrust". With the 7x3.8 tri-blade this corresponds to about 2.2 Amps per motor (8.8 Amps total). With the dual-blade this corresponds to about 1.8 Amps per motor (7.2 Amps total). With our 2200 mAh battery, this gives a flight time of:

Tri-blade -- 2.2 Amps*Hour * (60 min / Hour) * (1 / 8.8 Amps) = 15 minutes
Dual-blade -- 2.2 Amps*Hour * (60 min / Hour) * (1 / 7.2 Amps) = 18.3 minutes

This indicates that at hovering, the dual blades are (18.3-15) / 15 * 100% = 22% more efficient with respect to the smaller tri-blade props. Combined with the decreased vibrations due to the props being sturdier and balanced, this convinced us to use the two-blade in our final design.

4.4 INERTIAL MEASUREMENT UNIT (IMU)

For our IMU we used a 9 Degrees of Freedom (6 Degrees of Motion) board from SparkFun. This board has a dedicated ATmega 328P microcontroller to obtain values from the accelerometers, gyroscopes, and magnetometers and output a constant stream of angle values to the MaEvArM. At 1.95" x 1.10", the board is only slightly larger than the MaEvArM itself. The IMU has been programmed to communicate with a baud rate of 250,000 with the MaEvArM over UART. Despite this high baud rate, a full set of data is transmitted to the MaEvArM at a slow 120 Hz. Note that our PID loop only runs at 120 Hz, indicating that this is a major bottleneck in the speed of our system.

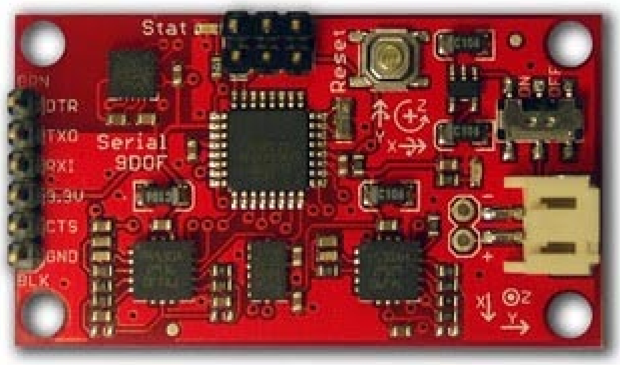

FIGURE 17: 6DOF RAZOR IMU

4.4.1 ACCELEROMETER

The IMU contains a three-axis accelerometer. This is to sense accelerations in the x, y, and z axes and can be used to determine orientation with respect to ground by using the earth's gravitational acceleration. However, when the quadrotor itself is accelerating these values are less precise and are therefore used in conjunction with the gyroscopes.

4.4.2 GYROSCOPE

A three-axis rate gyroscope is needed to determine change in angles. This is needed for accurate response in the derivative terms of the PID. An alternative method would be to determine change in angle by subtracting the current angle from a past angle using only the accelerometer, but this is slow and inaccurate.

4.4.3 MAGNETOMETER

A three-axis magnetometer is used for drift correction. By using the magnetometer we are able to give absolute orientation with respect to the earth's magnetic field. For instance, the yaw measurement is 0 degrees when motor four is pointed due north.

4.5 RF COMMUNICATION

Wireless communication is accomplished by using two nRF24L01+ transceivers with chip antennas from SparkFun. These boards communicate with the MaEvArM using SPI and are capable of transmitting up to 100 m line-of-sight.

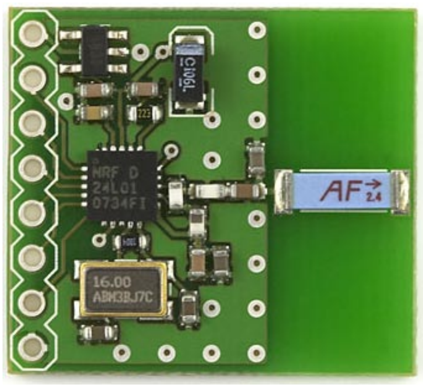

FIGURE 18: RF COMMUNICATION BOARD[5]

4.6 MICROCONTROLLER

The MaEvArM is a general purpose microcontroller platform using the ATmega32U4. The primary purpose of the MaEvArM on the robot platform is dedicated motor control to the four (4) ESC brushless motor controllers using 16-bit channels. A schematic of the PCB used with this microcontroller can be found in Appendix B.

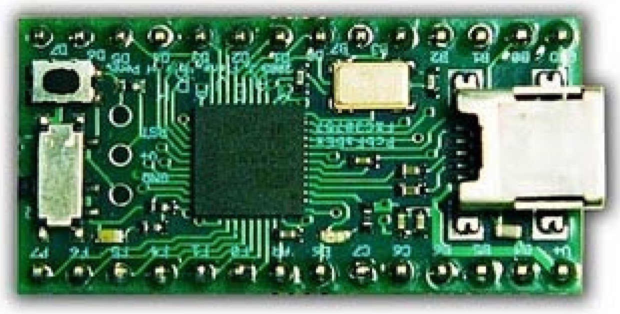

FIGURE 19: MAEVARM MICROCONTROLLER[6]

4.7 CONTROLLER

The wireless controller uses a second MaEvArM and wireless board to communicate with the quadrotor. We incorporated 2 joysticks from SparkFun on our wireless transmitter to allow us to control Thrust/Yaw with the left stick and Pitch/Roll with the right stick. We also used four LEDs (red, green, yellow, and blue) as status indicators (such as normal operation, wireless transmission/reception, etc.).

The inputs from the controller joysticks ranged from 0 to 5V and therefore from 0 to 1023 when using the 10 bit ADC on the ATmega32u4. This corresponded to +/- 512 resolution for yaw, pitch, and roll, and +512 resolution for thrust (ignoring negative thrust inputs since the joysticks returned to center when released). This was scaled appropriately to give smoother PWM responses. The idle thrust was also increased from (on a scale from 1000 to 2000) 1106 (our very lowest idle PWM) to 1200 in order to give us more resolution in controlling the thrust. Note that hovering is around 1300.

[5] http://www.sparkfun.com/commerce/product_info.php?products_id=691
[6] http://alliance.seas.upenn.edu/~medesign/wiki/index.php/Guides/MaEvArM

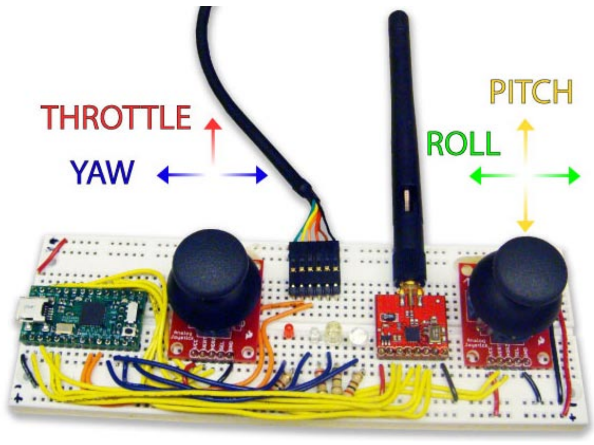

FIGURE 20: WYVERN CONTROLLER

In the future, buttons will be added to the controller for start and emergency stop. As of now we issue those commands by pressing 's' on the PC connected to the controller to start and SPACEBAR for emergency stop.

The controller receives yaw, pitch, roll, angular rate data, PID terms, and battery level from the quadrotor. If the battery is below 10.0 Volts, all LEDs on the controller will blink to indicate a low battery level to the user. A schematic of the controller can be found in Appendix B.

5. CODE

The code was broken into three sections: common code, quadrotor code, and controller code. The quadrotor and controller code were written in C using the AVR Studio IDE, while the IMU code was written using the Arduino IDE. The code structure was as follows:

- Include:
  - adc.h
  - commands.h
  - controller.h
  - pid.h
  - pwm.h
  - uart.h
  - wyvern-rf.h
  - wyvern.h
  -
- Wyvern:
  - Wyvern.c
- Controller:
  - WyvernController.c
- IMU (using open source AHRS code):
  - ADC.pde
  - Compass.pde
  - DCM.pde
  - Matrix.pde
  - Output.pde
  - SF9DOF_AHRS.pde
  - Vector.pde

5.1 COMMON CODE (INCLUDE)

5.1.1 MICROCONTROLLER HEADER FILE (WYVERN.H)

This is the main header file for the microcontroller. It contains several standard C include files, basic register manipulations, and register definitions. In addition, several definitions for changing the status LEDs as well the different structs (used for communication and flight control) are located here. Lastly, functions for initializing the MaEvArM itself and setting up the different port registers and clock speed are located in this file.

5.1.2 ANALOG TO DIGITAL CONVERSION(ADC) (ADC.H)

This file initializes ADC with init_ADC() and retrieves an ADC value as a uint16_t when get_adc(channel) is invoked.

5.1.3 UART (UART.H)

This file sets up buffered interrupt-controlled serial communication. This saves precious CPU time, allowing for our PID calculations, PWM Control, and Wireless Receive to run continuously. This code uses a Circular Buffer technique to get and store data when the CPU is in use and read it when it has time to. This buffer is cleared when a complete data set is obtained (to prevent overwriting and errors).

5.1.4 RF COMMUNICATION (WYVERN-RF.H)

This file includes a several register defines and controls the wireless communication using SPI to communicate with the RF transceivers. This is a modified version of code originally written by Joe Romano and Jonathan Fiene. Functions invoked by the user include RFtransmit() and RFreceive().

5.2 QUADROTOR CODE

5.2.1 QUADROTOR MAIN CODE (WYVERN.C)

This file handles pin change interrupts for wireless, overflow interrupts for timing, receives IMU data from the Razor via serial, calls the appropriate functions and sets the appropriate registers to initialize the ATMega32u4, UART communication, PWM pins, and ADC pins. This file also runs the ADC to determine battery voltage and populates the wireless transmit packet from Wyvern to controller every iteration.

5.2.2 PULSE-WIDTH MODULATION (PWM) (PWM.H)

This file handles four 16-bit PWM channels (0-65536 resolution) to control the four motor controller (Brushless ESCs). All four PWM channels are set to output at 50 Hz (the frequency specified by the ESCs) with duty cycles between 1 and 2ms. This allows for about 1000 point resolution between 0 velocity and max velocity which is reduced to a little under 900 points given the min idle value of 1106 (on a scale from 1000 to 2000).

5.2.3 COMMANDS (COMMANDS.H)

This file is somewhat redundant in its current use (although with future sensors it will become utilized more). It is used to pass wireless commands issued from the control unit to the PID loop where updates to the thrust and euler angles can be incorporated. This is done with one function, executeCommand().

5.2.4 PROPORTIONAL, INTEGRAL, DERIVATIVE (PID) CONTROLLER (PID.H)

This file contains functions to update thrust, yaw, pitch, and roll, and the main PID loop to stabilize the aircraft. Distinct gain terms are defined for proportional, integral, and derivative terms for yaw, pitch, and roll separately (9 values in total). With our 120 Hz PID loop we found that the following values, in measurements of 1/10,000 were adequate to remain stable.

P_yaw = 0; I_yaw = 0; D_yaw = 3500;

P_pitch = 30; I_pitch = 0; D_pitch = 250;

P_roll = 46; I_roll = 0; D_roll = 270;

For the yaw axis we need only have a derivative term, because we do not care about absolute yaw position. In fact, it would be somewhat inconvenient if every time we took off the quadrotor tried to point itself due north. The derivative term here is much higher than the others because changes in this axis are smaller and the response is generated in an axis not directly in line with the propellers; inducing a yaw requires increasing the moment of one direction of the counter-rotating props. For pitch and roll we use both a proportional and a derivative term to force the unit to be as close to 0 degrees pitch and roll as possible unless the user desires a different angle. All integral gains are equal to 0 because it is unrealistic to try to correct for small errors over time when the quadrotor is constantly changing orientation. Integral terms are used to correct for small inaccuracies, and are often low or 0 in embedded PID systems.

5.3 CONTROLLER CODE

Code used to wirelessly control the Wyvern using a MaEvArM and an RF transceiver.

5.3.1 CONTROLLER MAIN CODE (WYVERNCONTROLLER.C)

This file contains similar code to the Wyvern.c file for implementing the RF pin change interrupt, the overflow interrupt, and initializing ADC. In addition, the controller interfaces with a computer via serial to display information to the terminal and take in inputs from the keyboard. This was very helpful when debugging; we were able to change PID gains on the fly and print out Euler angle data. The controller file makes use of 4 external status LEDs and also performs ADC conversions on 4 ports for yaw, pitch, roll, and thrust. These 4 values along with a command byte are sent wirelessly back to the Wyvern.

5.3.2 CONTROLLER (CONTROLLER.H)

This file is dedicated to the serial stream output by the controller to the computer. It has various menus that can be flipped between by the user. Additionally, controller.h allows for the user to view Yaw, Pitch, Roll angles as well as the current PID gains for one of the axes at a time.

6. DEVELOPMENT

Methods and challenges in developing a quadrotor on a limited budget and short timeframe.

6.1 APPROXIMATE SCHEDULE

In order to meet the goals laid out for this project in the given time limit we required a rigid timeline. An estimate of our progress is given below.

Week 1 – Order materials and parts
Week 2 – Finish airframe construction
Week 3 – Modify IMU code and implement UART between IMU and MaEvArM
Week 4 – Open Loop tests of ESCs and motors individually and mounted on the airframe
Week 5 – Initial PID design started
Week 6 – Finish tweaking PID gains and first test flights

6.2 ISSUES ENCOUNTERED

6.2.1 RAZOR IMU BOOTLOADER

After connecting the Razor 9DOF IMU to the Arduino IDE and attempting the upload the code to the board, the following errors were displayed:

avrdude: stk500_getsync(): not in sync: resp=0x72
avrdude: stk500_disable(): protocol error, expect=0x14, resp=0x72

The board was connected correctly to the computer (since we were reading values output from the sample code pre-loaded onto the chip) and the IDE settings were correct. The correct COM port as well as the correct board (Arduino Pro or Pro Mini (3.3V, 8Mhz) w/Atmega328) were selected.

After several hours of searching online and looking at other posts/replies regarding this type of issue (such as resetting the board, updating the FTDI drivers, etc), it was mentioned that it may be possible that either the Arduino Bootloader was the wrong version (since SparkFun just changed these boards from the Atmega168 to the Atmega328) or that even the Bootloader might be corrupted. With these in mind, we tracked down a programmer to try re-burning the Bootloader to the chip**.**

We weren't able to find an ISP programmer (any from the list in the Arduino IDE, such as the AVRISP or AVR ISP mkII), however we were able to find a AVR JTAGICE mkII programmer. This programmer is able to program through either the JTAG or ISP pins, which is great since the only connections available on the board are Serial and ISP/SPI. Since this programmer is incompatible with the Arduino IDE, we had to use AVRStudio4. Here are the steps we took to re-burning the Arduino Bootloader and getting the Arduino IDE to communicate with the chip:

1. Connected the programmer to the computer with the USB cable (and made sure that the computer had the correct drivers for it).
2. Connected the programmer to the board. After soldering pin headers to the 6-pin ISP/SPI connection, the squid breakout cable (which allows for each of the 10 JTAG cables to be connected individually) was connected using the following information: JTAGICE mkII (for the programmer connections) and Razor 9DOF IMU Schematic (for the Razor ISP/SPI connections).
3. In AVRstudio, click "Tools" and then "Program AVR".
4. In the "Main" tab, change the device to "ATmega328P" and the programming mode to "ISP".
5. In ISP settings, we used 500kHz since the board has a 8MHz clock (ISP requires something under 1/4 the system clock for any board running below 12 MHz. 8/4 -> anything under 2MHz). We chose this just to stay on the safe side.
6. Click "Read Signature". We got a long hex string back (in the box under device name). This is a very good sign, since it means that we are getting data back from the board.
7. In the "Program" tab under "Flash", find the bootloader (this hex file is located in the Arduino folder so for us it was under
   arduino-0018\hardware\arduino\bootloaders\atmega).

We used the **ATmegaBOOT_168_atmega328_pro_8MHz.hex** (several posts we found indicated that this was what should be used on this board).

8. Click "Program". The Razor board as well as the Tx and Rx pins on our FTDI board lit up as the data was sent and the new Bootloader was burning onto the chip. We didn't see any errors listed at the bottom of the screen, but we clicked "Verify" to make sure.

6.2.2 IMU DATA CORRUPTION

While receiving data from the IMU over the serial connection, a strange event occurred. Every once in a while (approximately every 20 data transmissions when we printed the data over serial to the computer) a large jump in the data would occur. These "hiccups" produced severe responses. While attempting to work with our PID controls, the quadrotor would be smooth and then violently respond to the data (for instance, the pitch would read 0 and then jump to -14 degrees).

After disabling subsections of our code one-by-one, the problem was found. It was determined that the wireless transmission from the controller was causing our data from the IMU to be sporadically corrupted. These errors in the data were large enough to throw our quadrotor into an unstable state. Our solution to this problem was to disable the wireless receive interrupt (the pin-change interrupt that indicates there is data to be read) while reading the IMU data over serial. This allowed the data to be loaded in its entirety before the wireless was read, thus preventing corruption in our IMU data.

6.2.3 CODE EFFICIENCY

After we completed our first full version of code we spent several days in the lab trying to fine-tune our PID terms. After a lot of frustration, we decided to take a look at the quadrotors in Penn's GRASP lab as a comparison. After feeling their PID response, it was painfully obvious that our response time was not nearly as fast as we thought it was. The GRASP lab units run their full PID loop at 1000 Hz. We were running our PID loop somewhere around 40 Hz, and, being a little naive, we thought this would be adequate.

In order to increase our code efficiency we did several things. First, we increased the UART baud rate from the IMU to the MaEvArM from 38,400 to 250,000. This allowed us to read in angle and gyro values faster so the PID functions didn't hang up waiting for information. Second, we unrolled some of the UART *while* loops. Alternatively, we could have specified a more optimized compile flag. This had an amazing effect, changing our 40 Hz response to 120 Hz. Of course, this was not nearly as fast as the response in the GRASP lab, but, having read reports of successful PID loops as low as 70 Hz, we thought we'd give the new code a try.

The difference was incredibly clear; the derivative terms were much more responsive, and our morale was much improved after fruitlessly toying at PID gains for so many days.

After this, we constantly checked the frequency of our PID loop after adding any new considerable block of code. Also, the bottleneck for increasing our code speed was determined to be the output of the IMU. Running the DCM and correction algorithms produced a max speed of 120 Hz. One possible method to increase this (without changes in code) would be to replace the 8 MHz oscillator on the Razor IMU with a 16 MHz or even a 20 MHz oscillator. The ATmega328P is able to handle these clock speeds, and increasing to them would result in large improvements in the data output speed.

6.2.4 IMU DATA ERRORS

After almost two weeks of attempting to tune our PID controls, we found an odd occurrence. While at low RPMs, the IMU would output correct data for both Euler angles and rotational velocities. However, when we increased the speed of the motors to see how the PID controls reacted with higher RPMs, the IMU began to sense that it was a consistent 3 degrees from horizontal while it was actually held flat. In addition to this offset, the data for the Euler angles (Pitch and Roll) as well as their respective angular velocities would vary greatly (about +/- 20 degrees for Euler angles and about +/- 200 degrees/sec for the angular velocities).

This was far from usable. These results would cause the quadrotor to move quickly in one direction without correction while also becoming wildly unstable. Hoping that it was a purely mechanical problem since the code on the IMU (Open-Source Razor AHRS code) uses high-level Direction Cosine Matrix (DCM) and normalization algorithms that are already tuned to the IMU characteristics, we started with two possible culprits: vibrations and electro-magnetic interference (EMI).

Our first idea was to try reducing the vibrations that the IMU experienced while the motors were running. The screws that were holding the IMU in place on top of nylon spacers were removed and small pieces of foam tape were placed in-between to act as dampening washers. This helped a little, but the noise in the IMU data was still too much.

Our next idea was to replace the screws holding the IMU onto the nylon spacers with rubber bands placed in an "X" with the foam tape pieces in between. This helped even more, reducing the errors we were seeing in the angular velocity as well as decrease the magnitude of the noise in the Euler angles. However, we were still seeing the 3 degree offset at higher RPMs.

After trying a couple vibration-reduction methods with small improvements in the IMU output, the possibility of EMI was addressed. Since the IMU was being placed close to the outputs of the ESCs (which had cables that changed direction quickly and carried high currents), it was thought that this could be causing some error in the IMU when the motors were run at higher currents/speeds. Although moving a magnet near the IMU did not seem to have any effect, we tried this solution anyway. Using conductive tape, each of the wire groups output from the ESCs were covered and then grounded. This produced no visible effects in our tests; Noise and offset were still present.

Next, the idea that the rotors were causing too much vibration was addressed. When we visited the GRASP laboratory, we were able to inspect their quadrotors (which they purchased) and see them in action. One noticeable difference we saw was that the motors on the GRASP lab quadrotors cause little to almost no vibration. With this in mind we attempted to remount our rotors to try and balance them. This was not very effective with the tri-blade rotors for some reason (possibly since they are of lesser quality). When we switched to the dual-blade rotors, we saw a significant improvement in the amount of vibration. In addition, the noise produced by the rotors greatly decreased. We were surprised at this, since when we did our research we read that the tri-blade props were supposed to be quieter and produce fewer vibrations. Although this change did result in less noise, the errors in the data output from the IMU were still enough to prevent our controls from working effectively at speeds high enough to fly.

Our last idea before we attempted a software correction for this data error was to move the IMU to a new position and use a better dampening material. The IMU was swapped with the MaEvArM, which placed it directly onto the frame instead of on nylon spacers. To absorb vibrations, a square of open-cell foam was placed in between the frame and the IMU. The

MaEvArM was then mounted on top of the nylon spacers (this actually helped a lot since the Load/Run swich and reset button on the MaEvArM were difficult to reach before). When the motors were spun up to speed, the IMU output was almost perfect (although there were small errors, these were less than a degree and would not prevent us from flying). The nylon spacers were believed to be amplifying the vibrations since they were not completely rigid (as they were able to flex a small amount).

6.2.5 BRITTLE PROPS

After switching to the two-blade props, the improvement in noise and stability was such that we could not go back to our three-blade props. However, the stiffness of the two-blade props that helped with noise caused them to break very easily (sometimes on landings that tipped just enough allow the props to touch a hard floor). We saw no easy solution to this beside constructing a protective ring around all four rotors. This is a step that will likely be taken on future iterations.

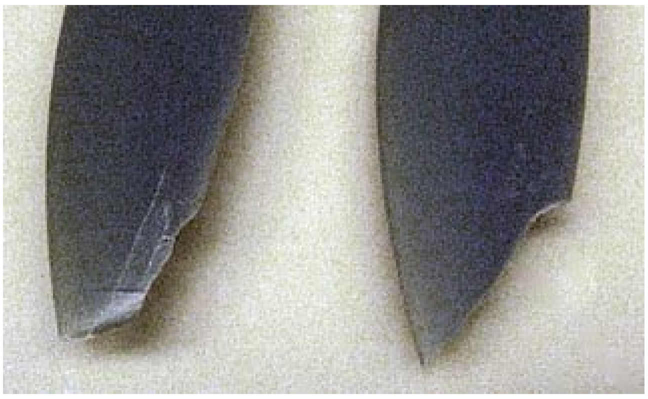

FIGURE 21: CHIPPED PROPS

7. CONCLUSION

This project was very ambitious, especially given the limited time frame in which we were working. Our main goal was merely to have our quadrotor hover, but even this was far from trivial. Our original hope was that we would have time to make this quadrotor autonomous. This was not very reasonable without any real means of localizing our aircraft. Nevertheless, we succeeded in creating a quadrotor from scratch that hovered reasonably in a fairly stable orientation for a long period of time. We have not yet hovered for a complete battery cycle, but our calculations show that a full flight should be able to last up to 12 minutes.

This project demonstrates that a full quadrotor system can be built in such a way that is manufacturable, expandable, and inexpensive. The entire design from bottom to top took a mere 6 weeks and approximately $400 (see Appendix A).

It was suggested to us during the initial design of our system that we try to achieve hovering at around 1/4 to 1/3 of our maximum thrust. This was achieved easily with our light airframe; the entire thrust is limited to about 35% to prevent the user from accelerating to fast, and the unit hovers at around 30% of the maximum thrust. We have not tried accelerating at maximum thrust (doing so would greatly limit the capability of our PID stabilization), but we are eager to push our system to its limits so that we can discover new areas to improve for the next iteration.

This project has taught us a lot about what needs to be considered when designing a UAV, and it has also given us insight into things that should be avoided. Things that we would change include:

- Though our hollow aluminum and ABS frame served us well for this first iteration, it was apparent that the liftime of this frame is not very long. After only a few flights (and some pretty rough crashes) a couple of the arms became twisted and needed to be straightened out before each flight.
- Our IMU from SparkFun was inexpensive, well packaged, and convenient to use. Unfortunately, the ICs used on the board were very susceptible to noise. A second iteration would likely involve designing our own breakout board with less noisy ICs.
- This iteration did not include an altimeter because our barometer ordered from SparkFun did not function properly. Future iterations would include either a ping sensor or barometer in order to determine altitude and perform a PID loop on that information.
- Our wireless boards performed very respectably and any error created by dropped packets was not noticeable. However, we may switch to a more robust wireless technology like Zigbee.
- We currently have no method of localization. This must be implemented in future iterations if we are to have any sort of swarming behavior and communication between multiple units. This will be helped in part by the addition of an altimeter, but a robust means of position localization (motion capture or GPS) will also be looked into.

APPENDIX A: PRICING

Item	Quanity	Supplier	Price per Unit ($)	Total Price ($)
30 A ESC (motor controllers)	4	Dino Direct	14.99	59.96
Brushless Motors	4	Dino Direct	11.99	47.96
9DOF IMU	1	SparkFun	125.00	125.00
Microcontroller	2	(Approximate Replacement) AdaFruit	20.00	40.00
Battery	1	HobbyPartz	16.95	16.95
2-Blade Props	2	RCToys	7.59	15.18
Bullet Connectors	4	RCDude	2.49	9.96
Deans Connector	1	ToySonics	6.5	6.5
Deans Connector Couplers	1	RCDude	3.95	3.95
Aluminum Rods	3	McMaster	6.03	18.09
ABS Plastic	1	McMaster	25.09	25.09
RF Boards	2	SparkFun	19.95	39.90
Joysticks	2	SparkFun	3.95	7.90
Joystick Breakout Board	2	SparkFun	1.95	3.90
			Project Total	**$420.34**

TABLE 4: WYVERN PROJECT PRICING

APPENDIX B: SCHEMATICS

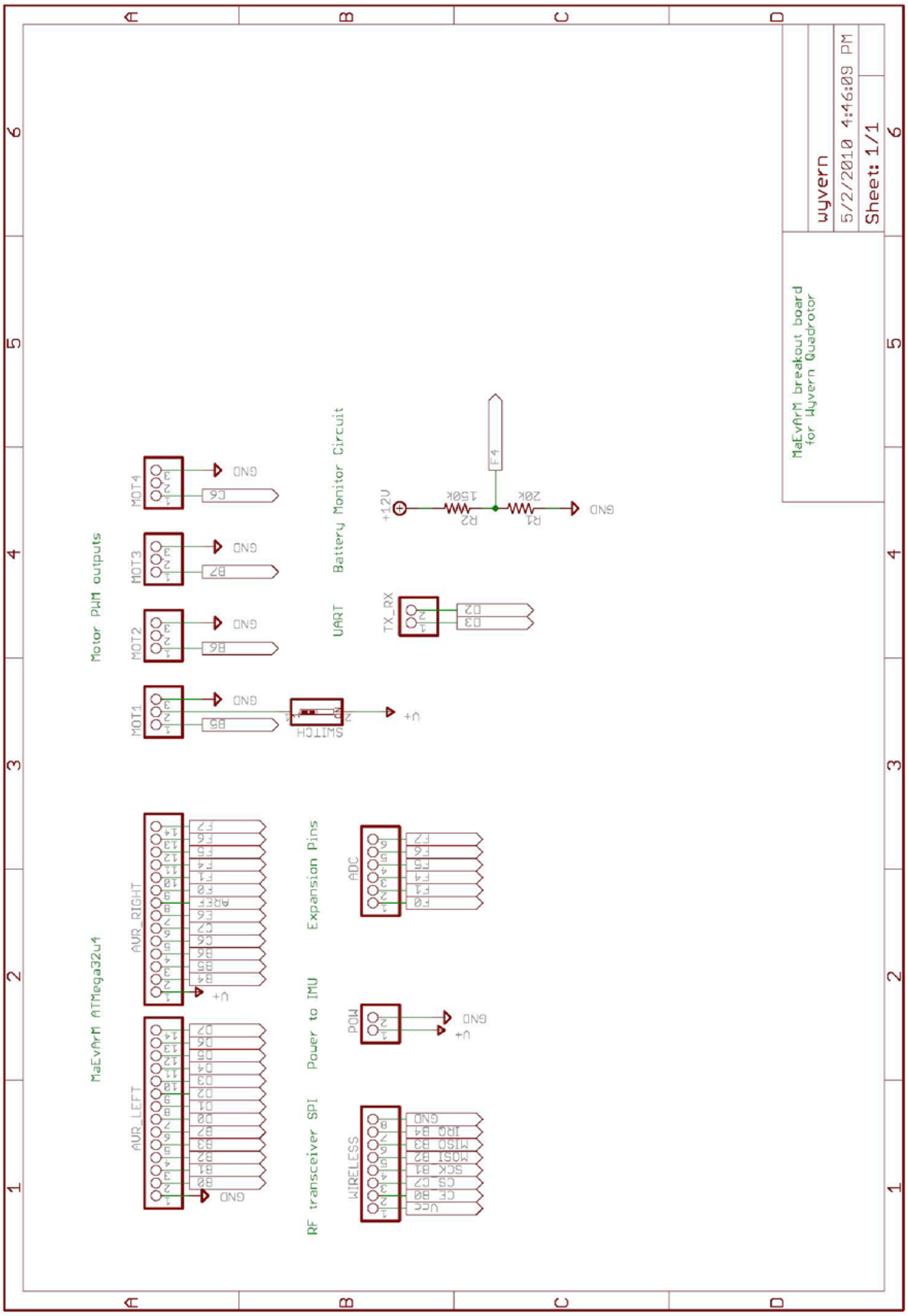

FIGURE 22: WYVERN MICROCONTROLLER BREAKOUT SCHEMATIC

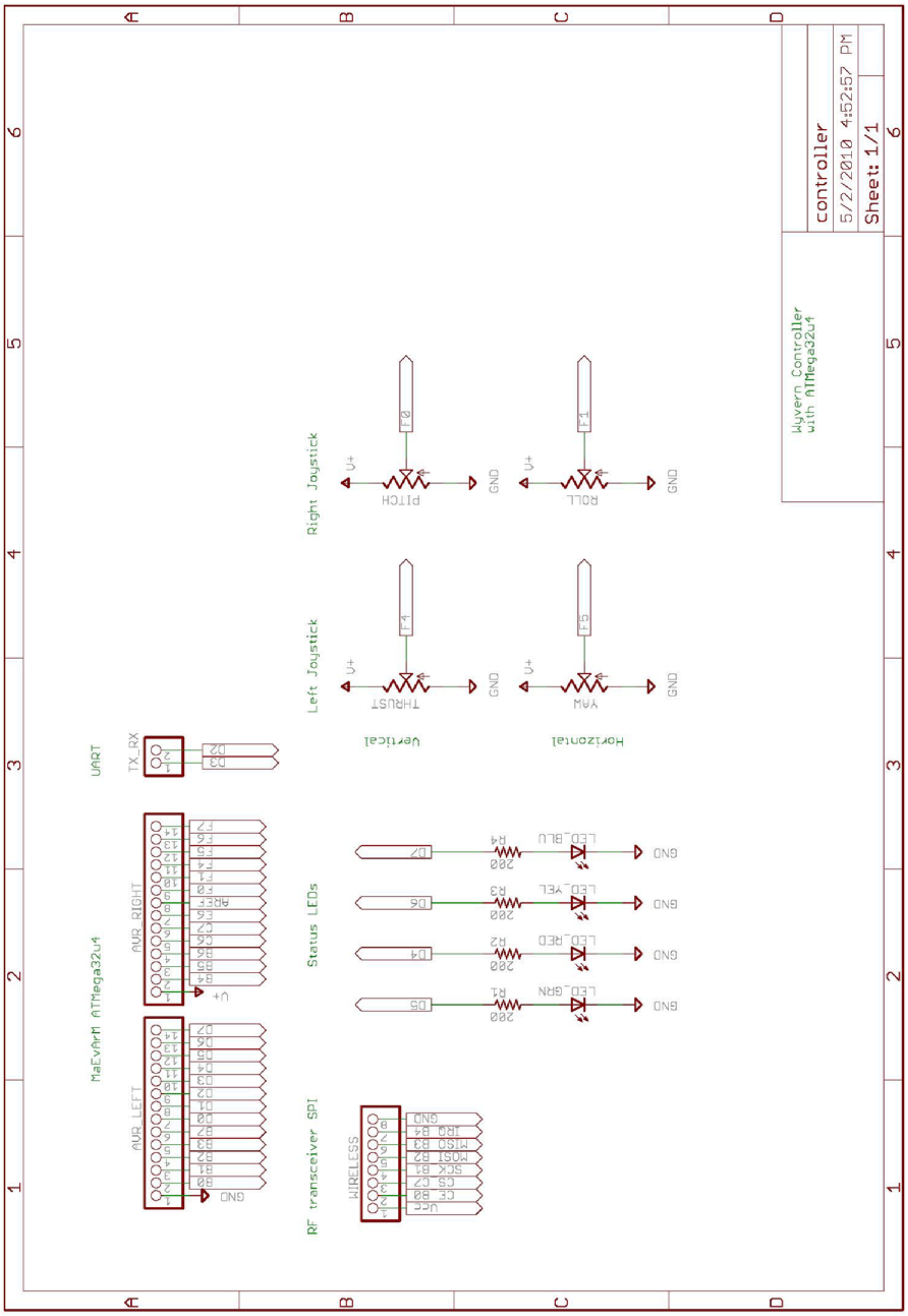

FIGURE 23: WYVERN CONTROLLER SCHEMATIC

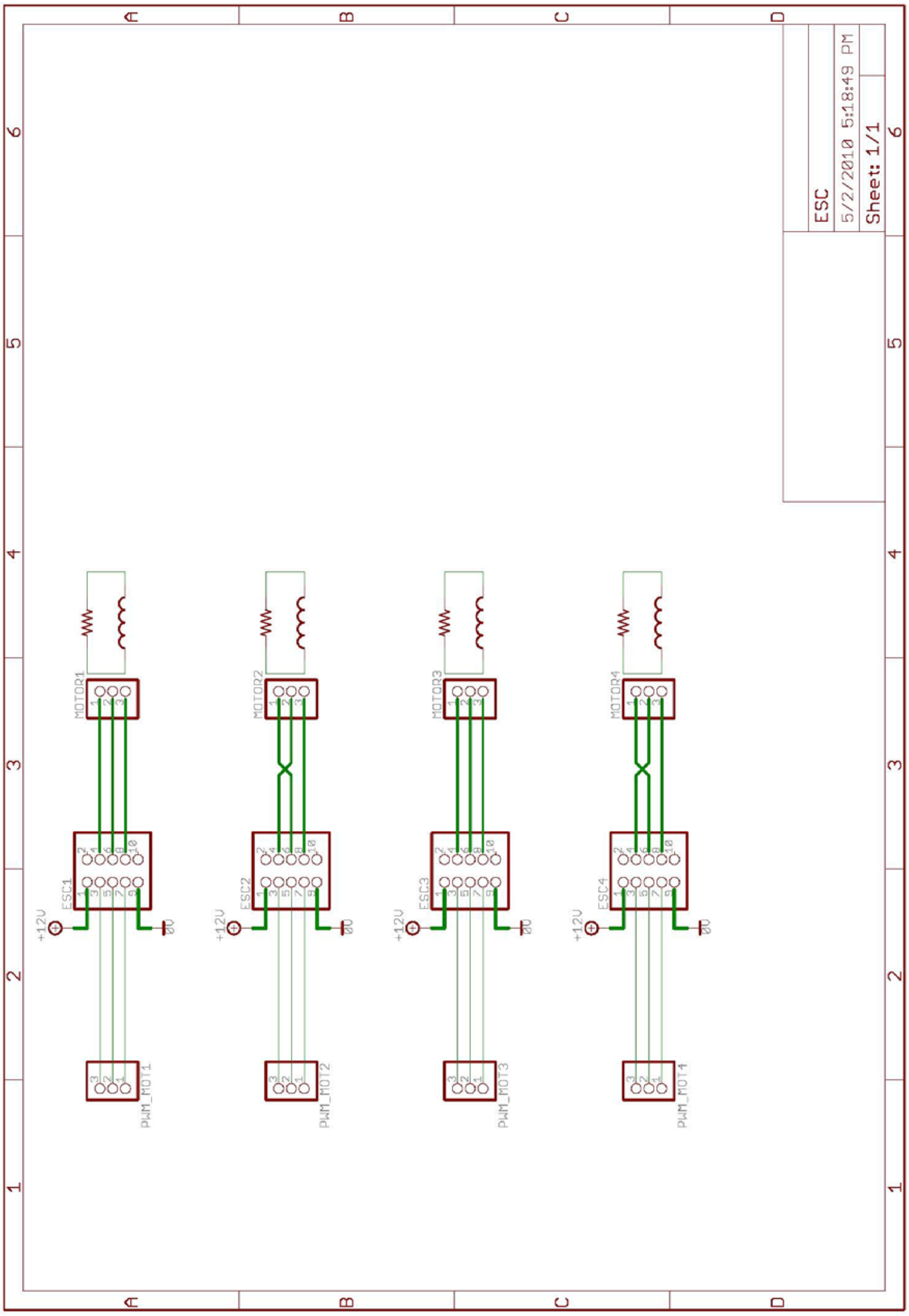

FIGURE 24: WYVERN ESC SCHEMATIC

APPENDIX C: CODE

COMMON CODE

MICROCONTROLLER HEADER FILE (WYVERN.H)

```
// -------------------------------------------------
// Wyvern Quadrotor
// Custom Header File
// version: 1.0.7
// date: April 29, 2010
// authors: William Etter, Paul Martin, Uriah Baalke
// -------------------------------------------------

/* HEADER FILE INFORMATION
      -Include files
      -Register operations
      -Basic LED operations
      -Register name definitions
      -Communication structs
      -Functions:
            max()
            clockSet()
            init_maevarm()
            disableJTAG()
*/

// ============================================================

#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

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

// operations to set, clear, toggle, and check individual register bits
// clear_flag sets the register to 1 (uses =, not &= or |= )
#define set(reg,bit)        reg |= (1<<(bit))
#define clear(reg,bit)      reg &= ~(1<<(bit))
#define toggle(reg,bit)     reg ^= (1<<(bit))
#define check(reg,bit)      (reg & (1<<(bit)))
#define clear_flag(reg,bit)     reg=(1<<(bit))

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

// Basic LED Functions
// Red and Green UC LEDs
#define LED_ucgreen_on() clear(PORTE,2)
#define LED_ucgreen_off()set(PORTE,2)
#define LED_ucgreen_toggle()    toggle(PORTE,2)
#define LED_ucred_on()          clear(PORTE,6)
#define LED_ucred_off()         set(PORTE,6)
#define LED_ucred_toggle()      toggle(PORTE,6)

// Red, Green, Yellow, and Blue LEDs
#define LED_red_on()            clear(PORTD,4)
```

```
#define LED_red_off()           set(PORTD,4)
#define LED_red_toggle() toggle(PORTD,4)
#define LED_green_on()          clear(PORTD,5)
#define LED_green_off()         set(PORTD,5)
#define LED_green_toggle()      toggle(PORTD,5)
#define LED_yellow_on()         clear(PORTD,6)
#define LED_yellow_off() set(PORTD,6)
#define LED_yellow_toggle()     toggle(PORTD,6)
#define LED_blue_on()           clear(PORTD,7)
#define LED_blue_off()          set(PORTD,7)
#define LED_blue_toggle()toggle(PORTD,7)

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

// Set up Atmega32U4 specific register names
// (In case non-32U4 code is used)
#define UBRRH UBRR1H
#define UBRRL UBRR1L
#define UDR UDR1
#define USBS USBS1
#define UCSRA UCSR1A
#define UDRE UDRE1
#define RXC RXC1
#define UCR UCSR1B
#define UCSRB UCSR1B
#define RXEN RXEN1
#define TXEN TXEN1
#define RXCIE RXCIE1
#define UDRIE UDRIE1
#define UCSRC UCSR1C
#define UCSZ0 UCSZ10
#define UCSZ1 UCSZ11
#define UCSRC_SELECT 0

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

#ifndef max
      #define max( a, b ) ( ((a) > (b)) ? (a) : (b) )
#endif

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

// packet struct
typedef struct{
      char command;
      int pitch;
      int roll;
      int yaw;
      int throttle;
}packet_com_t;

typedef struct{
      uint16_t yaw;
      uint16_t pitch;
      uint16_t roll;
      uint16_t omegazero;
      uint16_t omegaone;
      uint16_t omegatwo;
}packet_razordata_t;

typedef struct{
	int yaw;
	int pitch;
	int roll;
	int altitude;
	int battery;
	int p;
	int i;
	int d;
	int pidband;
	char stat0;
	char stat1;
	char stat2;
	char stat3;
	char stat4;
}packet_inf_t;

/*************************************************************************
Function: clockSet()
Purpose:  Sets clock to 8MHz
Input:    None
Returns:  None
*************************************************************************/
void clockSet(){
	CLKPR = (1<<CLKPCE);
	CLKPR = 0;
}


#define F_CPU 8000000UL

/*************************************************************************
Function: disableJTAG()
Purpose:  Allows access to F4-F7 as normal port pins
	       to disable the JTAG system, set the JTD bit in MCUCR twice
within 4 clock cycles
	       |= is too slow, so we're going to write the entire register
	       fortunately, all the other bits in MCUCR should be 0, so this is
fine
Input:    None
Returns:  None
*************************************************************************/
void disableJTAG(){
	MCUCR = (1 << JTD);
	MCUCR = (1 << JTD);
}

/*************************************************************************
Function: init_uc()
Purpose:  Sets up the microcontroller board
		  -Sets system clock to 8MHz
		  -Sets LED Ports (LEDs OFF)
Input:    None
Returns:  None
*************************************************************************/
void init_uc(){
	// Set system clock to 8MHzs
	clockSet();

	// Set up Red and Green UC LED Ports
	set(DDRE,2);
	set(DDRE,6);
	set(PORTE,2);
	set(PORTE,6);

	// Set up Red, Green, Yellow, and Blue LEDs
	set(DDRD,4);
	set(DDRD,5);
	set(DDRD,6);
	set(DDRD,7);
	set(PORTD,4);
	set(PORTD,5);
	set(PORTD,6);
	set(PORTD,7);

	// Enable Pins F4-F7
	disableJTAG();
}
```

ANALOG TO DIGITAL CONVERSION(ADC) (ADC.H)

```
// --------------------------------------------------
// Wyvern Quadrotor
// ADC
// version: 1.0.1
// date: April 29, 2010
// authors: William Etter, Paul Martin, Uriah Baalke
// --------------------------------------------------

// CONSTANT VARIABLES

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

// GLOBAL VARIABLES

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

// FUNCTION HEADERS
void init_adc(void);
uint16_t get_adc(unsigned int ADC_Channel);

// ===========================================================

/**************************************************************************
Function: init_adc()
Purpose:  Sets up the ACD
Input:    None
Returns:  None
**************************************************************************/
void init_adc(void){
	// ADC Multiplexer Selection Register
	// Use Vcc, right justified
	ADMUX |= (0<<REFS1)|(1<<REFS0)|(0<<ADLAR);

	// Digital Input Disable Register 0
	// Used to disable digital input on ADC Channels
	// Channels 0,1,4,5,6,7 (1=disabled)
	// DIDR0 = [ADC7D,ADC6D,ADC5D,ADC4D, - , - , ADC1D, ADC0D]
	DIDR0 |=0xF3;

	// ADC Control and Status Register A
	// Turn on ADC,prescale ADC clock by 64 (8MHz/64 = 125kHz)
	ADCSRA |=(1<<ADEN)|(1<<ADPS2)|(1<<ADPS1)|(0<<ADPS0);
```

```
}

/*************************************************************************
Function:    get_adc()
Purpose:     Read a value from one of the 6 ADC values
Input:       ADC Channel to obtain value on (0,1,4,5,6,7)
Returns:     ADC Result
*************************************************************************/
uint16_t get_adc(unsigned int ADC_Channel){
	// Channel set with 	ADMUX => (MUX4,MUX3,MUX2,MUX1,MUX0)

	// Clear ADMUX Channel Select
	ADMUX &=0xE0;

	switch(ADC_Channel){
		case 0:		// Channel 0 (00000)
			ADMUX |=0x00;
			break;
		case 1:			// Channel 1 (00001)
			ADMUX |=0x01;
			break;
		case 4:			// Channel 4 (00100)
			ADMUX |=0x04;
			break;
		case 5:			// Channel 5 (00101)
			ADMUX |=0x05;
			break;
		case 6:			// Channel 6 (00110)
			ADMUX |=0x06;
			break;
		case 7:			// Channel 7 (00111)
			ADMUX |=0x07;
			break;
		default:
			break;
	}

	// Start converstion
	set(ADCSRA,ADSC);

	// Wait until complete
	while(!(check(ADCSRA,ADIF))){
	}

	// Clear Flag
	set(ADCSRA,ADIF);
	// Obtain value
	return ADC;
}

// CURRENTLY IN DEVELOPEMENT
/*************************************************************************
Function:    get_ucTemp()
Purpose:     Get the (rough) UC temperature from the on-board sensor
Input:
Returns:     ADC Temperature Sensor Result
*************************************************************************/
uint16_t get_ucTemp(){
	// Save previous ADMUX settings
	int oldADMUX = ADMUX;
	// Clear ADMUX Channel Select
	ADMUX &= 0xE0;

	// Set voltage refernce to internal 2.56V
	ADMUX |= 0xC0;

	// On-chip Temperature Sensor (11111)
	ADMUX |= 0x1F;

	// Start converstion
	set(ADCSRA,ADSC);

	// Wait until complete
	while(!(check(ADCSRA,ADIF))){
	}

	// Clear Flag
	set(ADCSRA,ADIF);

	// Read a second time
	// Start converstion
	set(ADCSRA,ADSC);

	// Wait until complete
	while(!(check(ADCSRA,ADIF))){
	}

	// Clear Flag
	set(ADCSRA,ADIF);

	// Return old ADMUX settings
	ADMUX = oldADMUX;

	// Obtain value
	return ADC;
}
```

UART (UART.H)

```
// -----------------------------------------------------
// WyvernMaEvArM microcontroller board
// UART
// version: 1.0.3
// date: April 29, 2010
// authors: William Etter, Paul Martin, Uriah Baalke
// -----------------------------------------------------

// STATIC VARIABLES
#define UART_RX_BUFFER_SIZE 512 // 2,4,8,16,32,64,128 or 256 bytes
#define UART_TX_BUFFER_SIZE 512 // 2,4,8,16,32,64,128 or 256 bytes
#define UART_RX_BUFFER_MASK ( UART_RX_BUFFER_SIZE - 1 )
#if ( UART_RX_BUFFER_SIZE & UART_RX_BUFFER_MASK )
	#error RX buffer size is not a power of 2
#endif
#define UART_TX_BUFFER_MASK ( UART_TX_BUFFER_SIZE - 1 )
#if ( UART_TX_BUFFER_SIZE & UART_TX_BUFFER_MASK )
	#error TX buffer size is not a power of 2
#endif
// Baudrate - Set Baud Rate given U2X1 = 0
// 38,400 -> 0x0c 		250,000 -> 0x01;
unsigned int baud = 0x01;
```

```
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

// GLOBAL VARIABLES
static unsigned char UART_RxBuf[UART_RX_BUFFER_SIZE];
static volatile unsigned char UART_RxHead;
static volatile unsigned char UART_RxTail;
static unsigned char UART_TxBuf[UART_TX_BUFFER_SIZE];
static volatile unsigned char UART_TxHead;
static volatile unsigned char UART_TxTail;

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

// FUNCTION HEADERS
void init_uart(void);
unsigned char ReceiveByte(void);
void TransmitByte(unsigned char data);
void TransmitString(const char *s );
void TransmitInt(int i );
unsigned char DataInReceiveBuffer(void);
int uart_available(void);
void uart_flush(void);

// ============================================================

/**************************************************************************
Function: init_uart()
Purpose:  Sets up UART communication
               8-bit transmission, no parity, 1-stop bit, U2X1 = 0
Input:
Returns:
**************************************************************************/
void init_uart(void)
{
      //UCSR1A |= 0x02;  // Uncomment to set U2X1 = 1
      // Set baudrate
      UBRR1L = baud;
      // Enable Rx and Tx
      UCSR1B |=(1<<RXEN1)|(1<<TXEN1);
      // Set to 8 bit tranmission, 1 stop bit
      UCSR1C |=(1<<UMSEL11)|(1<<UCSZ10)|(1<<UCSZ11);
      // Enable USART Receive Complete Interrupt
      UCSR1B |=(1<<RXCIE1);
      // Flush receive buffer
      UART_RxTail = 0;
      UART_RxHead = 0;
      UART_TxTail = 0;
      UART_TxHead = 0;
}

/**************************************************************************
Function: ISR(USART1_RX_vect)
Purpose:  RXC (Receive Complete) Interrupt
               Interrupt Vector 26
Input:
Returns:
**************************************************************************/
ISR(USART1_RX_vect)
{
      unsigned char data;
      unsigned char tmphead;
```

```
	// Read the received data
	data = UDR1;
	/// Calculate buffer index
	tmphead = ( UART_RxHead + 1 ) & UART_RX_BUFFER_MASK;
	// Store new index
	UART_RxHead = tmphead;
	if ( tmphead == UART_RxTail )
	{
		//ERROR! Receive buffer overflow
		uart_flush();
	}
	// Store received data in buffer
	UART_RxBuf[tmphead] = data;
}

/**************************************************************************
Function: ISR(USART1_UDRE_vect)
Purpose:  UDRE (USART Data Register Empty) Interrupt
              Interrupt Vector 27
Input:
Returns:
**************************************************************************/
ISR(USART1_UDRE_vect)
{
	unsigned char tmptail;
	// Check if all data is transmitted
	if ( UART_TxHead != UART_TxTail )
	{
		// Calculate buffer index
		tmptail = ( UART_TxTail + 1 ) & UART_TX_BUFFER_MASK;
		// Store new index
		UART_TxTail = tmptail;
		// Start transmition
		UDR1 = UART_TxBuf[tmptail];
	}
	else
	{
		// Disable UDRE interrupt
		UCSR1B &= ~(1<<UDRIE);
	}
}

/**************************************************************************
Function: ReceiveByte()
Purpose:  Returns the next byte in the Rx buffer
              Waits for incoming data if nothing in buffer
Input:
Returns:  Next char in Rx buffer
**************************************************************************/
unsigned char ReceiveByte(void)
{
	unsigned char tmptail;
	// Wait for incoming data
	while ( UART_RxHead == UART_RxTail );
	// Calculate buffer index
	tmptail = ( UART_RxTail + 1 ) & UART_RX_BUFFER_MASK;
	// Store new index
	UART_RxTail = tmptail;
	 // Return data
	return UART_RxBuf[tmptail];
}
```

```
/***********************************************************************
Function: TransmitByte()
Purpose:  Places data into Tx buffer and enable UDRE interrupt to transmit
               Waits for free space if no room in buffer
Input:    Data to transmit
Returns:
***********************************************************************/
void TransmitByte(unsigned char data)
{
      unsigned char tmphead;
      tmphead=0;
      // Calculate buffer index
      tmphead = ( UART_TxHead + 1 ) & UART_TX_BUFFER_MASK;
      // Wait for free space in buffer
      while ( tmphead == UART_TxTail );
      // Store data in buffer
      UART_TxBuf[tmphead] = data;
      // Store new index
      UART_TxHead = tmphead;
      // Enable UDRE interrupt
      UCSR1B |= (1<<UDRIE1);
}

/***********************************************************************
Function: TransmitString()
Purpose:  Transmit string to UART
Input:    String to be transmitted
Returns:
***********************************************************************/
void TransmitString(const char *s )
{
    while (*s)
      TransmitByte(*s++);
}

/***********************************************************************
Function: TransmitInt()
Purpose:  Transmit integer to UART
Input:    Integer to be transmitted
Returns:
***********************************************************************/
void TransmitInt(int i )
{
      char s[8];
      itoa(i,s,10);
      TransmitString(s);
}

/***********************************************************************
Function: DataInReceiveBuffer()
Purpose:  Determine if there is data in the Rx buffer
Input:
Returns:  Return 0 (FALSE) if Rx buffer is empty
***********************************************************************/
unsigned char DataInReceiveBuffer(void)
{
      return ( UART_RxHead != UART_RxTail );
}

/***********************************************************************
```

```
Function: uart_available()
Purpose:  Determine the number of bytes waiting in the receive buffer
Input:
Returns:  Integer number of bytes in the receive buffer
**************************************************************************/
int uart_available(void)
{
      return   (UART_RX_BUFFER_MASK   +   UART_RxHead   -   UART_RxTail)   %
UART_RX_BUFFER_MASK;
}

/*************************************************************************
Function: uart_flush()
Purpose:  Flush bytes waiting in the receive buffer.  (Actually ignores them)
Input:
Returns:
**************************************************************************/
void uart_flush(void)
{
      UART_RxHead = UART_RxTail;
}
```

RF COMMUNICATION (WYVERN-RF.H)

```
// --------------------------------------------------
// Wyvern controller board
// RF
// version: 1.0.4
// date: April 29, 2010
// authors: William Etter, Paul Martin, Uriah Baalke
// --------------------------------------------------

/*
Wyvern RF header for MaEvArM
Based on code by Joe Romano and J. Fiene

Provides the following functions:
 to set the packet size, define PACKET_SIZE before including - must be less
than 32
 ( #define PACKET_SIZE 17

      void RFsetup(char * recvAddr)
            // initialize the wireless node and start receiving
            - recvAddr is a pointer to a 5-element char array

      void RFtransmit( char* txDat, char* destAddr)
            // suspend receive mode and transmit a char array to a specific
wireless node
            // will only send one packet, and does not check for receipt
            - txDat is a pointer to an n-element char array
            - destAddr is a pointer to a 5-element char array

      int RFtransmitUntil( char* txDat, char* destAddr, int txTries)
            // suspend receive mode and transmit a char array to a specific
wireless node
            // if no acknowledgement is returned from the other node, it will
retransmit up to txTries times
            // returns 0 if unsuccessful, or the number of tries it took
before success
            - txDat is a pointer to an n-element char array
            - destAddr is a pointer to a 5-element char array
            - txTries is the maximum number of retransmit attempts

      int RFRXdataReady()
            // check to see if a message has been received
            - returns 1 if data, 0 if empty

      void RFreceive( char* retDat)
            // read the latest data from receive buffer
            - retDat - pointer to an n-element char array storage container
*/

// make sure we don't include this file twice
#ifndef _RF24L01_HELPER
#define _RF24L01_HELPER

//basic WINAVR includes
#include <avr/io.h>
#include <util/delay.h>

/* Memory Map */
#define CONFIG      0x00
#define EN_AA       0x01
#define EN_RXADDR   0x02
#define SETUP_AW    0x03
#define SETUP_RETR  0x04
#define RF_CH       0x05
#define RF_SETUP    0x06
#define STATUS      0x07
#define OBSERVE_TX  0x08
#define CD          0x09
#define RX_ADDR_P0  0x0A
#define RX_ADDR_P1  0x0B
#define RX_ADDR_P2  0x0C
#define RX_ADDR_P3  0x0D
#define RX_ADDR_P4  0x0E
#define RX_ADDR_P5  0x0F
#define TX_ADDR     0x10
#define RX_PW_P0    0x11
#define RX_PW_P1    0x12
#define RX_PW_P2    0x13
#define RX_PW_P3    0x14
#define RX_PW_P4    0x15
#define RX_PW_P5    0x16
#define FIFO_STATUS 0x17

/* Bit Mnemonics */
#define MASK_RX_DR  6
#define MASK_TX_DS  5
#define MASK_MAX_RT 4
#define EN_CRC      3
#define CRCO        2
#define PWR_UP      1
#define PRIM_RX     0
#define ENAA_P5     5
#define ENAA_P4     4
#define ENAA_P3     3
#define ENAA_P2     2
#define ENAA_P1     1
#define ENAA_P0     0
#define ERX_P5      5
#define ERX_P4      4
```

```
#define ERX_P3      3
#define ERX_P2      2
#define ERX_P1      1
#define ERX_P0      0
#define AW          0
#define ARD         4
#define ARC         0
#define PLL_LOCK    4
#define RF_DR       3
#define RF_PWR      1
#define LNA_HCURR   0
#define RX_DR       6
#define TX_DS       5
#define MAX_RT      4
#define RX_P_NO     1
#define TX_FULL     0
#define PLOS_CNT    4
#define ARC_CNT     0
#define TX_REUSE    6
#define FIFO_FULL   5
#define TX_EMPTY    4
#define RX_FULL     1
#define RX_EMPTY    0

/* Instruction Mnemonics */
#define R_REGISTER    0x00
#define W_REGISTER    0x20
#define REGISTER_MASK 0x1F
#define R_RX_PAYLOAD  0x61
#define W_TX_PAYLOAD  0xA0
#define FLUSH_TX      0xE1
#define FLUSH_RX      0xE2
#define REUSE_TX_PL   0xE3
#define NOP           0xFF

// Pins that the 24L01 connects to the MEAMAVR board on
#define P_RF_CS                 PORTC7
#define PORT_CS                 PORTC
#define      DDR_P_RF_CS        DDC7
#define DDR_RF_CS         DDRC

#define P_RF_CE                 PORTB0
#define      PORT_CE                  PORTB
#define DDR_P_RF_CE             DDB0
#define      DDR_RF_CE          DDRB

// Pins that are used for SPI on the MEAMAVR board
#define PORT_SPI    PORTB      // the port that corresponds to the SPI pins of
our 32UF AVR chip
#define DDR_SPI     DDRB       //  the  data  register  that  corresponds  to  the
SPI of our 32UF AVR chip
#define DD_MISO     DDB3       // the data register that corresponds with the
MISO SPI pin of the 32UF AVR chip
#define DD_MOSI     DDB2       // the data register that corresponds with the
MOSI SPI pin of the 32UF AVR chip
#define DD_SCK      DDB1       // the data register that corresponds with the
SCK SPI pin of the 32UF AVR chip
#define DD_SS       DDB0       // the data register that corresponds with the
SS SPI pin of the 32UF AVR chip

// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

```
char PACKET_SIZE;
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

// Config Options - set all options to default values
#define CONFIG_VAL (1<<EN_CRC) | (1<<MASK_TX_DS) | (1<<MASK_MAX_RT)

// set up the spi port in master mode with the polarity options (etc.) that the
24L01 requires. Also set DDR of the CE pin of rf module.
void SPIsetup(){
      //Make sure Power Save didn't turn SPI off
      PRR0 &= ~(1 << PRSPI);

      // Set MOSI and SCK and SS output, all others input. SS MUST be config.
to output
      // Also configure the CE pin of the rf module as an output
      DDR_SPI |= (1 << DD_MOSI) | (1<<DD_SCK) | (1<<DD_SS);

      // Enable SPI, Master, set clock rate sys_clk/128
      SPCR = (1<<SPE) | (1<<MSTR) | (1<<SPR1) | (1<<SPR0);

      //PAUSE to let the wireless chip go through initialization
      _delay_ms(200);
}

/*************************************************************************
Function: RFwriteReg()
Purpose:  Writes a data register on the 24L01 RF chip
Input:    writeReg - the register we want to write data to
               writeDat - the storage container for the data we plan on writing
               numBytes - the number of bytes to write. !!
                    The writeDat storage container must be at least this size in
bytes (char's)
Returns:  None
*************************************************************************/
void RFwriteReg(char writeReg, char* writeDat, int numBytes){

      // turn the CS channel low to begin transmission
      PORT_CS &= ~(1<<P_RF_CS);

      // send 1 byte out the spi port to request write of writeReg
      // in order to do this we add 0x20 to our write register as per the
format of the 23L01 datasheet
      SPDR = W_REGISTER + writeReg;

      // wait for transmission to finish
      while(!(SPSR & (1<<SPIF) ) );

      // write the number of bytes we want from writeDat
      int i;
      for(i = 0; i < numBytes; i++)
      {
            // send 1 byte out the spi port. this is the data we want to write
to the RF chip
            SPDR = writeDat[i];

            // wait for transmission to finish
            while(!(SPSR & (1<<SPIF) ) );
      }

      // turn the CS channel high to end transmission
      PORT_CS |= (1<<P_RF_CS);
```

```
}




// function to fill the transfer (TX) buffer with data we wish to transmit
wirelessly
// txDat - an appropriately sized storage container containing the data we plan
on pushing into the TX register
// numBytes - the number of bytes to push in back. !! The txDat storage
container must be at least this size in bytes (char's)
void RFfillTransferBuffer( char* txDat, int numBytes ){
	// turn the CS channel low to begin transmission
	PORT_CS &= ~(1<<P_RF_CS);

	// send 1 byte out the spi port that indicates we want to fill the TX
buffer
	SPDR = W_TX_PAYLOAD;

	// wait for transmission to finish
	while(!(SPSR & (1<<SPIF) ) );

	// write the number of bytes we want to write out to the TX buffer
	int i;
	for(i = 0; i < numBytes; i++)
	{
		// send 1 byte out the spi port. this is the data we want to write
to the RF chip
		SPDR = txDat[i];

		// wait for transmission to finish
		while(!(SPSR & (1<<SPIF) ) );
	}

	// turn the CS channel high to end transmission
	PORT_CS |= (1<<P_RF_CS);
}


// function to read a data register from the 24L01 RF chip.
// readReg - the register we want to read data from
// retDat - an appropriately sized storage container for the data we plan on
getting back
// numBytes - the number of bytes to read back. !! The retDat storage container
must be at least this size in bytes (char's)
void RFreadReg(char readReg, char* retDat, int numBytes){

	// turn the CS channel low to begin transmission
	PORT_CS &= ~(1<<P_RF_CS);

	// send 1 byte out the spi port to request write of writeReg
	SPDR = R_REGISTER + readReg;

	// wait for transmission to finish
	while(!(SPSR & (1<<SPIF) ) );

	// read the number of bytes we intend to receive into retDat
	int i;
	for(i = 0; i < numBytes; i++){
```

```
            // send 1 byte out the spi port. this is a dummy send just to read
the incoming MISO data from our read request above
            SPDR = 0xFF;

            // wait for transmission to finish
            while(!(SPSR & (1<<SPIF) ) );

            // read off the contents of our return data that is in SPDR (the
value we requested to read). This came from MISO
            retDat[i] = SPDR;
      }

      // turn the CS channel high to end transmission
      PORT_CS |= (1<<P_RF_CS);
}


// put the chip into receiving mode
void RFsetRxAddr(char * recvAddr, int numBytes){
      RFwriteReg(RX_ADDR_P1,recvAddr,numBytes);
}


// flush the tx buffer
void RFflushTXBuffer(){

      // turn the CS channel low to begin transmission
      PORT_CS &= ~(1<<P_RF_CS);

      // send 1 byte out the spi port to request write of writeReg
      SPDR = FLUSH_TX;

      // wait for transmission to finish
      while(!(SPSR & (1<<SPIF) ) );

      // turn the CS channel high to end transmission
      PORT_CS |= (1<<P_RF_CS);
}

// flush the rx buffer
void RFflushRXBuffer(){

      // turn the CS channel low to begin transmission
      PORT_CS &= ~(1<<P_RF_CS);

      // send 1 byte out the spi port to request write of writeReg
      SPDR = FLUSH_RX;

      // wait for transmission to finish
      while(!(SPSR & (1<<SPIF) ) );

      // turn the CS channel high to end transmission
      PORT_CS |= (1<<P_RF_CS);
}

// put the chip into receiving mode
void RFstartReceiving(){

      char writeDat[1] = { CONFIG_VAL | (1<<PWR_UP) | (1<<PRIM_RX) };

      // turn the PWR_UP and PRIM_RX bits high
```

```
        RFwriteReg( CONFIG, writeDat, 1);

        // wait a millisecond
        _delay_ms(2);

        // turn pin CE high
        PORT_CE |= (1<<P_RF_CE);
}

// take the chip out of receiving mode
void RFstopReceiving(){

        // turn pin CE low
        PORT_CE &= ~(1<<P_RF_CE);
}

// transmit txDat wirelessly and repeat until we receive verification that the
packet was received. Returns chip to receive state when done
char RFtransmitUntil( char* txDat, char* destAddr, char txTries){

        char txAttempt = 0;
        char success = 0;
        char tempWrite[1] = { 0x00 };

        // take out of receiving mode
        RFstopReceiving();


        // repeat transmission until the TX bit goes high (verifies receipt) or
we've tried txTries times
        while( (!success) &&  (txAttempt < txTries)  ){

                RFflushTXBuffer();

                // clear the TX transmission bit
                tempWrite[0] = (1<<TX_DS);
                RFwriteReg(STATUS, tempWrite, 1);


                // increment the attempt counter
                txAttempt++;

                // set up the destination transmit address
                RFwriteReg(TX_ADDR,destAddr,5);

                // set up the destination recive address for auto-acknowledgement
                RFwriteReg(RX_ADDR_P0,destAddr,5);

                // setup our write data to configure the chip into TX mode
                char writeDat[1] = { CONFIG_VAL | (1<<PWR_UP) };

                // turn the PWR_UP high and PRIM_RX bit low for transmitting
                RFwriteReg( CONFIG, writeDat, 1);

                //write data to TX register for outputting
                RFfillTransferBuffer(txDat,PACKET_SIZE);

                // turn pin CE high
                PORT_CE |= (1<<P_RF_CE);

                // wait 2 millisecond to make sure transfer fires
    }           _delay_ms(2);

                // end transmission by pulling CE low
                PORT_CE &= ~(1<<P_RF_CE);

                // delay so we have time to receive an ACK response
                _delay_ms(4);

                // temp variable to store our STATUS register state
                char tempStatus[1] = {0x00};
                RFreadReg(STATUS,tempStatus,1);

                // check for acknowledgement from the receiving node
                //success = tempStatus[0];
                success = ( tempStatus[0] & (1<<TX_DS));

                // clear any MAX_RT bits transmission bit
                tempWrite[0] = (1<<MAX_RT);
                RFwriteReg(STATUS, tempWrite, 1);
          }

          // if the TX bit is high clear the TX transmission bit
          if(success){
                tempWrite[0] =  (1<<TX_DS);
                RFwriteReg(STATUS, tempWrite, 1);
          }

          // turn receiving mode back on
          RFstartReceiving();

          if(success){
                return 1;
          }
          else{
                return 0;
          }
    }


    //  transmit  txDat  wirelessly  and  leave  the  chip  in  receive  mode  when  done.
    Function only fires 1 packet and does not check if data was received.
    void RFtransmit( char* txDat, char* destAddr){
          RFtransmitUntil(txDat,destAddr,1);
    }


    int RFRXdataReady(){
          // dummy write data
          char retDat[1] = {0xFF};

          RFreadReg(STATUS,retDat,1);

          if( retDat[0] & (1<<RX_DR) ){
                return 1;
          }

          else{
                return 0;
          }

    }
```

```
int RFRXbufferEmpty(){
	// dummy write data
	char retDat[1] = {0xFF};

	RFreadReg(STATUS,retDat,1);

	if( retDat[0] & 0x0E ){
		return 1;
	}

	else{
		return 0;
	}
}

// read data out of the receive FIFO and turn off the data-ready flag RX_DR if
the buffer is empty
// this function stores the previous value of the CONFIG register, which
indicated whether we were transmitting or receiving prior to this function
call, and restores this state after reading
// NOTE: it is not possible to receive or transmit new packets while readings,
all packets will be lost during this time
void RFreadRXFIFO( char* retDat){
	// take out of receiving mode
	RFstopReceiving();

	// dummy write data
	retDat[0] = 0xFF;

	// turn the CS channel low to begin transmission
	PORT_CS &= ~(1<<P_RF_CS);

	// send 1 byte out the spi port that indicates we want to read the RX
FIFO
	SPDR = R_RX_PAYLOAD;

	// wait for transmission to finish
	while(!(SPSR & (1<<SPIF) ) );

	// read the number of bytes we want to read out to the RX buffer
	int i;
	for(i = 0; i < PACKET_SIZE; i++){
		// read 1 byte out the spi port
		SPDR = retDat[i];

		// wait for transmission to finish
		while(!(SPSR & (1<<SPIF) ) );

		// store the returned data
		retDat[i] = SPDR;
	}


	// turn the CS channel high to end transmission
	PORT_CS |= (1<<P_RF_CS);

	_delay_ms(2);
```

```
        // if there is no new data ready to be read then turn the RX_DR data
ready pin low
        if(RFRXbufferEmpty()){
                char writeDat[1] = { (1<<RX_DR) };
                RFwriteReg(STATUS,writeDat,1);
        }

        // put back in receiving mode
        RFstartReceiving();
}


int RFRXbufferFull(){
        // dummy write data
        char retDat[1] = {0xFF};

        RFreadReg(FIFO_STATUS,retDat,1);

        if(retDat[0] & (1<<RX_FULL)){
                return 1;
        }else{
                return 0;
        }
}


// clear out the FIFO and return the last received packet
void RFreceive(char * buffer){

        clear(PCICR,PCIE0); // disable pin-change interrupts

        while(RFRXdataReady()){
                RFreadRXFIFO(buffer);
        }

        set(PCICR,PCIE0); // enable pin-change interrupts
}


// function to setup the rf chip registers for communication and boot it up in
receiving mode
void RFsetup(char * recvAddr, char packet_size){
        PACKET_SIZE = packet_size;
        _delay_ms(100);

        //setup the SPI port for use with the 24L01 chip
        SPIsetup();

        //setup the RF CE and CS pins as outputs

        DDR_RF_CS|=(1<<DDR_P_RF_CS);

        DDR_RF_CE |=(1<<DDR_P_RF_CE);

        //set the config register up
        char writeDat[1] = {CONFIG_VAL};
        RFwriteReg(CONFIG,writeDat,1);

        //enable PACKET_SIZE byte collection on pipe 1
        writeDat[0]  = PACKET_SIZE;
        RFwriteReg(RX_PW_P1,writeDat,1);
        //turn on auto acknowledgement on first two pipes
        writeDat[0] = 0x03;
        RFwriteReg(EN_AA,writeDat,1);

        //turn off auto retransmit
        writeDat[0] = 0x00;
        RFwriteReg(SETUP_RETR,writeDat,1);

        //enable receive on pipe 0 and 1  (this happens by default)
        //writeDat[0] = (1<<ERX_P0) | (1<<ERX_P1);
        //RFwriteReg(EN_RXADDR,writeDat,1);

        //setup address size as 5 bytes (probably deafult setup is already OK)
        //writeDat[0] = 0x03;
        //RFwriteReg(SETUP_AW,writeDat,1);

        // write receive address to pipe1
        RFsetRxAddr(recvAddr, 5);

        // turn the receiver on
        RFstartReceiving();
}

#endif
```

QUADROTOR CODE

QUADROTOR MAIN CODE (WYVERN.C)

```
// ---------------------------------------------------
// Wyvern Quadrotor
// On-Board Software
// version: 1.0.3
// date: April 29, 2010
// authors: William Etter, Paul Martin, Uriah Baalke
// ---------------------------------------------------

// DEFINES

// INCLUDES
#include "wyvern.h"
#include "uart.h"
#include "pwm.h"
#include "wyvern-rf.h"
#include "adc.h"
#include "pid.h"
#include "commands.h"

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

// CONSTANT VARIABLES

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

// GLOBAL VARIABLES
// RF Variables
char local[5] = {0x77, 0x79, 0x76, 0x30, 0x30}; // Wyvern Quadrotor Wyv00
char contr[5] = {0x63, 0x6F, 0x6E, 0x74, 0x72}; // Wyvern Controller
packet_com_t initial;
packet_com_t incoming;
```

```
packet_inf_t outgoing;
packet_razordata_t razordata;
int overflowcounter=0;
int overflowcounterbattery = 0;
extern int thrustFloor;

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

// FUNCTION HEADERS
void init_wyvern(void);
void ReceiveRazorData(void);
void batteryVoltage(void);

// ===========================================================

/************************************************************************
Interrupt Subroutines (ISRs)
************************************************************************/

ISR(PCINT0_vect)
{
	// change was high-to-low
	if(!check(PINB,4)){
		// a wireless packet was received
		RFreceive((char*) &incoming);
		// Run received command
		executeCommand(incoming.command);
	}
}

ISR(TIMER0_OVF_vect){
	// Used to send telemetry data back to Wyvern Controller
	// Timer1 Clock = 1MHz, 8-bit
	set(TIFR0,TOV0);   // Clear flag
	overflowcounter++;
	overflowcounterbattery++;
	if(overflowcounter >32){
		// Send Telemetry Data
		RFtransmitUntil((char*) &outgoing,contr,1);
		overflowcounter = 0;
	}
	if(overflowcounterbattery>200){
		batteryVoltage();
		overflowcounterbattery = 0;
	}
}

/************************************************************************
Function: main()
************************************************************************/
int main(void)
{
	init_wyvern();
	LED_ucgreen_on();
	set(DDRF,7);
	PID_setPosition(&initial);

	for(;;){
		ReceiveRazorData();
		PID_setPosition(&incoming);
		PID_setThrust(incoming.throttle);
```

```
            PID_updatePWM(&razordata);
      }
      return 0;
}

/***************************************************************************
Function: init_wyvern()
Purpose:  Runs all initialization functions
               Enables interrupts
Input:
Returns:
***************************************************************************/
void init_wyvern(void)
{
      // Setup Wyvern Systems
      init_uc();         //UC
      init_uart(); // Serial  Communication
      init_pwm();        // Motor PWM Control
      init_adc();        // ADC Initialization

      // Setup Wyvern RF
      RFsetup(local,max(sizeof(packet_com_t),sizeof(packet_inf_t)));
      set(PCICR,PCIE0); // enable pin-change interrupts
      PCMSK0 =0x00;
      set(PCMSK0, PCINT4); // demask PCINT4
      set(TCCR0B,CS02);
      set(TCCR0B,CS00);  // Timer0 Clock = System Clock/1024
      set(TIMSK0,TOIE0); // Enable Timer0 Overflow Interrupt

      // Enable Global Interrupts
      sei();

      // Force RF Interrupt (Pin Change Interrupt Channel 4) to run once
      clear(PORTB,4);
      if(RFRXdataReady()){
      RFreceive((char*) &incoming);
      }

      // Set initial orientation
      initial.yaw = 0;
      initial.pitch = 0;
      initial.roll = 0;
      initial.throttle = 0;
}

/***************************************************************************
Function: ReceiveRazorData()
Purpose:  Receives the data output from the razor
Input:
Returns:
***************************************************************************/
void ReceiveRazorData(void)
{
      clear(PCICR,PCIE0); // disable pin-change interrupts

      uart_flush();
      int startcounter = 0;
      char datain;
      char data[12];
      do{
            datain = ReceiveByte();
```

```
			if(datain == 0xFF){
				startcounter++;
			}
			else{
				startcounter =0;
			}
		}while(startcounter<3);
		data[0]=ReceiveByte();
		data[1]=ReceiveByte();
		data[2]=ReceiveByte();
		data[3]=ReceiveByte();
		data[4]=ReceiveByte();
		data[5]=ReceiveByte();
		data[6]=ReceiveByte();
		data[7]=ReceiveByte();
		data[8]=ReceiveByte();
		data[9]=ReceiveByte();
		data[10]=ReceiveByte();
		data[11]=ReceiveByte();
		memcpy(&razordata,data,12);
		if((int)razordata.pitch<0){
			ReceiveRazorData();
		}
		outgoing.yaw = (int)(razordata.yaw-20000);
		outgoing.pitch = (int)(razordata.pitch-20000);
		outgoing.roll = (int)(razordata.roll-20000);
		outgoing.p = P_pitch;
		outgoing.i = I_pitch;
		outgoing.d = D_pitch;
		outgoing.pidband = PID_BAND;

		set(PCICR,PCIE0); // enable pin-change interrupts
}

/**************************************************************************
Function: batteryVoltage()
Purpose:  Uses ADC on F4 to measure battery voltage
Input:
Returns:
**************************************************************************/
void batteryVoltage(void){
		// voltage divider: 19.8k -> 147.1k
		uint16_t battVoltage;
		battVoltage = get_adc(4);
		// return Volts*10
		battVoltage = ((double)4119*(double)battVoltage/10000);
		outgoing.battery = (int) battVoltage;
}
```

PULSE-WIDTH MODULATION (PWM) (PWM.H)

```
// --------------------------------------------------
// WyvernMaEvArM microcontroller board
// PWM
// version: 1.0.3
// date: April 29, 2010
// authors: William Etter, Paul Martin, Uriah Baalke
// --------------------------------------------------

// CONSTANT VARIABLES
unsigned int PWM_PERIOD = 20000; //50 Hz PWM given 8 MHz system clock
```

```
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

// GLOBAL VARIABLES

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

// FUNCTION HEADERS
void init_pwm(void);
void set_duty(unsigned int motornum, unsigned int duty);

// ===========================================================

/*************************************************************************
Function: init_PWM()
Purpose:  Sets up four 16-bit PWM outputs
             -Timer 1
                   -Channel A (Motor 1) (B5)
                   -Channel B (Motor 2) (B6)
                   -Channel C (Motor 3) (B7)
             -Timer 3
                   -Channel A (Motor 4) (C6)
             -Initializes OCRxx to 0 (0 duty cycle) on all motors
Input:    None
Returns:  None
*************************************************************************/
void init_pwm(void){
      //////// SET TIME PRESCALER ////////
      // Timer 1 - use system clock (system_clock/1)
      clear(TCCR1B,CS10);
      set(TCCR1B,CS11);
      clear(TCCR1B,CS12);

      // Timer 3 - use system clock (system_clock/1)
      clear(TCCR3B,CS30);
      set(TCCR3B,CS31);
      clear(TCCR3B,CS32);

      //////// SET PWM MODE ////////
      // Timer 1 - UP to ICR1 PWM Mode (Mode 14 - 16bit 65535)
      set(TCCR1B,WGM13);
      set(TCCR1B,WGM12);
      set(TCCR1A,WGM11);
      clear(TCCR1A,WGM10);
      ICR1 = PWM_PERIOD;

      // Timer 1 Channel A - clear at OCR1A, set at rollover
      set(TCCR1A,COM1A1);
      clear(TCCR1A,COM1A0);
      OCR1A = 0;

      // Timer 1 Channel B - clear at OCR1B, set at rollover
      set(TCCR1A,COM1B1);
      clear(TCCR1A,COM1B0);
      OCR1B = 0;

      // Timer 1 Channel C - clear at OCR1C, set at rollover
      set(TCCR1A,COM1C1);
      clear(TCCR1A,COM1C0);
      OCR1C = 0;

      // Timer 3 - UP to ICR3 PWM Mode (Mode 14 - 16bit 65535)
```

```
	set(TCCR3B,WGM33);
	set(TCCR3B,WGM32);
	set(TCCR3A,WGM31);
	clear(TCCR3A,WGM30);
	ICR3 = PWM_PERIOD;

	// Timer 3 Channel A - clear at OCR3A, set at rollover
	set(TCCR3A,COM3A1);
	clear(TCCR3A,COM3A0);
	OCR3A = 0;

	// Enable Timer 1 (B5,B6,B7) and Timer 3 (C6) Output
	set(DDRB,5);
	set(DDRB,6);
	set(DDRB,7);
	set(DDRC,6);
}

/***************************************************************************
Function: set_duty()
Purpose:  Sets the Duty Cycle of the four PWM controlled motors
			-Motor 1 (Timer 1 - Channel A)
			-Motor 2 (Timer 1 - Channel B)
			-Motor 3 (Timer 1 - Channel C)
			-Motor 4 (Timer 3 - Channel A)
			-Duty Cycle ranges between 0 and PWM_PERIOD
		   Auto corrects for invalid input
			-This correction will be handled earlier during motor speed
calculations
Input:    motor number, duty value
Returns:  None
***************************************************************************/
void set_duty(unsigned int motornum,unsigned int duty){
	// Prevent invalid Duty Cycle
	if(duty>2000){
		duty = 2000;
	}else if(duty<1000){
		duty = 1000;
	}

	switch(motornum){
		case 1:
			OCR1A = duty;
			break;
		case 2:
			OCR1B = duty;
			break;
		case 3:
			OCR1C = duty;
			break;
		case 4:
			OCR3A = duty;
			break;
		default:
			break;
	}
}
```

COMMANDS (COMMANDS.H)

```
// --------------------------------------------------
// Wyvern Quadrotor
// Commands
// version: 1.0.3
// date: April 29, 2010
// authors: William Etter, Paul Martin, Uriah Baalke
// -----------------------------------------------

// CONSTANT VARIABLES

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

// GLOBAL VARIABLES

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

// FUNCTION HEADERS
void executeCommand(char command);

// ===========================================================

/*************************************************************************
Function: executeCommand()
Purpose:  Runs commands based on user input
Input:    Command character
Returns:  None
*************************************************************************/
void executeCommand(char command){

      switch(command){
            case 'r':
                  LED_ucred_toggle();
                  break;
            case 'g':
                  LED_ucgreen_toggle();
                  break;
            case 's':   // Wyvern Startup
                  PID_command('s');
                  break;
            case ' ':   //  Emergency  Motor  Stop  (decrease  power  to  duty
factor of 0 immediately)
                  PID_command(' ');
                  break;
            case '=':   // increases thrust
                  PID_command('=');
                  break;
            case '-':   // decreases thrust
                  PID_command('-');
                  break;
            case '1':   // Start Motor 1
                  PID_command('1');
                  break;
            case '2':   // Start Motor 2
                  PID_command('2');
                  break;
            case '3':   // Start Motor 3
                  PID_command('3');
                  break;
            case '4':   // Start Motor 4
                  PID_command('4');
                  break;
            case 'p':
```

```
                        PID_command('p');
                        break;
                case 'o':
                        PID_command('o');
                        break;
                case 'l':
                        PID_command('l');
                        break;
                case 'k':
                        PID_command('k');
                        break;
                case 'i':
                        PID_command('i');
                        break;
                case 'u':
                        PID_command('u');
                        break;
                default:
                        break;
        }
}
```

PROPORTIONAL, INTEGRAL, DERIVATIVE (PID) CONTROLLER (PID.H)

```
// ------------------------------
// Wyvern Quadrotor
// PID
// version: 1.0.2
// date: April 29, 2010
// authors: William Etter, Paul Martin, Uriah Baalke
// ------------------------------

#define MIN_THRUST 1106
#define MAX_THRUST 1600
#define MOT1 1
#define MOT2 2
#define MOT3 3
#define MOT4 4
#define MOT1_BAL 0
#define MOT2_BAL 0
#define MOT3_BAL 0
#define MOT4_BAL 0
#define GAIN_DIVIDE 10000

// ----------------GLOBAL VARIABLES----------------
int PID_BAND = 120;

//    PID: yaw,pitch,roll
int P_yaw = 0;
int I_yaw = 0;
int D_yaw = 3500;

int P_pitch = 30;
int I_pitch = 0;
int D_pitch = 250;

int P_roll = 46;
int I_roll = 0;
int D_roll = 270;
```

```
//    PID: x,y,z
int P_acc = 0;
int I_acc = 0;
int D_acc = 0;

//    PID: altitude
int P_alt = 0;
int I_alt = 0;
int D_alt = 0;

//    current orientation, acceleration
int yaw = 0;int pitch = 0;int roll = 0;
int xAcc = 0;int yAcc = 0;int zAcc = 0;int alt = 0;

//    desired yaw,pitch,roll & x,y,z & altitude
int yaw_des = 0;int pitch_des = 0;int roll_des = 0;
int x_des = 0;int y_des = 0;int z_des = 0;
int alt_des = 0;

//    current errors (desired - actual)
int error_yaw = 0;int error_pitch = 0;int error_roll = 0;
int error_x = 0;int error_y = 0;int error_z = 0;
int error_alt = 0;

//    past errors (one cycle ago)
int error_yaw_past = 0;int error_pitch_past = 0;int error_roll_past = 0;
int error_x_past = 0;int error_y_past = 0;int error_z_past = 0;
int error_alt_past = 0;

//    yaw,pitch,roll PID calculations
int yaw_P = 0;int yaw_I = 0;int yaw_D = 0;int yaw_sum = 0;
int pitch_P = 0;int pitch_I = 0;int pitch_D = 0;int pitch_sum = 0;
int roll_P = 0;int roll_I = 0;int roll_D = 0;int roll_sum = 0;

int pitch_rate = 0;

//    x,y,z PID calculations
int x_P = 0;int x_I = 0;int x_D = 0;int x_sum = 0;
int y_P = 0;int y_I = 0;int y_D = 0;int y_sum = 0;
int z_P = 0;int z_I = 0;int z_D = 0;int z_sum = 0;

//    altitude PID calculations
int alt_P = 0;int alt_I = 0;int alt_D = 0;int alt_sum = 0;

//    thrust floor of all motors
int thrustFloor = 0;
int userThrust = 0;

// current PWM / Duty Cycle numbers
int PWM_MOT1 = 0;int PWM_MOT2 = 0;int PWM_MOT3 = 0;int PWM_MOT4 = 0;

// motor enable
int motorEnable = 0;

// have we started? do not allow quick starts
char isStarted = 0;
char activated = 0;

// ----------------FUNCTION HEADERS----------------
void PID_motorEnable(int status);
void PID_setAlt(int P, int I, int D);
void PID_setThrust(int verticalThrust);
void PID_command(char command);
void PID_setPosition(packet_com_t* data);
void PID_updatePWM(packet_razordata_t* data);

// ==============================================================
/*************************************************************************
Function: PID_motorEnable()
Purpose:  enables/disables motor PID controls
Input:    Motor PID Status
Returns:  None
*************************************************************************/
void PID_motorEnable(int status){
      motorEnable = status; // 0 or 1
}


/*************************************************************************
Function: PID_setAlt()
Purpose:  changes PID gain values
Input:    PID gains
Returns:  None
*************************************************************************/
void PID_setAlt(int P, int I, int D){
      P_alt = P;
      I_alt = I;
      D_alt = D;
}

/*************************************************************************
Function: PID_setThrust()
Purpose:  stabilize PWM using PID
Input:    IMU values: yaw,pitch,roll,altitude
Returns:  None
*************************************************************************/
void PID_setThrust(int verticalThrust){
      if(isStarted == 1){
            thrustFloor = (verticalThrust*6)/20 + 1200;
      }
}

/*************************************************************************
Function: PID_command()
Purpose:  PID Control Command execution
Input:    Command to execute
Returns:  None
*************************************************************************/
void PID_command(char command){
      switch(command){
            case 's':
                  // idle throttle
                  if(activated == 1){
                        thrustFloor = 1106;
                        motorEnable = 1;
                        set_duty(1,1106);
                        set_duty(2,1106);
                        set_duty(3,1106);
                        set_duty(4,1106);
                        isStarted = 1;
                  }
                  break;
            case ' ':
                  // emergency dead stop
                  //TransmitString("SPACE\n\r");
                  activated = 1;
                  motorEnable = 0;
                  thrustFloor = 1000;
                  set_duty(1,1000);
                  set_duty(2,1000);
                  set_duty(3,1000);
                  set_duty(4,1000);
                  isStarted = 0;
                  break;
            case '-':
                  userThrust -= 10;
                  break;
            case '=':
                  if(isStarted == 1)
                        userThrust += 10;
                  break;
            case '1':   // Start Motor 1
                  set_duty(1,1250);
                  break;
            case '2':   // Start Motor 2
                  set_duty(2,1250);
                  break;
            case '3':   // Start Motor 3
                  set_duty(3,1250);
                  break;
            case '4':   // Start Motor 4
                  set_duty(4,1250);
            case 'p':
                  P_pitch += 1;
                  break;
            case 'o':
                  if(P_pitch >= 1)
                        P_pitch -= 1;
                  break;
            case 'l':
                  D_pitch += 2;
                  break;
            case 'k':
                  if(D_pitch >= 2)
                        D_pitch -= 2;
                  break;
            case 'i':
                  PID_BAND += 1;
                  break;
            case 'u':
                  if(PID_BAND >= 1)
                        PID_BAND -= 1;
                  break;
            default:
                  break;
      }
}

/***************************************************************************
Function: PID_setPosition()
Purpose:  changes PID gain values
Input:    desired yaw,pitch,roll,alt
Returns:  None
```

```
***************************************************************************/
void PID_setPosition(packet_com_t* data){
      yaw_des      = (data->yaw)/10;
      pitch_des    = (data->pitch)/70;
      roll_des     = (data->roll)/70;
}

/**************************************************************************
Function: PID_updatePWM()
Purpose:  stabilize PWM using PID
Input:    IMU values: yaw,pitch,roll,altitude, angular accelerations
Returns:  None
***************************************************************************/
void PID_updatePWM(packet_razordata_t* data){
      if(motorEnable == 1){

            //    YAW
            error_yaw = 0 - ((int)((data->yaw)-20000));
            yaw_P = ((double)(P_yaw)*(double)(error_yaw))/GAIN_DIVIDE;
            if(yaw_P > PID_BAND){
                  yaw_P = PID_BAND;
            }else if(yaw_P < -1*PID_BAND){
                  yaw_P = -1*PID_BAND;
            }
            yaw_I += error_yaw;
            yaw_D = ((int)((data->omegatwo)-20000));
            if(yaw_D < 20 && yaw_D > -20){
                  yaw_D = 0;
            }
            yaw_D = 0-((double)(D_yaw)*(double)(yaw_D))/GAIN_DIVIDE;
            if(yaw_D > 100){
                  yaw_D = 100;
            }else if(yaw_D < -100){
                  yaw_D = -100;
            }
            yaw_sum  =  yaw_P  +  ((double)(I_yaw)*(double)(yaw_I))/GAIN_DIVIDE  +
yaw_D + yaw_des;


            //    PITCH
            //error_pitch_past = error_pitch;
            error_pitch = 0 - ((int)((data->pitch)-20000));
            pitch_P = ((double)(P_pitch)*(double)(error_pitch))/GAIN_DIVIDE;
            if(pitch_P > PID_BAND){
                  pitch_P = PID_BAND;
            }else if(pitch_P < -1*PID_BAND){
                  pitch_P = -1*PID_BAND;
            }
            pitch_I += error_pitch;
            pitch_D = 0-((int)((data->omegaone)-20000));
            if(pitch_D < 20 && pitch_D > -20){
                  pitch_D = 0;
            }
            pitch_D   =   ((double)(D_pitch)*(double)(pitch_D))/GAIN_DIVIDE   +
pitch_des;
            pitch_sum                =                pitch_P                 +
((double)(I_pitch)*(double)(pitch_I))/GAIN_DIVIDE + pitch_D;

            //    ROLL
            error_roll = 0 - ((int)((data->roll)-20000));
            roll_P = ((double)(P_roll)*(double)(error_roll))/GAIN_DIVIDE;
```

```
		if(roll_P > PID_BAND){
			roll_P = PID_BAND;
		}else if(roll_P < -1*PID_BAND){
			roll_P = -1*PID_BAND;
		}
		roll_I += error_roll;
		roll_D = ((int)((data->omegazero)-20000));
		if(roll_D < 20 && roll_D > -20){
			roll_D = 0;
		}
		roll_sum                 =                 roll_P                 +
((double)(I_roll)*(double)(roll_I))/GAIN_DIVIDE                           +
((double)(D_roll)*(double)(roll_D))/GAIN_DIVIDE + roll_des;

		//	Calculate motor responses

		PWM_MOT1 = thrustFloor + userThrust - roll_sum + yaw_sum;
		PWM_MOT2 = thrustFloor + userThrust + pitch_sum - yaw_sum;
		PWM_MOT3 = thrustFloor + userThrust + roll_sum + yaw_sum;
		PWM_MOT4 = thrustFloor + userThrust - pitch_sum - yaw_sum;

		//TransmitString("z_sum:          ");          TransmitInt(z_sum);
TransmitString("\n\r");

		if(PWM_MOT1 > MAX_THRUST){
			PWM_MOT1 = MAX_THRUST;
		}else if(PWM_MOT1 < MIN_THRUST){
			PWM_MOT1 = MIN_THRUST;
		}

		if(PWM_MOT2 > MAX_THRUST){
			PWM_MOT2 = MAX_THRUST;
		}else if(PWM_MOT2 < MIN_THRUST){
			PWM_MOT2 = MIN_THRUST;
		}

		if(PWM_MOT3 > MAX_THRUST){
			PWM_MOT3 = MAX_THRUST;
		}else if(PWM_MOT3 < MIN_THRUST){
			PWM_MOT3 = MIN_THRUST;
		}

		if(PWM_MOT4 > MAX_THRUST){
			PWM_MOT4 = MAX_THRUST;
		}else if(PWM_MOT4 < MIN_THRUST){
			PWM_MOT4 = MIN_THRUST;
		}

		set_duty(MOT1,PWM_MOT1);
		set_duty(MOT2,PWM_MOT2+10);
		set_duty(MOT3,PWM_MOT3+8);
		set_duty(MOT4,PWM_MOT4);
	}
}
```

CONTROLLER CODE

CONTROLLER MAIN CODE (WYVERNCONTROLLER.C)

```
// --------------------------------------------------
// Wyvern controller board
// Main Program
// version: 1.0.2
// date: April 29, 2010
// authors: William Etter, Paul Martin, Uriah Baalke
// --------------------------------------------------

// Includes
#include "wyvern.h"
#include "uart.h"
#include "pwm.h"
#include "wyvern-rf.h"
#include "adc.h"
#include "controller.h"

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

// CONSTANT VARIABLES

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

// GLOBAL VARIABLES
char local[5] = {0x63, 0x6F, 0x6E, 0x74, 0x72}; // Wyvern Controller
char wyv00[5] = {0x77, 0x79, 0x76, 0x30, 0x30}; // Wyvern Quadrotor Wyv00

packet_inf_t incoming;
packet_com_t outgoing;

char menuScreen = 'a';
int overflowcounter =0;
char printData = 0;
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

// FUNCTION HEADERS
void init_wyvern(void);
void controllerTransmit(void);
void getJoystick(void);

// ============================================================

/*************************************************************************
Interrupt Subroutines (ISRs)
*************************************************************************/
ISR(PCINT0_vect){
      if(!check(PINB,4)){
            LED_yellow_toggle();
            // a wireless packet was received
            RFreceive((char*) &incoming);
      }
}

/*
ISR(TIMER0_OVF_vect){
      // Timer1 Clock = 1MHz, 8-bit
      set(TIFR0,TOV0);   // Clear flag
      overflowcounter++;
      if(overflowcounter > 2){
            overflowcounter = 0;
      }
}
*/
```

```
/************************************************************************
Function: main()
************************************************************************/
int main(void){
      init_wyvern();
      LED_green_on();
      LED_ucgreen_on();
      char input;
      menuScreen = 'a';
      for( ; ; ){
            printMenu();
            input = '0';
            outgoing.command = input;
            if(DataInReceiveBuffer()){
                  input = ReceiveByte();
            }
            getJoystick();
            controllerCommand(input);
            controllerTransmit();
            _delay_ms(5);
      if(incoming.battery < 100){
                  LED_red_toggle();
                  LED_ucred_toggle();
                  LED_blue_toggle();
                  LED_yellow_toggle();
                  LED_green_toggle();
            }
      }
      return 0;
}

/************************************************************************
Function: init_wyvern()
Purpose:  Runs all initialization functions
              Enables interrupts
Input:    None
Returns:  None
************************************************************************/
void init_wyvern(void){
      // Setup Wyvern Systems
      init_uc();         //UC
      init_uart(); // Serial  Communication
      //init_pwm();            // Motor PWM Control
      init_adc();            // ADC
      // Setup Wyvern RF
      RFsetup(local,max(sizeof(packet_com_t),sizeof(packet_inf_t)));
      set(PCICR,PCIE0); // enable pin-change interrupts
      PCMSK0 =0x00;
      set(PCMSK0, PCINT4); // demask PCINT4

      //clear(TCCR0B,CS02);
      //set(TCCR0B,CS01);
      //clear(TCCR0B,CS00);    // Timer0 Clock = System Clock/1024
      //set(TIMSK0,TOIE0);     // Enable Timer0 Overflow Interrupt

      // Enable Global Interrupts
      sei();

      // Force RF Interrupt (Pin Change Interrupt Channel 4) to run once
      clear(PORTB,4);
      if(RFRXdataReady()){
```

```
	RFreceive((char*) &incoming);
	}

	incoming.battery = 150;
}

/*************************************************************************
Function: controllerTransmit()
Purpose:  Transmits the Controller Packet
Input:    None
Returns:  None
**************************************************************************/
void controllerTransmit(void){
	// LED_ucred_toggle();
	RFtransmitUntil((char*) &outgoing,wyv00,1);
}

/*************************************************************************
Function: getJoystick()
Purpose:  Gets the X and Y (Pitch and Roll) Joystick data
Input:    None
Returns:  None
**************************************************************************/
void getJoystick(void){
	int adcval;
	// get right X (Roll) F1
	adcval = 512-get_adc(1);
	if(adcval > -4 && adcval <4){
		outgoing.roll = 0;
	}else{
		outgoing.roll = adcval;
	}

	// get right Y (Pitch) F0
	adcval = get_adc(0) - 512;
	if(adcval > -4 && adcval <4){
		outgoing.pitch = 0;
	}else{
		outgoing.pitch = adcval;
	}

	// get left X (Yaw) F5
	adcval = 512-get_adc(5);
	if(adcval > -4 && adcval <4){
		outgoing.yaw = 0;
	}else{
		outgoing.yaw = adcval;
	}
	// get left Y (Throttle) F4
	adcval = get_adc(4);
	if(adcval > 512){
		outgoing.throttle = adcval-512;
	}else{
		outgoing.throttle = 0;
	}
}
```

CONTROLLER (CONTROLLER.H)

```
// ------------------------------------------------
// Wyvern Quadrotor
// Controller
// version: 1.0.2
// date: April 29, 2010
// authors: William Etter, Paul Martin, Uriah Baalke
// -------------------------------------------------

// CONSTANT VARIABLES

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

// GLOBAL VARIABLES
extern packet_com_t outgoing;
extern packet_inf_t incoming;
extern char menuScreen;
extern char printData;
char str[20];
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

// FUNCTION HEADERS
void printMenu();
void controllerCommand(char inputcommand);
extern void controllerTransmit(void);
void printEmpty(int num);
extern void getJoystick(void);

// ===========================================================

/*************************************************************************
Function: printMenu()
Purpose:  Prints the Wyvern User Interface
Input:    Menu Screen to print
Returns:  None
*************************************************************************/
void printMenu(){

      switch(menuScreen){
            case 'a': // Main Menu
                  TransmitString("~Wyvern Quadrotor Controller~\n\r");
                  TransmitString("s = START Quadrotor (Motor Spinup)\n\r");
                  TransmitString("SPACEBAR = EMERGENCY STOP\n\r");
                  TransmitString("r = Toggle On-board Red LED\n\r");
                  TransmitString("g = Toggle On-board Green LED\n\r");
                  TransmitString("d = Print Wyvern Telemetry Data\n\r");
                  TransmitString("m = Motor Menu\n\r");
                  TransmitString("p = PID Menu\n\r");
                  TransmitString("j = Joystick Menu\n\r");
                  printEmpty(6);
                  break;
            case 'b': // Motor Menu
                  TransmitString("~Motor Menu~\n\r");
                  TransmitString("s = START Quadrotor (Motor Spinup)\n\r");
                  TransmitString("SPACEBAR = EMERGENCY STOP\n\r");
                  TransmitString("= = Increase Thrust\n\r");
                  TransmitString("- = Decreaase Thrust\n\r");
                  TransmitString("1,2,3,4 = Start Motor X\n\r");
                  TransmitString("m = Main Menu\n\r");
                  TransmitString("p = PID Menu\n\r");
                  printEmpty(7);
                  break;
            case 'c': // Motor Menu
                  TransmitString("~PID Menu~\n\r");
```

```
                        TransmitString("s = START Quadrotor (Motor Spinup)\n\r");
                        TransmitString("SPACEBAR = EMERGENCY STOP\n\r");
                        TransmitString("= = Increase Thrust\n\r");
                        TransmitString("- = Decreaase Thrust\n\r");
                        TransmitString("p = Increase P gain\n\r");
                        TransmitString("o = Decrease P gain\n\r");
                        TransmitString("i = Increase PID BAND\n\r");
                        TransmitString("u = Decrease PID BAND\n\r");
                        TransmitString("l = Increase D gain\n\r");
                        TransmitString("k = Decrease D gain\n\r");
                        TransmitString("d = Print out P gain and D gain\n\r");
                        TransmitString("m = Main Menu\n\r");
                        printEmpty(2);
                        break;
                case 'd': // Joystick Menu
                        TransmitString("~JoyStick Menu~\n\r");
                        TransmitString("s = START Quadrotor (Motor Spinup)\n\r");
                        TransmitString("SPACEBAR = EMERGENCY STOP\n\r");
                        TransmitString("d = Print out Joystick Data\n\r");
                        TransmitString("m = Main Menu\n\r");
                        printEmpty(10);
                        break;
                default:
                        break;
        }

}

/************************************************************************
Function: controllerCommand()
Purpose:  Commands to run on the Wyvern Controller
Input:    Command characters
Returns:  None
************************************************************************/
void controllerCommand(char inputcommand){
        switch(menuScreen){
                case 'a': // Menu Screen A (Main Menu)
                        switch(inputcommand){
                                case 's': // Start Quadrotor
                                        //TransmitString("Starting Quadrotor\n\r");
                                        outgoing.command = inputcommand;
                                        break;
                                case ' ': // Emergency Stop
                                        outgoing.command = inputcommand;
                                        break;
                                case 'r': // Toggle On-board Red LED
                                        //TransmitString("Toggling Red LED");
                                        outgoing.command = inputcommand;
                                        break;
                                case 'g': // Toggle On-board Green LED
                                        //TransmitString("Toggling Green LED");
                                        outgoing.command = inputcommand;
                                        break;
                                case 'd': // Print out telemetry data
                                        // PRINT OUT VALUES CONTINUOUSLY
                                        while(!(DataInReceiveBuffer())){
                                                        TransmitString("Yaw = ");
                                                        TransmitInt(incoming.yaw);
                                                        TransmitString("   Pitch = ");
                                                        TransmitInt(incoming.pitch);
                                                        TransmitString("   Roll = ");
```

```
                    TransmitInt(incoming.roll);
                    TransmitString("   Battery = ");
                    TransmitInt(incoming.battery);
                    TransmitString("\n\r");
            }
            break;
        case 'm': // Go to Motor Menu (MenuScreen = 'b');
            menuScreen = 'b';
            break;
        case 'p': // Go to PID Menu (MenuScreen = 'c');
            menuScreen = 'c';
            break;
        case 'j': // Go to Joystick Menu (MenuScreen = 'd');
            menuScreen = 'd';
            break;
        default:
            break;
    }
    break;
case 'b': // Menu Screen B (Motor Menu)
    switch(inputcommand){
        case 's': // Start Quadrotor
            outgoing.command = inputcommand;
            break;
        case ' ': // Emergency Stop
            outgoing.command = inputcommand;
            break;
        case '=': // Increase Thrust
            outgoing.command = inputcommand;
            break;
        case '-': // Decrease Thrust
            outgoing.command = inputcommand;
            break;
        case '1': // Start Motor 1
            outgoing.command = inputcommand;
            break;
        case '2': // Start Motor 2
            outgoing.command = inputcommand;
            break;
        case '3': // Start Motor 3
            outgoing.command = inputcommand;
            break;
        case '4': // Start Motor 4
            outgoing.command = inputcommand;
            break;
        case 'm': // Return to Menu Screen 'a' (Main Menu)
            menuScreen = 'a';
            break;
        case 'p': // Go to Menu Screen 'c' (PID Menu)
            menuScreen = 'c';
            break;
        default:
            break;
    }
    break;
case 'c': // Menu Screen C (PID Menu)
    switch(inputcommand){
        case 's' : // Start Quadrotor
            outgoing.command = inputcommand;
            break;
        case ' ': // Emergency Stop
```

```
                        outgoing.command = inputcommand;
                        break;
                case '=': // Increase Thrust
                        outgoing.command = inputcommand;
                        break;
                case '-': // Decrease Thrust
                        outgoing.command = inputcommand;
                        break;
                case 'p':
                        outgoing.command = inputcommand;
                        break;
                case 'o':
                        outgoing.command = inputcommand;
                        break;
                case 'i':
                        outgoing.command = inputcommand;
                        break;
                case 'u':
                        outgoing.command = inputcommand;
                        break;
                case 'l':
                        outgoing.command = inputcommand;
                        break;
                case 'k':
                        outgoing.command = inputcommand;
                        break;
                case 'd': // Print out Gain Data
                        // PRINT OUT VALUES CONTINUOUSLY
                        while(!(DataInReceiveBuffer())){
                                        TransmitString("P_gyr = ");
                                        TransmitInt(incoming.p);
                                        TransmitString("   D_gyr = ");
                                        TransmitInt(incoming.d);
                                        TransmitString("   PID BAND = ");
                                        TransmitInt(incoming.pidband);
                                        TransmitString("\n\r");
                        }
                        break;
                case 'm': // Return to Menu Screen 'a' (Main Menu)
                        menuScreen = 'a';
                        break;
                default:
                        break;
        }
        break;
case 'd': // Menu Screen D (Joystick Menu)
        switch(inputcommand){
                case 's': // Start Quadrotor
                        outgoing.command = inputcommand;
                        break;
                case ' ': // Emergency Stop
                        outgoing.command = inputcommand;
                        break;
                case 'd': // Print out Joystick Data
                        // PRINT OUT VALUES CONTINUOUSLY
                        while(!(DataInReceiveBuffer())){
                                        getJoystick();
                                        TransmitString("Pitch = ");
                                        TransmitInt(outgoing.pitch);
                                        TransmitString("   Roll = ");
                                        TransmitInt(outgoing.roll);
                                        TransmitString("   Yaw = ");
                                        TransmitInt(outgoing.yaw);
                                        TransmitString("   Throttle = ");
                                        TransmitInt(outgoing.throttle);
                                        TransmitString("\n\r");
                                  }
                                  break;
                            case 'm': // Return to Menu Screen 'a' (Main Menu)
                                  menuScreen = 'a';
                                  break;
                            default:
                                  break;
                      }
                      break;
                default:
                      break;
      }
}

/**************************************************************************
Function: printEmpty()
Purpose:  Prints out a empty line to fill up the menu screen
Input:    Number of empty lines to print
Returns:  None
**************************************************************************/
void printEmpty(int num){
      for(int i=0;i<num;i++){
            TransmitString("\n\r");
      }
}
```