ECE 477 Final Project

Peter Bilodeau
peterjbilodeau@gmail.com

Brady Butler
m.brady.butler@gmail.com

Breanna Stanaway

Cody Morgan

May 2, 2012

Abstract

For this project, we designed and built an autonomous flying quadrotor. Our design anticipates automatic stability control and navigation. This project lays the foundation for integrated sensor and motor control systems. We built a custom aluminum frame and a custom electronic main board, and use stock hobby motors, propellers and lithium-polymer battery.

Contents

1 Flight Dynamics

Throughout this paper we us a number of terms to describe quadrotor flight dynamics. We begin with a brief discussion and some definitions. A quadrotor has six degrees of freedom. Two in the horizontal plane (forward/back and left/right), one vertically (altitude), and 3 axes of rotation. We decided to define one of the four rotors as the 'front' of the craft. This rotor we call 'north'; the others are 'south,' 'east,' and 'west' respectively. Some designs define a pair of rotors as the front. This is known as flying 'X' mode as opposed to '+' mode like our project. Using standard flight terminology, we defined three axes of rotation as follows: rotation about the vertical axis is called yaw, where counter-clockwise (turning left) is the positive direction, rotation about the horizontal axis perpendicular to the forward direction is called pitch, north rotor up (leaning back) is the positive direction, finally, rotation about the horizontal axis parallel to the forward direction is called roll, where leaning left is the positive direction.

Adjusting the roll and pitch of the craft in flight is fairly straightforward. If one rotor is lower than the others, then the quadrotor is leaning in that direction. We can increase the thrust of that motor to compensate and level out. Yaw, on the other hand, is slightly more complicated. We use 2 pair of propellers rotating in opposite directions. Each pair, then, generates a torque opposing that of the other pair. We can rotate in the yaw direction by speeding up one set of rotors with respect to the other. Since simply speeding up two of the motors will generate more overall lift, the quadrotor will fly up. Thus we need to reduce the thrust of the opposite pair of rotors any time as we increase the thrust of a given pair. This will keep the overall thrust constant, while still producing a yaw rotation. Similarly, when correcting roll and pitch, we decrease the thrust of the higher rotor at the same time we increase the thrust of the lower one. What the controls really adjust then, is the relative speed of opposing rotors or pairs of rotors.

2 System Architecture

The quadrotor control board integrates several distinct systems. For example, we have the motor control system. Each of the four motors requires a PWM control signal. On the ATmega328p, we use Timer0 and Timer2 each with two compare values to generate these signals. Since we also need Timer1 to controlsensor polling frequencies, there are no other timers available if we use a single chip. One of the sensors (the sonar) requires input capture and timing capability, so we added a second AVR to the control board. The two AVR communicate via I^2C with one chip in master mode, and the sonar chip in slave mode.

This setup is particularly convenient for future expansion because we are free to add additional AVRs in slave mode, if necessary. Also, the USART pins on the slave chip are no longer needed for XBee wireless communication, which lets us use serial sensor devices if desired. For example, some GPS modules communicate via USART.

3 Motor Control

We selected brushless outrunner motors that supply about 700+ grams of thrust depending on the propellers used. These motors have three connections and require an electronic speed controller or ESC. The ESC handles alternating power to each of the the three connections in order to make the motor spin, as well as switching power on and off. This cannot be done by the microcontroller because each motor draws about 13 Amps under full power.

In addition to the three motor connections, each ESC has high-current connections to battery ground and +12V. There is also a servo connector which carries the control signal. The servo connector has 3 wires: signal, ground, and +5V. The +5V connection is actually a power supply. We could use this to power our control board or other electronics; however, if the ESC fails, the control board would lose power at the same time. This is less important for a quadrotor, which is probably crashing anyways, but may matter for a fixed wing aircraft that could glide to a safe landing if control is maintained. For redundancy, we use the battery eliminator circuit discussed elsewhere.

The ESCs expect a special kind of PWM signal as input and use that to set the motor speed. For example, ESCs will take a 50Hz singal with pulses between 1 and 2ms wide. A 1ms wide pulse then represents the 'minimum signal,' or 0% power, and a 2ms wide pulse represents the 'maximum signal,' or 100% power. Since we are using an 8 bit timer which overflows every 20ms, we only have 14 different compare values in the desired range. This provides only about 7% resolution for motor control, which is very coarse for this application. Luckily, the ESC we selected are programmable, so we were able to step up to a 488Hz and stretch the minimum and maximum signal as far appart as the ESCs would accept. We can now use compare values between 78 and 254, giving us a resolution of about 0.57%. One other benefit of using a higher frequency control signal is that the control board can now update the motor speed roughly every 2ms rather than the 20ms imposed by the 50Hz signal.

The plots below show the control signals for both 50Hz and 488Hz frequencies with minimum and maximum pulse widths for each.

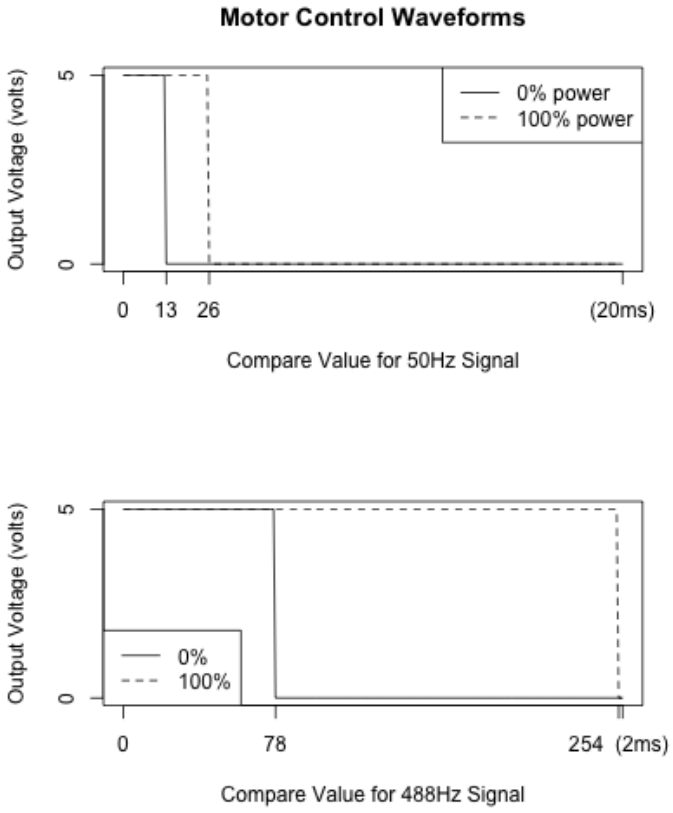

Figure 1: Motor control signals for two different frequencies

4 Onboard Power

This section discusses how the copter distributes power. An 18-volt 3-cell lithium polymer battery powers the quadrotorcopter. At full power this battery gives approximately 3-4 minutes of flying time. Lithium polymer batteries should never be subjected to an impact, overcharging, or undercharging. Special precautions avoided these events. ESCs monitored the battery charge levels to avoid a charge below the threshold. Foam padding secured the battery holder, avoiding severe impacts.

Four electronic speed control devices (ESCs) distribute the power and regulate current to the motors. The ESCs supply a 5V voltage regulator and a low battery-warning feature. The ESC voltage regulator was not used as the AVR voltage supply. This way even if one particular motor stopped working the AVR would continue to function. Two separate onboard voltage regulators drop the supply voltage down to 5-volts, used for the AVR, and 3.3 volts, supplied to the sensor.

5 Communication

The serial communication setup for this project was based the setup used in previous ECE477 labs. However, changes needed to be made in order to compensate for the large amount of data that the AVR needs to send at a relatively low baud rate. The main differences lie in the addition of wireless hardware and the handling of data transmission on the AVR.

When communicating with a quadcopter, it is desirable to communicate wirelessly. XBee wireless modules were used to accomplish this. An XBee module was connected to the Sheaff-Monk AVR board via the TX and RX lines to provide wireless capability to the PC. A second XBee module, tuned to the same frequency, was connected to the TX and RX lines of the master ATmega328P on the quadcopter. This wireless configuration behaved the similarly to the wired connection between the PC and AVR in lab 5.

One change to the wireless modules that was needed was to change the bandwidth of the on-board bandpass filter. This reduced the interference seen and allowed the PC and quadcopter to communicate when in range of other XBee wireless devices.

The biggest change made to the serial communication was in the handling of data transmission on the AVR side. The first change was the creation of a queue of data to be sent. The next change was setting up an interrupt to be called when a transmission was completed. The interrupt allowed the AVR to check the contents of the transmission queue in an interrupt subroutine and begin sending the next piece of data.

The interrupt and queue were set up in *AVRserial.h*. The interrupt used was the Usart Data Register Empty (UDRE) interrupt, which is enabled using bit 5 of the USART Status and Control Register B (UCSRB). The transmission queue was set up as a column array of bytes to be sent. To ensure that no data was skipped, the a pointer was used to cycle through all the items in the array until the array was empty.

6 Sensors

To create a stable system different aspects of the flight were monitored using sensors. Section 7.2 discusses the integration of all sensors to create an attitude. This section discusses the following sensors: accelerometer, gyroscope, magnetometer, barometer, and sonar rangefinder.

6.1 Wii Nunchuck 3-Axis Accelerometer

The Wii Nunchuck accelerometer is an I^2C device that provides 10 bits of data representing the magnitude of acceleration in each of the x, y, and z axes. The data obtained from the accelerometer form the foundation of the roll and pitch estimations for the quadrotor. The I^2C write address of the accelerometer is 0xA4 while the read address is 0xA5. All code related to the accelerometer can be found in the *nunchuck.h* header file.

Initialization Before it can be used, the device must first be initialized in one of two different ways. The first method involves simply writing the value 0x00 to address 0xA4 of the accelerometer. This will cause the device to send back encrypted data that must be decrypted before it can be used. Since we sought to poll the sensory devices, calculate the new motor values, and then update the quadrotor as fast as possible, we chose to forgo the encryption route.

The second initialization method is accomplished by writing the value 0x55 to device address 0xF0 and then writing the value 0x00 to address 0xFB. This will put the device in unencrypted mode and allow all data to be immediately useable as it comes back from the accelerometer. We initialize our quadrotor this way using the **power_on_nunchuck()** function.

Reading the Data Once a read is initiated, data comes back from the accelerometer in six byte chunks. The first two bytes pertain to the position of the previously attached joystick and are ignored for the purposes of this project. The next three bytes that come from the device are the x, y, and z axis 8 MSB, respectively. The final byte contains the 2 LSB for each axis as well as button press data for the two buttons that were previously attached as part of the Wii nunchuck. For each axis, to reconstitute the data, we left shift the 8 MSB two places and then OR that result with the 2 LSB to yields the full 10 bit resolution. The accelerometer data read and reconstitution is accomplished via the **get_data_nunchuck()** function. It is important to note that within this function, **send_zero_nunchuck()** is called, which simply

sends the value 0x00 to the device. This must be done after each data read or the accelerometer will stop responding to read requests.

Interpreting the Data The 10 bit value of each axis retrieved from the accelerometer is an unsigned integer representing the magnitude of the acceleration along that distinct axis. Since the 10 bit value is unsigned with a range between 0 and 1023, each axis has a characteristic offset so that the zero value falls close to the middle of the range. These values were determined for the x and y axes by reading the raw values from the accelerometer while the device was flat and stationary (thus experiencing zero x and y acceleration). Once acquired, we hardcoded these values using defines in the header file. The z axis was not calibrated since it was not needed for attitude calculation, as will be explained in the subsequent paragraph.

Calculating the attitude of the quadrotor using the accelerometer presented a challenge because it involved a significant amount of trig operations, which are expensive in general, but especially so on the AVR. Therefore we decided to use the small angle approximation which gives a value within 5% of the actual angle up to 30°. To do this, we measured the maximum value given by the accelerometer for the x and y axes (i.e. when acceleration for each axis was at a maximum) and then used the following equation to obtain the quadrotor's roll and pitch, respectively.

$$\theta = \frac{axis\_value}{axis\_max} \times \frac{180}{\pi} \tag{1}$$

This method was sufficient to accurately predict the roll and pitch of the quadrotor for angles up to 35°.

6.2 2-axis Gyroscope

The 2-axis gyroscope requires Analog-to-Digital Conversion (ADC) and gives a rotational velocity in degrees per second about the x and y axes. This data was used to help smooth out the roll and pitch measurements derived from the data given by the 3-axis accelerometer. All code related to the gyroscope can be found in the *gyro.h* header file.

Setting up the ADC A previous lab addressed the specifics of performing ADC on the AVR. The ADC used for the quadrotor is not significantly different. Since the 2-axis gyro has two different analog outputs, the x and y outputs are connected to ADC Channels 0 and 1 on the AVR, respectively. The only major difference between the use of ADC in that lab and the use of ADC for the quadrotor is that we chose to use the AREF pin to provide a reference voltage of 3.3V. We only learned of the danger in using this method after our quadrotor was already built, otherwise we would have used the AV_{CC} as a reference voltage. AREF mode is set by clearing the REFS1:0 bits of the ADMUX register.

Triggering and Reading the Conversion An ADC is triggered by writing a 1 to the ADSC bit of the ADCSRA register. We handle the reading of the data through the *ADC_vect* interrupt. When the interrupt is triggered, we read the 8 LSB from the ADCL register followed by the 2 MSB from the ADCH register. Just like the accelerometer, the gyroscope also has a bias that must be taken into account. We measured the bias of each axis when the gyroscope was at rest and then hardcoded the values in using defines. When we do calculations, the bias is subtracted to give the true rotational velocity.

One other important thing to note is that gyroscope actually has four different outputs. It has the standard x and y axis (roll and pitch, respectively), but it also has two other channels for High Resolution Mode. These high resolution outputs are 4.5 times more sensitive to change than the standard outputs. We do not use these outputs from the gyroscope, but our code is capable of handling them through the use of a ternary operation in the interrupt vector where the gyroscope value is multiplied by a scaling factor to give degrees per second. Once the value is converted to this form, we check which channel was being used, x or y axis, and then put the data into the proper sensor data cache variable. Finally, we switch the ADC channel to the other axis by toggling the LSB of the ADMUX register so that we alternate between reading the x and y axis rotational velocity from the gyroscope.

6.3 3-axis Magnetometer

The magnetometer is an I^2C device that provides three 16 bit values representing the magnitude of the magnetic field along the x, y and z axes. This information is used to measure the heading of the quadrotor. In future iterations, this heading data will be used to correct for yaw. The I^2C write address of the magnetometer is 0x3C while the read address is 0x3C. All code related to the magnetometer can be found in the *magnetometer.h* header file.

Initialization Before the magnetometer can be used, it must be initialized by writing three bytes to the device. First, the value 0x70 must be written to device address 0x00. Then the value 0x01 must written to device address 0xA0. Finally, the value 0x02 must be written to device address 0x00. Once this is done, the magnetometer is ready to be used. This process is handled on the quadrotor by the **power_on_magnetometer()** function.

Reading and Interpreting the Data Data is read from the magnetometer by writing the address 0x03 to the device, and then performing a 6 byte read. The bytes that come back are the 8 MSB, followed by the 8 LSB of the x, y, and z axes, respectively. This data does not have any offset that needs to be taken into account. Instead, the three values are the x, y, and z components of a vector pointing towards Magnetic North. The **compensate_for_tilt()** function uses these three raw vector values, as well as the current roll and pitch values, to give the heading of the quadrotor in degrees. The **get_data_magnetometer()** function performs the 6 byte read, calls the conversion function, and then puts the heading value into the **compass_heading** component of the **sensor_data_cache**.

6.4 Barometer

We had originally planned on using a barometer for altitude readings, but the device arrived with calibration values that were outside of the acceptable range as set by the manufacturer, thus indicating a defective item. We wrote all of the code before actually checking the EEPROM burned calibration values, so there is a *barometer.h* header file that contains all of the code to use this device despite the fact that our quadrotor does not utilize any of it. The barometer is a standard I^2C device with a few peculiarities. The device must have a control byte written to it to start a pressure or temperature reading, and then it requires 4.5 *ms* before the value can actually be read. That is a very long time to wait in terms of updating the motor values, so perhaps is was for the best that this device was broken.

6.5 Sonar

To handle the altitude reading of our quadrotor we used a sonar device. The sonar device is not, like most of our devices, I^2C. Instead, it gives a rising edge on its output pin when it sends out a sound wave, and then gives a falling edge on its output pin when the wave returns. Thus, in order for the sonar device to be used, we needed two compare values and an input capture interrupt on a single clock. Since the clocks on our AVR were already being used for other things (motor control, control loop timing), we decided to put the sonar device on another AVR and then setup that AVR as an I^2C slave. All code related to the sonar can be found in the *sonar_slave.h* header file.

Initialization The sonar device does not need any initialization. Instead, the AVR slave must be setup with the proper clock prescalar and interrupt mask to use input capture with the sonar device. This is done within the **setup_sonar()** function. A prescalar of 8 is used with the 8 *MHz* to give a clock that counts in 1 μs increments. The OCR1A value is set to 60000 so that the Timer 1 COMPA interrupt is called every 60ms. The timer mode is also set so that the clock resets when it hits the OCR1A value. The OCR1B value is set to 5 so that the Timer 1 COMPB interrupt is called 5 μs after the COMPA interrupt.

Starting a Reading A sonar reading is triggered on the slave AVR every time the Timer 1 COMPA interrupt vector is called. In this interrupt, PORTB is set to output mode and then Pin 0 is set high. 5 μs later the Timer 1 COMPB interrupt vector is called which sets Pin 0 low and changed PORTB to input mode. This interrupt also sets the **WAITINGFORECHO** flag which indicates that a sonar reading is in progress. The 5 μs long pulse triggers the sonar device to send out a sound wave and thus Pin 0 needs to start listening for a rising edge from the sonar device. The input capture interrupt catches the rising edge of the sonar device, notes the time, and then starts listening for a falling edge. When the falling edge is captured, the $\Delta time$ is calculated and then the sonar distance in feet is calculated via the formula:

$$distance = \frac{\Delta time \times 1130 ft/s/2}{1000000} \tag{2}$$

Notice that we divide the speed of sound by 2 since sound must travel out and back, which is twice the distance we want to measure, and we divide by one million since $\Delta time$ is in microseconds.

6.6 Device Polling

While the accelerometer, gyroscope, and magnetometer can be polled at a frequency approaching *200 Hz*, the sonar device can only be polled at a maximum of about *50 Hz*. Therefore it was imperative to come up with a polling scheme. This was accomplished by using Timer 1 on the master AVR in CTC mode with ICR1 as the top value and the COMPA interrupt. We used a prescalar of 8 so that the *8 MHz* clock would count in $1\mu s$ increments. The top value of the clock, ICR1, was set to 0x7FFF so that the clock would overflow every $32.77\mu s$, or *30.5 Hz*. The OCR1A value was then initially set to 0x2000 so that it would trigger $8.19\mu s$ after the clock begins. Every time the COMPA interrupt is called, it increments the OCR1A value by 0x2000 so that the interrupt is called again $8.19\mu s$.

This setup yields two interrupts firing at different frequencies. The first interrupt, Timer 1 Capture, is triggered at a frequency of *30.5 Hz*. The second interrupt, Timer 1 COMPA, is triggered at a frequency of about *122 Hz*. In order to use these interrupts for device polling, we set the **TOP_flag** and **compare_A_flag** flags in these two interrupts, respectively. The first flag indicates that the sonar device should be polled while the second flag indicates that the accelerometer, gyroscope, and magnetometer should be polled. The main control loop of the code simply checks if the flag is set, and then polls the proper devices.

7 Data Filtering

The data coming in from our sensory devices was capable of getting quite noisy. Considering that the quadrotor is always moving and shaking when in flight, it was necessary to smooth out the data using a filtering scheme. The altitude and attitude calculations were smoothed out using two separate methods. Data filtering is handled through functions located in the *data.h* header file.

7.1 Altitude

Every time an altitude reading is obtained by querying the slave AVR, the master AVR calls **update_adj_alt()**. This function takes a weighted average of the old altitude reading and the new altitude reading before assigning this value as the new altitude. The weighted average is expressed below:

$$new\_altitude = 0.75 \times old\_altitude + 0.25 \times new\_altitude\_reading \tag{3}$$

7.2 Attitude

The attitude calculations (i.e. roll and pitch) are smoothed out using a method called a Complementary Filter contained within the **update_adj_rp()** function. This function is called every time a new roll and pitch have been calculated following polling of the accelerometer and gyroscope. The filter takes a weighted average of the new roll/pitch value and the old roll/pitch value plus an integration of the rotational velocity.

This helps to smooth out situations where the quadrotor has acquired lateral drift and the accelerometer reads an x or y acceleration not due to gravity. Since the gyroscope will read a rotational velocity close to zero, the majority term will be the old roll/pitch. An example calculation follows:

$$new\_roll = 0.75 \times (old\_roll + \delta \cdot gyro\_roll) + 0.25 \times new\_roll\_reading \tag{4}$$

8 I^2C Bus

I^2C is a two wire serial communication protocol. The I^2C bus consists of a data line (SDA) and a clock line (SCL). On the bus it is possible to have multiple master and slave devices that are able to read and write from each other.

Communication is initiated with a start bit followed by an address packet consisting of the device address, and a read (1) or write (0) bit. If the the slave exists, it sends an acknowledge bit (ACK) and begins transmitting or receiving data packets. Data is sent one byte at a time until all the desired bytes have been read, with the receiving device sending an ACK after every byte until the last byte when a stop condition is sent.

Address packets: 9 bits; 7 address bits, one read/write, one acknowledge bit **Data packet**: 9 bits; 8 data bits (one byte), one acknowledge bit

Address and data packets are sent MSB first and the acknowledge bit is represented by a low bit on SDA. The start bit is represented by a high to low transition on the SDA while SCL is high. The stop bit is represented by a low to high transition on the SDA with the SCL high. Note that sampling of the SDA occurs when the SCL is high.

Drawbacks of I^2C:

- Limited number of device addresses (seven bit addresses means only 128 devices are available)
- Limited range of speeds (standard frequency is 100kbit/s, higher frequency modes are not commonly supported by most devices)
- Shared bus, if one device faults, it interrupts communication for all other devices

On the AVR I^2C communication uses the two-wire serial interface (TWI). The Atmega328p supports up to 400kHz bus speeds. Referring to the ATmega328p datasheet it is found that the TWI uses six registers on the microcontroller: TWBR, TWCR, TWSR, TWDR, TWAR, TWAMR.

The SCL frequency is controlled by the Two-Wire Bit Rate Register (TWBR) and the prescaler bits (bits 1 and 0) in the Two-Wire Status Register (TWSR).The prescaler is determined by using the following formula:

$$SCLfrequency = \frac{CPUClockfrequency}{(16 + 2(TWBR)(PrescalerValue))} \tag{5}$$

The Two-Wire Control Register (TWCR) contains the bits used to enable the TWI and its various features. As well as the prescaler code, the TWSR contains five bits which denote the status of the TWI and one reserved bit which will always be read as zero. The Two-Wire Data Register (TWDR) will either contain the last byte received or the next byte to be transmitted depending upon the mode that the microcontroller is in (receive or transmit respectfully). The last two registers are the Two-Wire Address Register (TWAR) and the Two-Wire Address Mask Register (TWAMR) are used to assign a seven bit slave address to a microcontroller in slave mode and a mask to allow the slave to respond to multiple addresses.

9 PID Control

This section discusses the PID stability control used in this project. Subsection 9.1 explains the PID theory used in this project, and subsection 9.2 discusses how the quadrotorcopter utilizes PID controllers to stabilize the copter during flight. Subsection 9.3 discusses the tuning method used for the controllers.

9.1

In a PID controller P denotes proportional, D derivative, and I integral. The proportional control moves the response of the plant (the copter), in the correct direction. Proportional control alone will seldom result in the desired output so the derivative and integral terms are added. A PD controller improves the transient response of the system, or the oscillatory response. PI controllers improve the system steady state error. A PID controller improves both the steady state error and transient response simultaneously. Equation 6 gives the generic equation for a PID controller.

$$K_p + K_d s + K_i \frac{1}{s} \tag{6}$$

9.2 PD Controller with bias

Ideally a PID controller would have resulted in the most stable flight. For simplicity purposes, this project implements a PD controller with a constant bias term instead of an integral term. Equation 7 shows how we model a PD controller.

$$K_P + K_D s + Bias \tag{7}$$

Equation 8 shows how to calculate the derivative term of the controller.

$$derivative = E - E' \tag{8}$$

where E is the previous error and E is the current error.

9.3 Ziegler-Nichols Tuning Method

Given the complexity of quadrotor dynamics, we quickly deemed the plant impossible to model. Ziegler-Nichols tuning method was selected to tune the individual controllers. The individual controllers yaw, pitch, roll, and thrust were tuned independently. To tune a single controller all other controllers were turned off and we fixed the quadrotor so that it could only move with the one given degree of freedom. For example, to tune the roll controller the pitch component and altitude component were held constant by tying the north/south axis down, leaving the quadrotor free to roll.

Using the Ziegler-Nichols method the proportional constant was increased until a steady oscillation was found. This value gave the ultimate gain constant K_u. At the ultimate gain the period of oscillation was found. Equation 9 shows how to use the ultimate gain constant and ultimate period to make the P and D terms of the controller.

$$K_P = \frac{K_u}{1.7} \qquad K_D = \frac{P_u}{8} \tag{9}$$

10 Master Control

The main code for the quadrotor can be found in the *master.c* file. The program first sets up serial communication and the various devices to be used by the AVR (I^2C, ADC, and motors) in addition to the takeoff controller. It then enters into the main control loop where it continuously polls three different flags.

The first flag, **serial_command_ready**, indicates that a serial command has been received and thus should be handled. In this case, **forward_command()** is called which processes takes any action necessary to fulfill the command.

The second flag, **TOP_flag**, causes the AVR to poll the sonar device attached to the slave AVR chip and send flight data back to the PC via the **send_spam()** function. If the quadrotor is currently in takeoff or landing mode, this interrupt is also where the base thrust of the motors is changed accordingly.

The final flag, **compare_A_flag**, prompts the AVR to poll the gyroscope, magnetometer, accelerometer, and gyroscope in that order. The gyroscope is polled twice to get data for both the x and y rotational axes. When this is done, the roll and pitch are filtered as previously described and the motor values are updated via the **update_motors()** function call.

A pseudocode representation of the main code follows:

```
begin:
  setup serial
  setup devices
  loop forever:
      if serial command is ready:
          handle command
      if 30 Hz flag is set:
          poll sonar
          filter and update altitude
          send flight info to PC
      if 122 Hz flag is set:
          poll gyroscope
          poll magnetometer
          poll accelerometer
          poll gyroscope
          filter and update roll and pitch
          if quadrotor is flying:
              update motors
  end loop
end
```

11 PC side

The PC side setup which we use to communicate with the quadrotor consists of two parts. First we have a Sheaf-Monk utility board connected to an XBee wireless modules. This lets us open a serial port to send and receive data wirelessly. Secondly, we have a PC console application which displays the realtime sensor data readings and saves serial communication messages to a log file. The console application also allows the user to type in and send ASCII based commands to the quadrotor. Typical usage would be something like:

```
BOOT
IDLE // checks motor operations and communications
STOP
BEGIN // enable balancing and altitude controllers
TAKEOFF // enable takeoff controller
LAND
STOP
```

The PC side application is run from the command line and typically appears as illustrated below:

```
Last Command Received:                    Hex:

Compass Heading (degrees): 0.0            Sonar Distance (feet): 0.0
Barometer Temperature: 0.0                Barometer Pressure: 0.0
Barometer Altitude: 0.0
Nunchuck X: 0
Nunchuck Y: 0
Nunchuck Z: 0
Gyroscope Roll (deg/sec): 0               Gyroscope Pitch (deg/sec): 0
Yaw   (deg): 0                            Alt Gain: 0
Pitch (deg): 0                            Pitch Gain: 0
Roll  (deg): 0                            Roll Gain: 0









Type 'q' to quit.
:
Number of bytes sent:  0
Last Command Sent:                        Hex:
```

12 Parts List

Table L.1: Parts List

Part	Make and Model	Quantity	Estimated Cost
Frame	3/4" square aluminum tubing	2 x 55cm	$14.00
Motor	Hextronik DT700	4	43.88
Propeller	10x6 HCB	4	3.48
Prop Saver	4mm w/ Band	4	4.18
ESC	HobbyWing Fly-Fun 18A Brushless ESC	4	79.99
Battery	ZIPPY Flightmax 3000mAh 3S1P 20C	1	12.99
Voltage Detector	Tunigy 3 8S Voltage Detector	1	2.49
Power Distributer	custom	1	-
Sonar Sensor	Parallax Ping))) Ultrasonic	1	29.99
Accelerometer	Wii Nunchuck	1	19.99
Gyroscope	IDG500 Dual 500/s breakout	1	39.95
Magnetometer	3-Axis HMC5883L breakout	1	14.95
UBEC	custom	1	-
5v to 3.3v converter	custom	1	-
Logic Level Converter	1.8 to 5V range, BOB-08745	1	1.95
Master / Slave controller	ATmega 328p	2	9.90
XBee		2	49.90
XBee adapter board	XBee to Serial	2	33.90
Serial Port Adapter	Sheaf-Monk Utility Board	1	10.00
Reset Button	Basic default-off push-buton	1	0.50
Total:			$372.04

13 Schematics

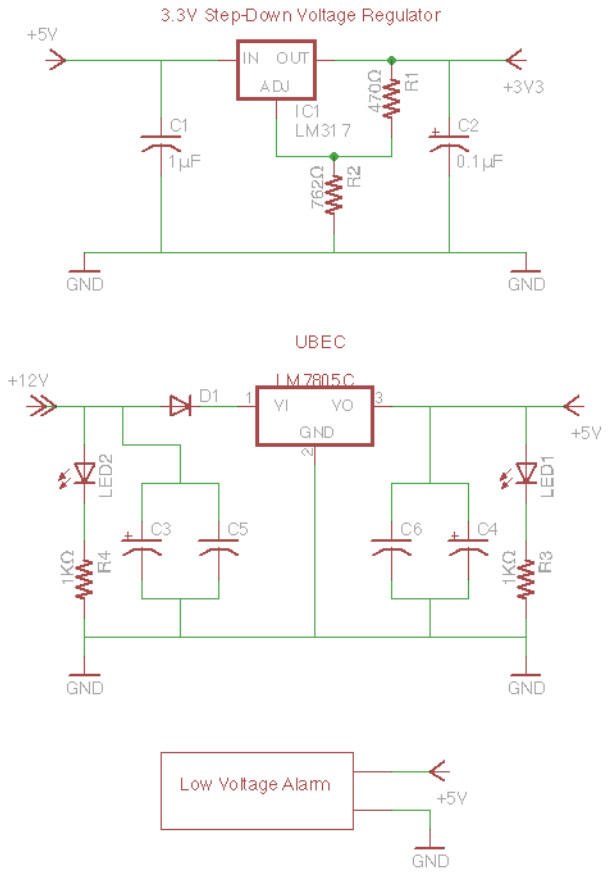

Figure 2: Power Conversion Circuits

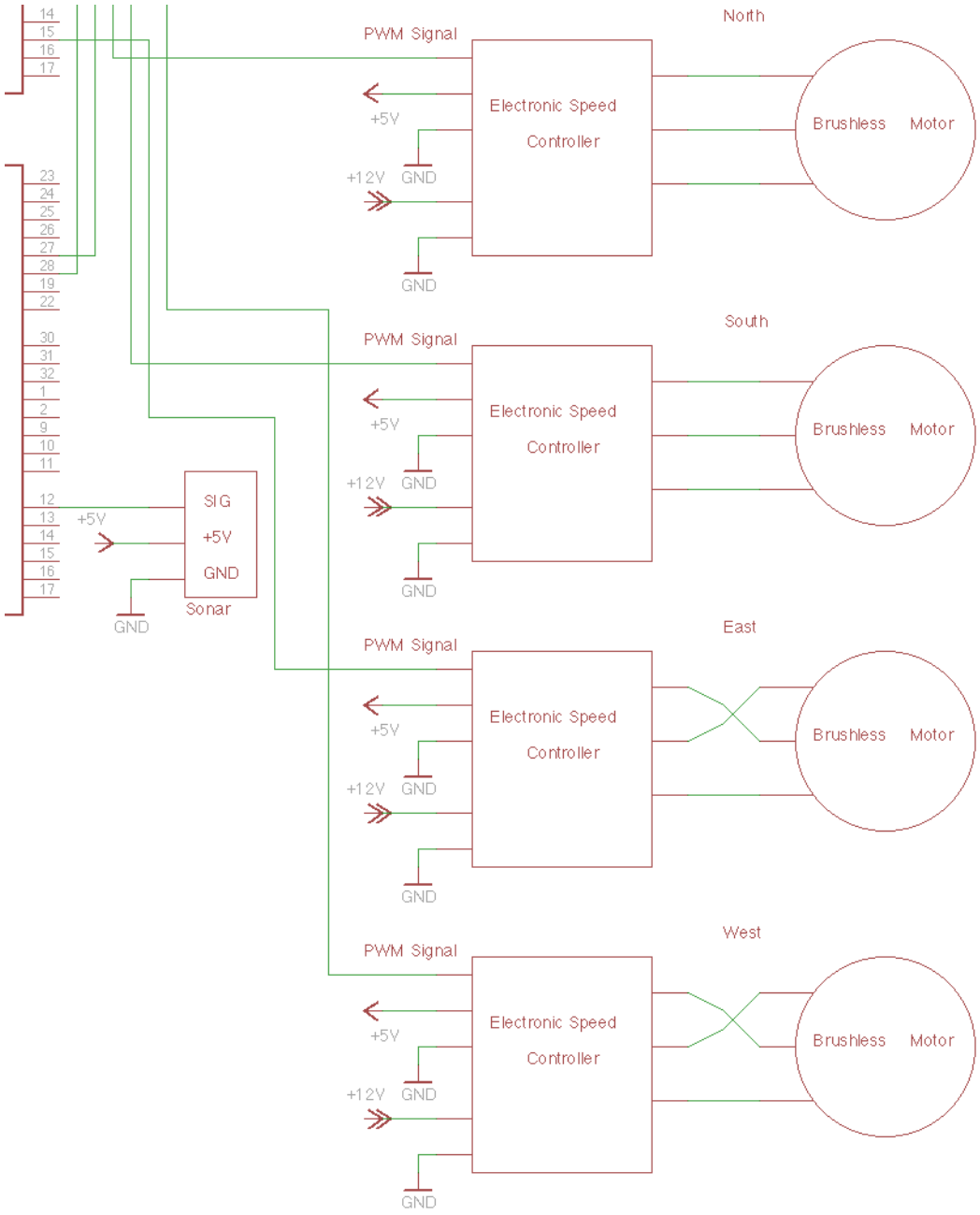

Figure 3: Motor Connections

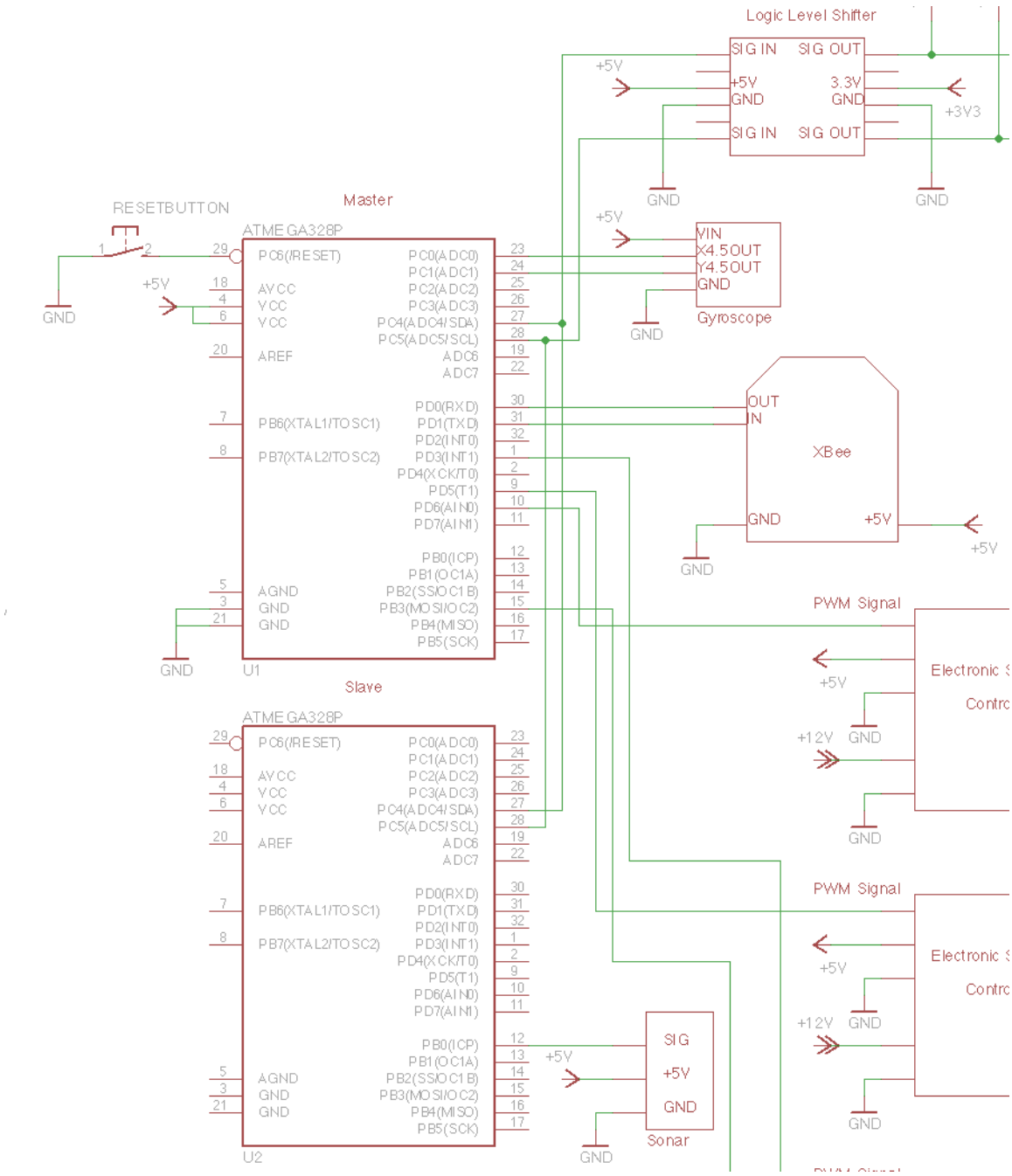

Figure 4: AVR Microcontroller Connections

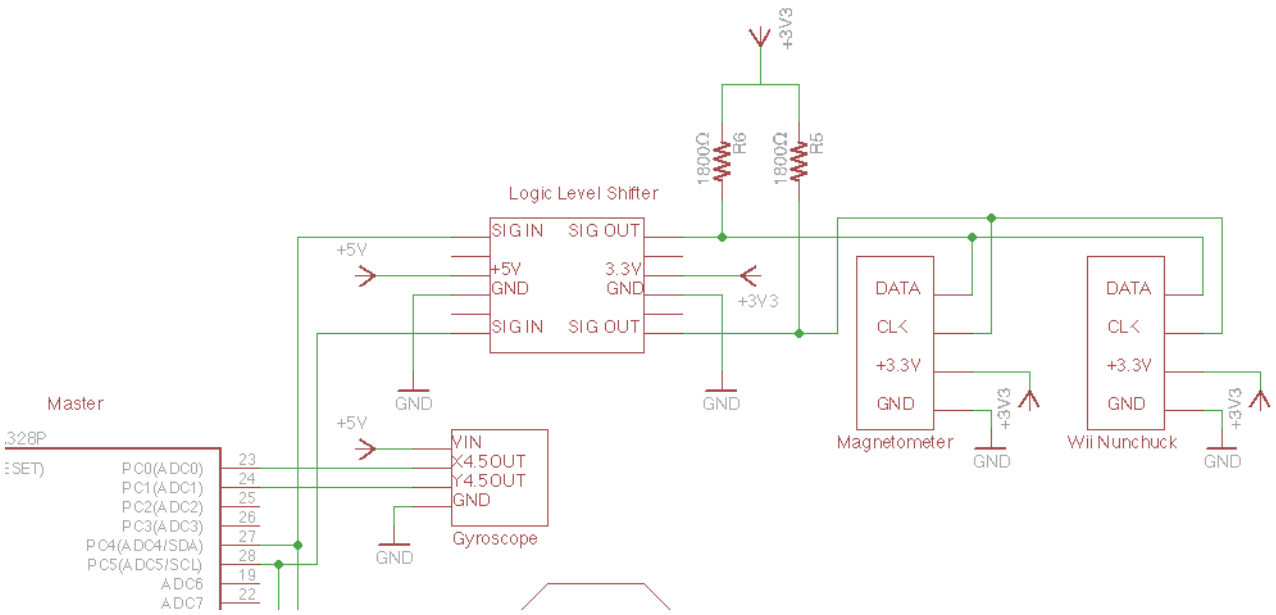

Figure 5: I^2C Bus Circuit

14 Code

We wrote around 8,000 lines of code for this project, so we have not printed the source code. All of the source code can, however, be found in the repository at the following address:

`https://github.com/bilodeau/ECE477PROJECTS/tree/master/quadcopter`

The file directory structure is as follows:

control/	: contains master control code, controllers, and Makefile. UI.h has the big if block that handles user commands.
gps/	: empty (not implemented)
i2c/	: contains headers which implement i2c communication for both the master and slave chips. also contains the main() loop for the slave chip (in slave.c)
motors/	: contains motor control header file, a motors test routine (motors.c, with Makefile)
lib/	: contains PC side code (PCstatus.c, display.h), serial communications code (AVRserial.h, baud.h), delay helper function (delay.h), and sensor data cache and filtering structures (data.h).
lib/devices/	: contains interfacing code for each sensor device used, poll_devices.h and spam.h are used to abstract away the specifics of each device and send data back to the PC, respectively. Note that there are two versions of sonar sensor code, sonar.h is designed to run on a master chip, sonar_slave.h is designed to run on the slave chip.