Autonomous Inductively Charged Quadcopter

by

Sean McQuay

Bryan Murphy

Steven Sumpter

A PROJECT REPORT

Presented to the Electrical and Computer Engineering Faculty of the

MISSOURI UNIVERSITY OF SCIENCE AND TECHNOLOGY

In Partial Fulfillment of the Requirements for

ELECTRICAL ENGINEERING SENIOR PROJECT II

13 May 2014

Cost: $583.77

______________________________ ______________________________

Advisor Instructor

Table of Contents

Executive Summary..3
Project Team..4
Introduction..5
Project Objectives..9
Project Specifications..11
Detailed Design..12
Experimental Results..24
Timeline and Deliverables..27
Budget..29
Appendix:..31

Executive Summary

The Autonomous Inductively Charged Quadcopter team seeks to go above and beyond average projects. While the team realizes that the project and objectives chosen to complete are quite bold, it feels that it has made the best use of the senior design courses.

The overall goal of the project is to deliver a fully autonomous quadcopter capable of full obstacle avoidance based on an array of infrared and ultrasonic sensors covering the airframe. Autonomous navigation will operate in several modes. The quadcopter will be capable of waypoint navigation either via GPS coordinates or a fixed distance input. It will also be capable of a simple "drift mode" in which the quadcopter has no current objective but instead is capable of simply maintaining itself within its environment. Inductive power transfer has also been added as a side objective. The team seeks to transfer roughly 80-100W of power wirelessly to the quadcopter to be available for charging of the onboard batteries.

It is the team's hope that a stable platform has been created upon which, those in the future might branch out. It is the goal that those following the members of this team are able to implement an autonomous function capable of fully utilizing the inductive power transfer by monitoring the battery health and returning to the charger when necessary without user input.

Project Team

Sean McQuay	**Degree:** *Electrical Engineering*	**Contact:** *srm985@mst.edu*

Member Info:
Sean currently serves as the team leader and therefore is responsible for ensuring goals and objectives are being met. He also serves as a liaison between the team and those who require weekly updates on the team's progress. On the physical project, Sean is heading all airborne development. Sean works to implement and revise all autonomous flight functions and all interfacing between the onboard flight management board and the flight controller. Sean is also a collaborator on the inductive power transfer portion of the project, ensuring that goals are met in accordance with the proposed timeline. He will graduate in May of 2014 with a degree in the field of Electrical Engineering. Sean contributed roughly 34% of the work to the overall completion of the project.

Bryan Murphy	**Degree:** *Electrical Engineering*	**Contact:** *bmmy9f@mst.edu*

Member Info:
Bryan's main focus is on the grounded side of the project. Bryan works closely with Dr. Shamsi in development of a wireless power transfer system. Bryan has been working to revise the coil design in the inductive power transfer system to ensure maximum efficiency. Bryan will also develop the airborne portion of the inductive power transfer system. He is responsible for the secondary coil and all electronics required to charge the onboard Lithium-Polymer batteries. Bryan also aids Sean in airborne autonomous testing. He will graduate in May of 2014 with a degree in the field of Electrical Engineering. Bryan contributed roughly 33% of the work to the overall completion of the project.

Steven Sumpter	**Degree:** *Electrical Engineering*	**Contact:** *srs2k9@mst.edu*

Member Info:
Steven is a newcomer to the team. Taking the place of Joseph Strathman, Steven has been working hard to integrate himself into the team. He has been working closely with Bryan on the inductive power transfer system. Steven and Bryan perform regular testing of coil designs with Dr. Shamsi. Steven has experience in the field of switching power supplies and therefore will be a great aid to Bryan in the development of the charging system for the onboard batteries. He will graduate in May of 2014 with a degree in the field of Electrical Engineering. Steven contributed roughly 33% of the work to the overall completion of the project.

Introduction

The normally accepted objective of senior design is to select a project that can be completed within the given timeframe. Students learn about time management and working within deadlines. This is a great concept but it results in students selecting projects that they can easily complete within the timeframe they are given. The projects generally chosen do not benefit the skill sets of the students. This senior design group however chose to take a rather different approach. The aim of this project is to drastically increase the knowledge that the team members possess by the end of the project. Extreme goals were selected and new territories were explored. The members of this team stepped outside of their normal fields in attempts to acquire new skill sets. The members may not complete every goal they initially posed but no matter what, the students will come away with more new knowledge than if no risks were taken. This is essence of senior design. Students should dare to branch out and explore new fields. That is what makes one an engineer, finding a solution no matter the problem.

The broad goal of the team is to create a fully autonomous quadcopter capable of waypoint navigation and obstacle detection via an array of onboard sensors and embedded algorithms. As a side deliverable the team will also implement a wireless power transfer system capable of transferring enough power to charge the onboard batteries. The quadcopter has a span of roughly 60cm and weighs slightly over 1kg including the onboard battery accounting for around 300g of the total weight.

The main focus of the project has been the airborne portion. Most components were purchased off the shelf as time restrictions did not permit design of them nor was it a reasonable endeavor. The main focus of the design was selection of a flight controller. The flight controller on a typical RC quadcopter is responsible for interpreting the signals received from the RC receiver and feeding the motor controllers (known as Electronic Speed Controllers or ESCs) with the correct commands required to maintain stable flight. The flight controller has an array of sensors available to aid it in maintaining stable flight. Without this component, manual flight of a quadcopter would be nearly impossible. The controller accepts four key signals of throttle, pitch yaw, and roll. The flight controller selected was the MultiWii Pro V2.0. This controller was selected because of its low price, open source concept, and because of being open source it has a large support community online. For the time constraints given, it is simply not reasonable to attempt to develop a personal flight controller board. The team started with this board and was able to modify the code to adhere to their specific needs of the project. This board also now supports GPS receiver interface. A GPS receiver was fitted to the quadcopter as well after testing. This receiver amplifies possible autonomous functions. With the use of the GPS receiver the team has been able to implement an autonomous hover function with which the quadcopter is able to maintain its current coordinates within about one meter in any axis. The receiver also offers the possibility of GPS waypoint navigation and a "return to home" function which could simplify location of the charging station.

A secondary onboard microcontroller was added to simplify the build. All autonomous functions could have been implemented on the flight controller board but as it is already heavily taxed on resources by maintaining stable flight, a decision was made to add an additional AVR microcontroller to interpret all additional onboard sensors. This board then generates and feeds six PWM signals to the flight controller board to mimic receiver operation. The team currently relies of five IR rangefinders and one ultrasonic rangefinder for autonomous navigation along with all sensors required by the flight controller. An IR rangefinder is mounted on each of the four legs and serves as obstacle detection in the XY-plane. An additional IR rangefinder is mounted facing downwards to detect possible obstacles.

The second aspect that the team is attempting to tackle is the wireless transfer of power to charge the onboard batteries. While this technology is becoming increasingly popular and more available to the public, the team aimed to increase its skills and manufacture all components themselves. They will design and implement a primary coil on a ground charging station, a secondary coil that will be affixed to the quadcopter and all onboard circuitry on the ground station and the quadcopter required to efficiently transfer power and charge the batteries. Later goals applied to the charging station would be "rough" and "fine" location of the station. By storing the GPS coordinates of the charging station the quadcopter could theoretically return to the station after a low battery condition was sensed onboard. Additional sensors such as a stereovision camera could then be implemented to bring the quadcopter accurately to the charging base and land successfully on the station.

The team has no objectives for the use of the quadcopter. It is instead the goal of the team to create a stable platform that might be used for further experimentation or the implementation of desired functions. Potential uses could be those such as autonomous power line inspection or recording of sporting events without any required user input except for a set of GPS coordinates and a few additional components fitted to the airframe. Once a stable platform has be developed and refined, the possibilities for its use are nearly endless as all of the difficult work has already been completed.

Project Objectives

Objective: **1kg Payload**

The team aimed to generate enough lift to support a payload of 1kg. The quadcopter itself weighs just over 1kg itself so to support an additional 1kg it must produce around 4kg of vertical thrust. During certain maneuvers, two motors may be nearly stopped and therefore the full 2kg would need to be supported by just two of the motors. The team uses four 900KV brushless three-phase motors running at 14.8V paired with 8"x4.5° propellers to generate the thrust required to complete this objective. This requirement is considered important because as stated, the team's broad goal is to develop a stable platform upon which users could implement future operations.

Objective: **Autonomous Position Regulation**

The team felt that the first step towards achieving autonomous navigation was autonomous position regulation. It was necessary that the quadcopter be capable of regulating its own position while stationary before further complicating things with the addition of algorithms used to regulate navigation and obstacle avoidance. Once this goal had been met, the team could move on to those goals which were more demanding.

Objective: **Autonomous Waypoint Navigation**

The team's main focus of the quadcopter project has been to develop and implement autonomous navigation. The final deliverable goal the team seeks to achieve is autonomous navigation. The quadcopter should be able to detect and avoid obstacles while completing a given path or task. The quadcopter should be capable of navigating to set GPS coordinates.

Objective: **Autonomous Obstacle Avoidance**

Paired with the autonomous waypoint navigation goal is an obstacle avoidance requirement. The team set the requirement that a crude obstacle avoidance algorithm be set in place along with a sensor network to support it. The main objective of this goal is the detection and avoidance of large objects such as buildings or humans.

Objective: **Inductive Power Transfer**

As a side deliverable, the team hopes to implement a wireless charging system for the quadcopter. The final objective is to develop a base charging system capable of efficiently transferring 80-100W of power to the quadcopter to charge the onboard Lithium-Polymer batteries. The team will design and wrap all coils themselves along with implementing their own charging circuit on the quadcopter side and the driver circuit required on the base charging station. The team would like to see this goal further developed and automated by future teams. As a final revision, the quadcopter potentially would be able to locate and autonomously land on the charging station when a low battery condition has been detected onboard. This however, is beyond the scope of this team's project.

Project Specifications

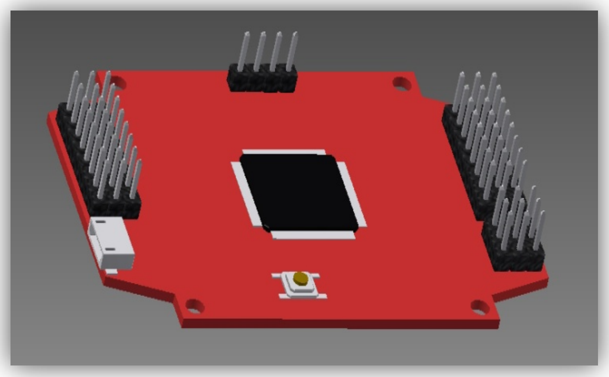
Technical Specifications

Airborne Weight:	*1456 g*
Power Consumption (min):	*500 mA*
Power Consumption (max):	*42.9 A*
Battery Capacity:	*4000 mAh*
Battery Voltage:	*14.8 V*
Battery Construction:	*Lithium-Polymer (4S)*
Generated Thrust (max):	*3.56 kg*
Service Ceiling:	*100 m (User Limited)*
Inductive Power Transfer (min):	*80 W*
Inductive Power Transfer (max):	*100 W*
Flight Control:	*MultiWii Pro V2.0*
Secondary Control:	*AVR ATMega32p*
Flight Controller Software:	*http://tinyurl.com/n5lzt6s*
Interface Controller Software:	*http://tinyurl.com/l6fqltc*

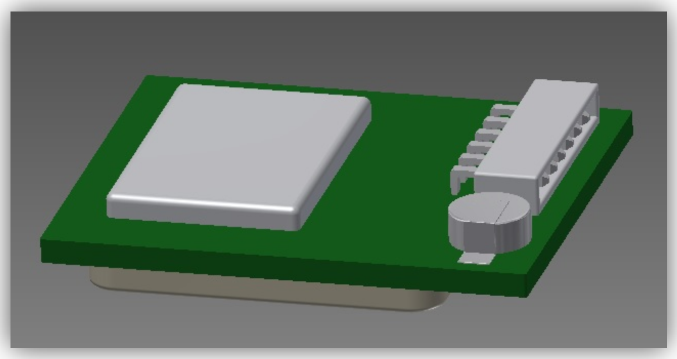
Detailed Design

As the project was broken into two portions, the same will occur here. The airborne components portion of the project will be discussed first and will be followed by the inductive charging portion of the project.

Work on the quadcopter began by calculating all required specifications and ordering the initial parts required to achieve manual flight. The team began with the basic quadcopter frame along with manual flight maintained by the flight controller. The flight controller selected was the MultiWii Pro V2.0. This flight controller was considered a "budget" controller but because it was capable of running open source software, it offered the possibility of infinite customization and the team could implement their own functions onboard the flight controller. The flight controller makes available a barometer, accelerometer, gyrometer, and magnetometer. When loaded in stock form, the software is capable of maintaining basic flight conditions with the help of its onboard sensors. The team began by simply tuning the software to achieve the desired basic flight conditions such as stable flight for the current quadcopter setup and removing any erratic functions.



Once manual flight had been achieved, the team moved on to their first autonomous objective of autonomous position regulation. When the team initially specified this goal, they were uncertain as to how they would achieve the results desired. It was the initial hope that the flight controller itself would provide enough stability to maintain position of the quadcopter through its internal functions. It was immediately discovered that this was not feasible. The team attempted to implement auto-leveling functions which returned the quadcopter to a level position when no roll or pitch inputs were present. This aided in position regulation but the quadcopter still was plagued with drift. While the flight controller does monitor the gyrometer and accelerometer, it is not capable of detecting the subtle changes in position. This can be attributed to the cost of the flight controller as it was considered a low-cost controller. The team then felt that the simplest solution would be to append a GPS module to the project and to modify the software on the flight controller to regulate position based on GPS coordinates. The flight controller already had available additional serial ports and the team simply needed to add a header to the flight controller board to interface with the GPS module. The GPS module selected was the MTK 3329 which simply provides GPS coordinates in the form of serial data with a refresh rate of 10Hz. Obviously GPS is inherently inaccurate but the team felt that this would be the most effective solution with the flight controller



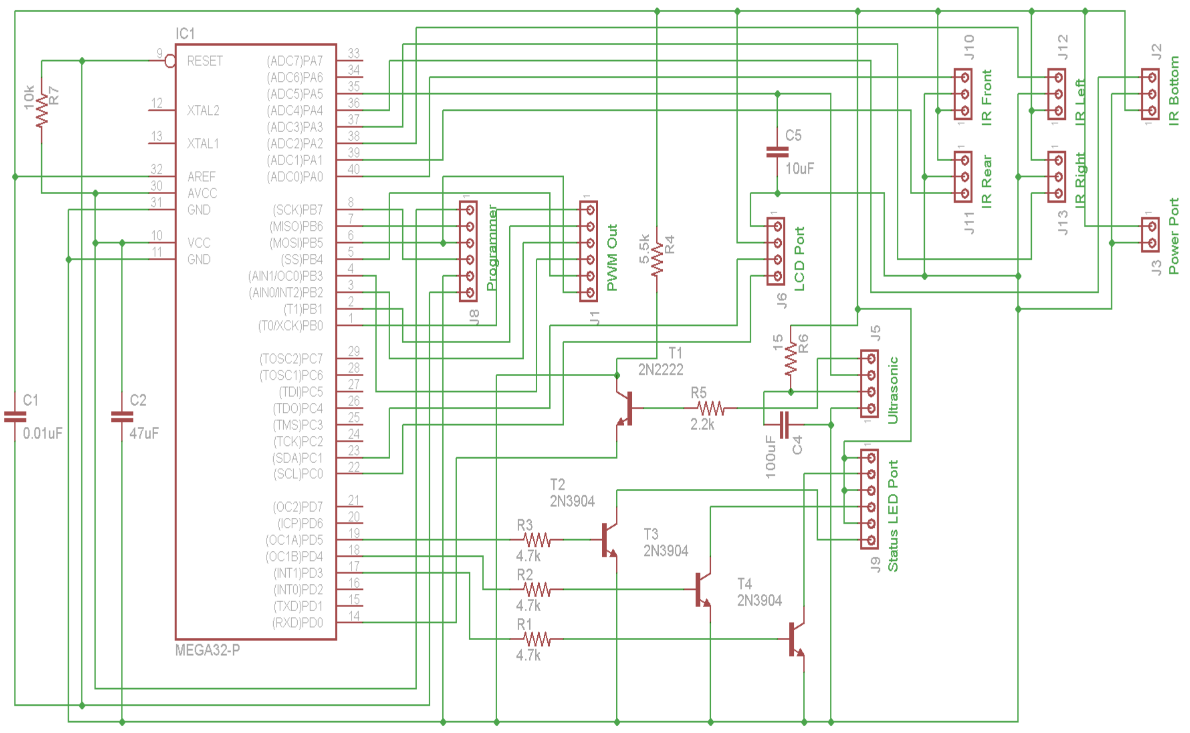
that had been selected. With the addition of the GPS module, the quadcopter was capable of regulating its position within one meter. This accuracy is more than sufficient for the proposed tasks given for the quadcopter to complete. To initiate this function, the interface controller generates a 1.5msec pulse on the AUX1 pin. This is configured in the flight controller to begin the autonomous position regulation function assuming a GPS fix is present. Once initiated the quadcopter simply compares its current GPS coordinates on a five window moving average with those recorded when the function was initialized. Based on the calibration of the PID loop, the quadcopter attempts to return itself to its initial position. The quadcopter also is calibrated with a 100µsec window dead band on throttle, pitch, roll, and yaw while set to autonomous position regulation. If an input signal exceeds this dead band, the quadcopter will respond to the signal and adjust its position and once the inputs return to falling within the dead band, it will record the new GPS coordinates and set them as its target position.

Once the team had developed an accurate position regulation algorithm, work was then shifted towards the next objective of achieving autonomous waypoint navigation. To achieve this task, the team once again turned to the GPS module. Functions were once again implemented on the flight controller board to allow waypoint navigation. The quadcopter currently is capable of accepting waypoints via a GUI. Also definable in the implemented functions is navigation speed both maximum and minimum along with a navigation altitude and maximum service ceiling. To achieve navigation, the quadcopter relies on the onboard magnetometer, barometer, and GPS module on top of already monitoring the accelerometer and gyrometer. The quadcopter begins by orienting itself based on the magnetometer while regulating its altitude with the help of the barometer and GPS module. As the quadcopter begins to navigate, it shifts focus away from the magnetometer as it is able to establish an accurate heading based on the GPS module. The quadcopter is capable of regulating its altitude while navigating and upon reaching a waypoint it is capable of adjusting its altitude to the desired altitude at the waypoint. Once the waypoint has been considered reached, the quadcopter then is able to navigate to the next given waypoint, maintain its current position, or if defined, return to its home position and potentially land if the user so decides.

To achieve autonomous navigation the quadcopter simply compares its current GPS location to the desired coordinates and makes adjustments to throttle, roll, pitch, and yaw based on the calibrated PID loops and the user-limited values such as maximum and minimum navigation speed along with all presets for altitude and once its coordinates are within a user-defined window of the desired coordinates, the waypoint has been considered to have been reached. This function is enabled by generating a 2msec pulse on AUX2 pin.

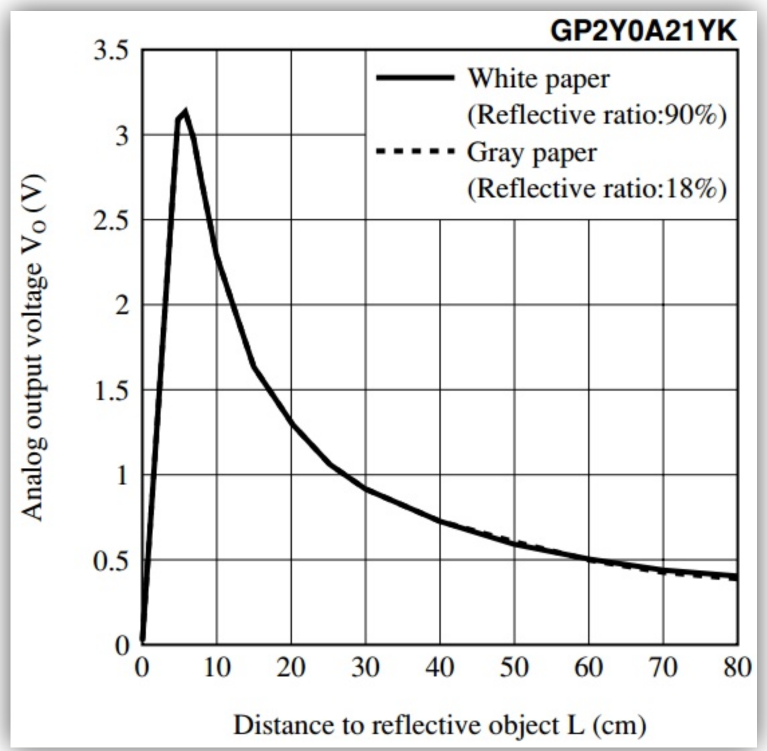
As both the autonomous position regulation and the autonomous waypoint navigation were implemented onboard the flight controller and contained within thousands of lines of code, it was chosen to simply publish the team's edited version of the software and make it available to all of those who might be undertaking similar projects. The software has been updated to work well with the quadcopter designed in this project and has various edits and modifications including the autonomous waypoint navigation and position regulation. Other features include altitude regulation based on barometric pressure which is accurate to within about 0.5m limited by the accuracy of the barometer and inherent difficulties in measuring subtle changes in barometric pressure on a moving device. This function is enabled by generating a 2msec pulse on the AUX1 pin. Also available is a return to home function which when engaged returns the quadcopter to the GPS coordinates recorded when the quadcopter first left the ground. The return to home function maintains the quadcopter at a user-defined altitude above the recorded GPS coordinates. This function currently is not able to be enabled via an AUX pin but all functions are present in the final source code. If it is desired that the quadcopter land, an additional function has been implemented which allows the quadcopter to land at roughly its original start position. Once again, this function relies on GPS data and therefore cannot be considered entirely accurate. The function does however provide a rough position location and with the addition of a stereovision camera, the quadcopter could be capable of locating and landing accurately on its charging station. This function currently is not able to be enabled via an AUX pin but all functions are present in the final source code.

The previous functions were all implemented onboard the flight controller but it was decided that in order to reduce load on the already taxed flight controller, a secondary interface controller would be implemented that would monitor all external sensor inputs and provide high-level control to the flight controller. The interface controller communicates with the flight controller via six PWM signals, emulating a receiver connected to the flight controller. The channels are throttle, pitch, roll, and yaw along with two additional auxiliary pins. The interface controller updates these pins approximately every 50msec. An I2C header was added to the flight controller board with a final objective of I2C communication between the flight controller and interface controller. The flight controller is also capable of PPM communication, reducing the required pins from six to one. The interface controller is responsible for enabling the various functions onboard the flight controller and generates the required PWM signals based on the onboard obstacle avoidance algorithms.



The interface controller currently has onboard hardware to support the connection of five Sharp GP2Y0A21YK IR rangefinders and one Maxbotix EZ0 ultrasonic rangefinder. These sensors are interfaced via five headers present on the board. The IR rangefinders supply the interface controller with a decaying voltage function based on object distance, supporting a maximum range of about 130cm.

The quadcopter uses four of these sensors around the perimeter to detect approaching obstacles. A single function is called and polls from these sensors four sensors. It returns a single byte containing which sensors have detected an object within the user-defined range. As it can be seen from the figure to the right, the voltage function of the IR rangefinders is a polynomial function and therefore it is difficult to determine the measured distance without a lookup table. Instead of this, a calibrated distance value is specified which triggers an object detected flag if an object is within the bounds of the given value.



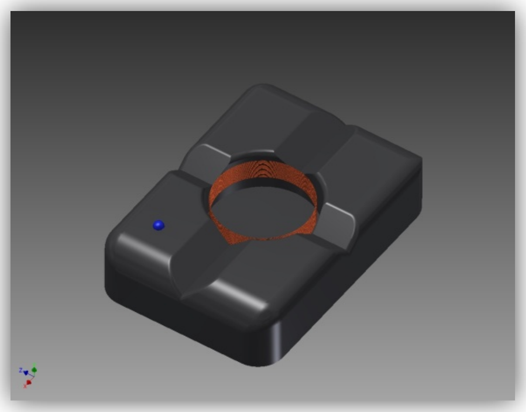
A switch-case function then determines the most effective navigation option based on the given values returned. An additional IR rangefinder is used to detect objects under the quadcopter and can serve as a short-range altimeter as the maximum range is only 130cm. It was originally the goal that the ultrasonic rangefinder would be implemented on the underside of the quadcopter but because of a required sensor calibration every time the sensor powers on it was swapped with the topside IR rangefinder. The team instead currently relies on barometric pressure to determine elevation. This value is zeroed each time the quadcopter is powered on, establishing a ground reading.

The ultrasonic rangefinder header is capable of interfacing with the rangefinder either by feeding the voltage function of the sensor to the ADC of the AVR or by polling the serial output of the rangefinder. Serial distance is transmitted via asynchronous serial with a baud rate of 9600. The data stream consists of the transmission of the ASCII character 'R' followed immediately by three ASCII characters indicating the distance in inches from 6"–255" (15cm–647cm). A simple function is implemented to receive and interpret this data and can be seen listed in the appendix. The main function then uses the value to detect those objects above it in conjunction with the data provided from the underside IR rangefinder. As the header supports and ADC connection, an IR rangefinder could potentially be implemented in place of the ultrasonic rangefinder.

The interface controller board also contains a header which allows the connection of three LEDs. In manual flight mode these LEDs serve to aid in orientation of the quadcopter. In autonomous modes, these LEDs are controlled by the AVR and can be configured as status indicators for the various autonomous functions. To further aid in function diagnostics, an I2C header has been included which allows the interfacing of an LCD module to display various statistics and errors without the need to connect any test equipment. As the LCD is inherently slow, it is only used as a diagnostic tool and cannot be used during actual flight conditions. Finally, the interface controller board features a six pin header which allows the interface controller to controller the flight controller via PWM signals as previously discussed.

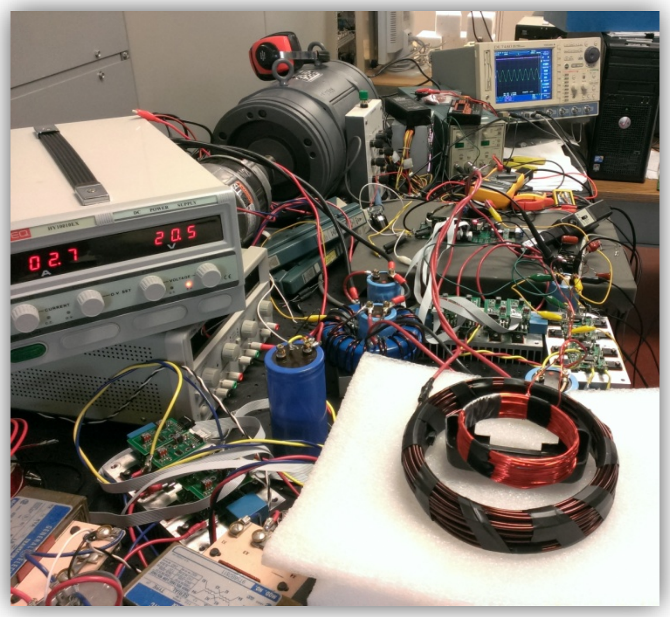
At the heart of the interface controller is an Atmel ATMega32. This microcontroller provides sufficient I/O pins for all currently implemented functions and provides a significant number of additional pins to support the majority of the user-implemented functions in the future. The microcontroller supports most communication protocols such as I2C and serial transmission. As the source code for the interface controller is particularly lengthy, the full source code for the project can be found in the appendix below and has been published publicly for those who would like to reference it in the future. As stated previously, the AVR makes the final call on the quadcopter navigation. It is capable of arming and disarming the flight controller and controls all aspects of the flight plan.

The secondary goal of the team was to develop an inductive power transfer system capable of supplying a significant current transfer capable of charging the onboard batteries. A final goal beyond the scope of this project is to implement battery health monitoring functions and the required hardware to return the quadcopter autonomously to its charging station when required.

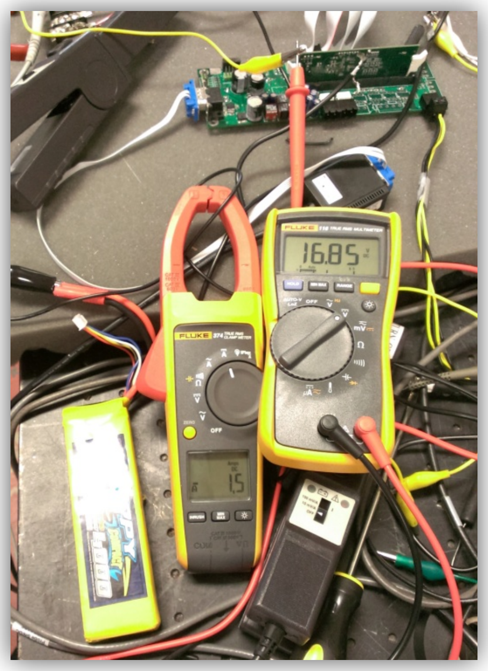


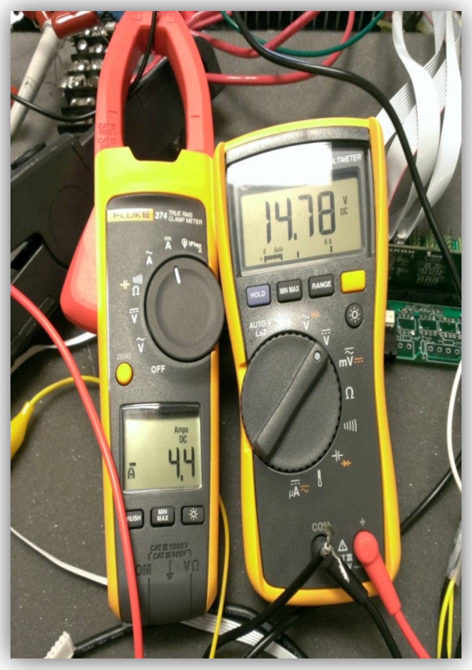
To transfer the required power the team worked to develop a primary terrestrial coil and a secondary airborne coil. As the team coordinated with Dr.

Shamsi, his drivers were selected to feed the primary coil. After much experimentation the team settled on a pulsed DC waveform with a switching frequency of 38.4kHz. This frequency was determined to be the optimum coupling frequency after much experimentation. This frequency was found to work best with the calculated resonance frequency of the coils. Selecting this frequency did result in the highest coupling efficiency but the team did encounter issues as a result. As this frequency is quite high for the quantity of power fed into the primary (~500W), the skin effect observed on the primary was quite significant. It is strongly advised that future revisions of the primary coil be wound with Litz wire and potentially the secondary coil as well. Experimentation with capacitors on the primary coil was also conducted in hopes of reducing the reactive component of power consumed.



As stated previously, the overall objective of the inductive power transfer system was to transfer sufficient power to charge the onboard batteries. The team aimed initially for around 100W of power provided on the secondary coil but this was later determined to be excessive and the team scaled their desired power back to around 60W. The charger is capable of supplying much higher values than this but it simply is not required. As can be seen in the photos, the primary coil is supplied with 2.7A @ 20.5V to generate 1.5A @ 16.85V on the secondary. The battery charging in the photo has nearly reached a full charge and therefore does not sink a large current. A further purely resistive test was conducted and as can be seen, 4.4A @ 14.78V was easily attainable on the secondary coil.





The primary coil consists of 52 turns of 10AWG solid-core wire and is paired with a secondary coil consisting of 20 turns of 18AWG solid-core wire. The calculated coupling efficiency achieved is approximately 30% which is considered quite high without the aid of Litz wire and the team therefore is content with the final efficiency as there were not standards initially specified at the beginning of the project. The most efficient coupling configuration was determined to be when the secondary coil's bottom plane was aligned with the top plane of the primary coil as depicted in the previous photo. The primary coil diameter was 18cm and the secondary coil diameter was finalized at 10cm.

Experimental Results

The team considers all objectives to have been met although some compromises were made in the final specifications of the goals as they did not affect the desired operation of the project. Turning over full autonomous control to a device capable of flight proved to be quite worrisome and therefore many tethered tests prefaced the final "live" tests. A wireless kill switch was always present but due to the nature of PWM communication, the user was not always able to regain manual control of the quadcopter while testing certain autonomous functions when the kill switch was initiated which meant that if the kill switch was to be triggered at a high altitude the quadcopter would simply return to the ground with gravity against it. Initial live tests were therefore performed with extreme caution.

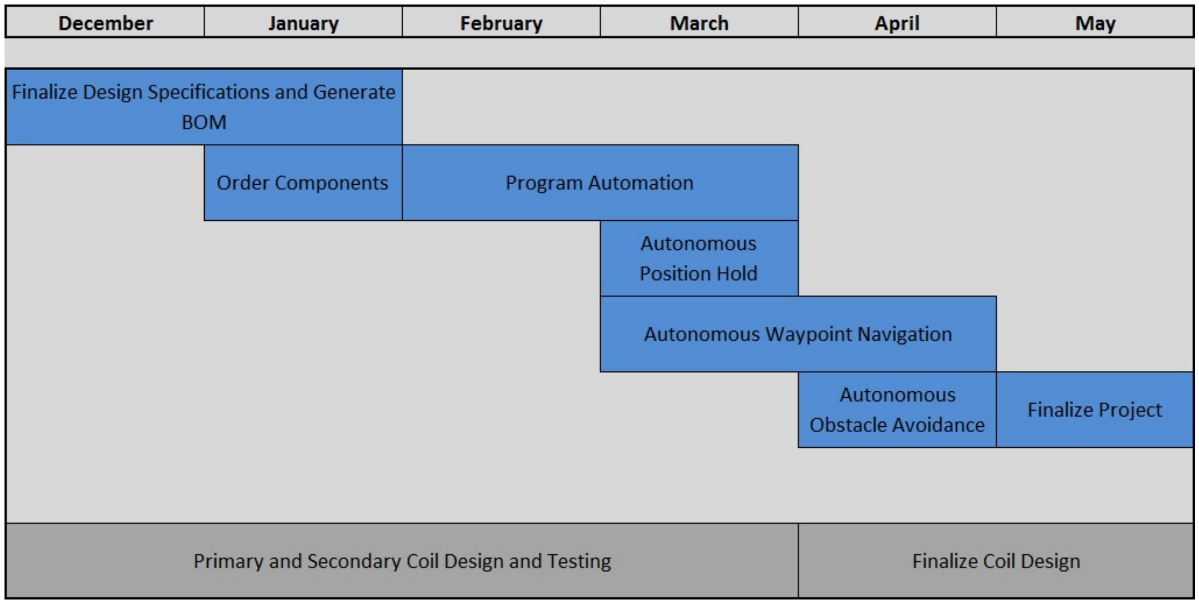
Testing began initially on the autonomous position regulation. Initial tests were simply performed with manual control of the quadcopter while testing the flight controller drift under various trim and PID settings. Once the team appended the GPS module, more complex testing was required but the autonomous position regulation was triggered via an auxiliary input on the flight controller which meant that manual control could be regained if the function did not operate as planned. After tuning, the quadcopter was capable of maintaining a fixed position based on GPS data with a resolution of about one meter in any axis.

Testing of the autonomous waypoint navigation was carried out in a similar fashion to those tests performed for the autonomous position regulation. The function was triggered via an auxiliary input on the flight controller and the user could regain manual control of the quadcopter when needed. Initial issues arose with calibration of the magnetometer along with calculating the current magnet declination for Rolla, MO required for an increased accuracy on the GPS module. A moving five-value average filter was added to the GPS output to help reduce erratic coordinates. The team was able to demonstrate that the quadcopter was able to navigate to given coordinates with a resolution of about one meter in any axis.

Autonomous obstacle avoidance proved to be one of the most difficult functions to implement and test. Testing this function meant that the interface controller was given full control of the pitch and roll components of the quadcopter. To minimize potential disasters, manual control of the throttle was still maintained. Testing began with tethered flights. The quadcopter was suspended and balanced from a cable. The throttle was increased just short of creating a hovering condition for the quadcopter. Simulated obstacles were then placed in front of the various sensors and the quadcopter's responses were recorded. Adjustments were made to the maximum and minimum limiters and the response times were tuned to create adequate responses without overcorrection. Once the team felt adequately safe with the generated algorithms, the team began testing in a completely vacant room. The team tested the quadcopter by requesting a desired altitude and then allowing the interface controller to obtain control of pitch and roll commands. The team worked to tune the algorithms further during this testing and was able to finalize an algorithm capable of responding to obstacles with a significant response time to prevent collisions at any navigation velocity and yet still prevented it from overcorrecting at lower velocities. The team obtained video evidence of the quadcopter autonomously avoiding obstacles.

The team's final testing was fairly simple to complete. As the inductive power transfer system was stationary, the team was able to perform all tests on a stable test platform in a laboratory setting. The team tested various turns ratios of the primary and secondary coils and also experimented with the optimum coupling frequency. To demonstrate the power transfer capability the team initially recorded the power dissipated into a resistive load on the secondary coil and finalized their testing by injecting current into a battery and effectively charging it.

Timeline and Deliverables



As it can be seen, the team began making physical progress towards the project in early December. During the months of December and January, the team worked to finalize all designs and generate a bill of materials. Once everything had been finalized, the team began ordering parts in mid January. As many parts were sourced from international suppliers, shipping times were extended but extended shipping times were a minor concern when considering the overall savings by ordering from international suppliers. Once parts began to arrive, the team began to work on what they titled "Program Automation". During this period, the team worked to initialize the quadcopter. Before implementing additional functions and algorithms the team felt it was essential that the quadcopter complete verifiable manual flight testing. This testing was essential to demonstrate that there were no initial errors with the quadcopter before implementing complex algorithms that certainly would be error prone. Once manual flight testing had been completed, the team set out to complete their first autonomous objective. The team worked to generate algorithms capable of autonomous position regulation during the month of March. As the team relied on GPS for position regulation, they felt that they could also concentrate efforts on autonomous waypoint navigation as this feature also relied on GPS data. The team worked through April on the autonomous waypoint navigation algorithms. As the team was finalizing the autonomous waypoint navigation algorithms they began to transition into working on the final airborne objective of autonomous obstacle avoidance. This proved to be one of the most complex functions to implement and as such, the team was required to increase the number of hours spent during the month of April working on the quadcopter but in the end, was successful in completing this objective as well.

While the team was working on the airborne portion of the quadcopter, secondary work had also begun on the inductive charging system design. The team spent several months in a research and development phase where various coil designs and coupling locations were tested. This testing lasted until about the end of March at which time the team began to finalize the inductive charging design.

The team was able to maintain forward progress throughout the course of the project. For those objectives and deliverables that required less time than allotted, the team was able to transition into working on additional aspects of the projects. At other points, the team was required to increase efforts on the project to meet the objectives required. Through this technique, the team was able to adhere strictly to the schedule which it originally generated and from this, was able to complete successfully all objectives and deliverables.

Budget

The team initially generated a budget projection of around $600. This however was a rough projection and the team did not want to limit itself to a fixed budget. It is accepted that senior design should provide students with an introduction to managing a project budget and staying within the confines of the projected budget but the members of this team felt that setting a budget maximum would limit their productivity. The members instead chose to spend whatever was necessary to realize their goals and ultimately advance their knowledge in the field of electrical engineering. The students received $300 of initial funding from the university and chose to supply whatever additional funds were required to complete the project. Surprisingly, the projected budget cost was above the final expenditure summation.

To accomplish this project on such a minimal budget, the team traded convenience and ease of use for component costs. Many of the airborne components were sourced from international suppliers and therefore required long delivery periods. This required that the team order all components in advance and be certain that they had accounted for all required components. Those components which were required further into the build were sourced from national suppliers. By utilizing this strategy the team was able to minimize the total project cost yet still maintain a comfortable level of convenience throughout the project build.

Qty:	Description:	Unit Price:	Total:		Final Expenditure:
1	MultiWii Pro Flight Controller	$65,36	$65,36		**$583,77**
1	550mm Carbon Fiber Frame	$56,39	$56,39		
2	4000mAh 4S 25C Li-Po Battery	$35,70	$71,40		
4	28-30S 900kv 270W Motor	$15,16	$60,64		
4	Propeller Adapter	$2,09	$8,36		**Initial Projection:**
1	Switching DC Power Supply	$14,34	$14,34		**$600,00**
1	6 Amp Li-Po Charger	$24,07	$24,07		
6	AE-30A 30 Amp Brushless ESC	$12,08	$72,48		
3	8045SF Propeller (4pc)	$2,60	$7,80		
3	8045R Propeller (4pc)	$2,60	$7,80		
3	8045SF Propeller (2pc/2pc)	$3,50	$10,50		
1	8045SF Propeller Red (2pc/2pc)	$3,53	$3,53		
5	JST Jumper Wire	$1,50	$7,50		
1	Hook-Up Wire	$16,95	$16,95		
5	Sharp GP2Y0A02YK0F IR Rangefinder	$14,95	$74,75		
1	Maxbotix LV-EZ0 Ultrasonic Rangefinder	$27,95	$27,95		
1	AVR Programmer	$14,95	$14,95		
2	AVR ATMega32	$8,80	$17,60		
5	ESC Bullet Connector Kit	$4,28	$21,40		

Appendix:

main.c

```
/******************************
 *********AICQC Testbed********
 ******************************/

#include <asf.h>
#include <avr/io.h>
#include <avr/interrupt.h>

#include "main.h"
#include "LED_Disp.h"
#include "Sharp_IR.h"
#include "I2C.h"
#include "LCD_Disp.h"
#include "UART.h"
#include "Ultrasonic.h"

#define F_CPU 1000000
#include <util/delay.h>
#define __DELAY_BACKWARD_COMPATIBLE__


int main(void)
{
       //*******Initialize External Hardware*******
       //LED_init();
       IR_init();
       //I2C_init();
       _delay_ms(10);
       LCD_init();
       UART_init(12);
       //*******************************************


       //***************Define Ports***************
       DDRA &= 0b11000000;
       //PORTA &= 0b11000000;
       DDRB |= 0b00011111;
       PORTB &= 0b11100000;
       DDRD |= 0b00111100;
       PORTD |= 0b00111100;
       //*******************************************


       //*************Define Variables*************
       uint16_t throttle = minPWM;
       uint16_t roll = midRoll;
       uint16_t pitch = midPitch;
       uint16_t yaw = midYaw;
       uint16_t aux1 = minPWM;

       int altitude;
	int prev_alt = 12;

	unsigned char highPitch = 0;
	unsigned char lowPitch = 0;
	unsigned char highRoll = 0;
	unsigned char lowRoll = 0;
	//*******************************************


	//********Establish PWM Safety Timer********
	/*TIMSK = 1 << TOIE0;
	TCNT0 = 0x6C; //Load timer0 counter
	TCCR0 |= (1 << CS02) | (1 << CS00);*/
	//*******************************************

	//sei();	//Enable interrupts after main definitions
	_delay_ms(1000);

	while(1)
	{
		_delay_ms(25);

		altitude = US_Read_ADC();

		if(altitude >= setAltitude && throttle <= maxThrottle && prev_alt >=
altitude)
		{
			throttle += 10;
		}
		else if(altitude <= setAltitude && throttle >= minThrottle && prev_alt
<= altitude)
		{
			throttle -= 10;
		}

		prev_alt = altitude;

		switch(Obstacle_Detect())
		{
			case 0x00:
				if(highRoll)
				{
					roll = midRoll - offset;
					highRoll --;
				}
				else if(lowRoll)
				{
					roll = midRoll + offset;
					lowRoll --;
				}
				else
				{
					roll = midRoll;
				}

				if(highPitch)
				{
```

```
                        pitch = midPitch - offset;
                        highPitch --;
                }
                else if(lowPitch)
                {
                        pitch = midPitch + offset;
                        lowPitch --;
                }
                else
                {
                        pitch = midPitch;
                }
        break;

        case 0x01:
                if(highRoll)
                {
                        roll = midRoll - offset;
                        highRoll --;
                }
                else if(lowRoll)
                {
                        roll = midRoll + offset;
                        lowRoll --;
                }
                else
                {
                        roll = midRoll;
                }

                pitch = midPitch - offset;
                lowPitch = count;
        break;

        case 0x02:
                if(highRoll)
                {
                        roll = midRoll - offset;
                        highRoll --;
                }
                else if(lowRoll)
                {
                        roll = midRoll + offset;
                        lowRoll --;
                }
                else
                {
                        roll = midRoll;
                }

                pitch = midPitch + offset;
                highPitch = count;
        break;

        case 0x03:
                roll = midRoll + offset;
                highRoll = count;
```

```
		if(highPitch)
		{
			pitch = midPitch - offset;
			highPitch --;
		}
		else if(lowPitch)
		{
			pitch = midPitch + offset;
			lowPitch --;
		}
		else
		{
			pitch = midPitch;
		}
	break;

	case 0x04:
		roll = midRoll + offset;
		highRoll = count;

		if(highPitch)
		{
			pitch = midPitch - offset;
			highPitch --;
		}
		else if(lowPitch)
		{
			pitch = midPitch + offset;
			lowPitch --;
		}
		else
		{
			pitch = midPitch;
		}
	break;

	case 0x05:
		roll = midRoll + offset;
		highRoll = count;

		pitch = midPitch - offset;
		lowPitch = count;
	break;

	case 0x06:
		roll = midRoll + offset;
		highRoll = count;

		pitch = midPitch + offset;
		highPitch = count;
	break;

	case 0x07:
		roll = midRoll + offset;
		highRoll = count;
```

```
                if(highPitch)
                {
                        pitch = midPitch - offset;
                        highPitch --;
                }
                else if(lowPitch)
                {
                        pitch = midPitch + offset;
                        lowPitch --;
                }
                else
                {
                        pitch = midPitch;
                }
        break;

        case 0x08:
                roll = midRoll - offset;
                lowRoll = count;

                if(highPitch)
                {
                        pitch = midPitch - offset;
                        highPitch --;
                }
                else if(lowPitch)
                {
                        pitch = midPitch + offset;
                        lowPitch --;
                }
                else
                {
                        pitch = midPitch;
                }
        break;

        case 0x09:
                roll = midRoll - offset;
                lowRoll = count;

                pitch = midPitch - offset;
                lowPitch = count;
        break;

        case 0x0A:
                roll = midRoll - offset;
                lowRoll = count;

                pitch = midPitch + offset;
                highPitch = count;
        break;

        case 0x0B:
                roll = midRoll - offset;
                lowRoll = count;

                if(highPitch)
                {
                        pitch = midPitch - offset;
                        highPitch --;
                }
                else if(lowPitch)
                {
                        pitch = midPitch + offset;
                        lowPitch --;
                }
                else
                {
                        pitch = midPitch;
                }
        break;

        case 0x0C:
                if(highRoll)
                {
                        roll = midRoll - offset;
                        highRoll --;
                }
                else if(lowRoll)
                {
                        roll = midRoll + offset;
                        lowRoll --;
                }
                else
                {
                        roll = midRoll;
                }

                pitch = midPitch + offset;
                highPitch = count;
        break;

        case 0x0D:
                if(highRoll)
                {
                        roll = midRoll - offset;
                        highRoll --;
                }
                else if(lowRoll)
                {
                        roll = midRoll + offset;
                        lowRoll --;
                }
                else
                {
                        roll = midRoll;
                }

                pitch= midPitch - offset;
                lowPitch = count;
        break;

        case 0x0E:
                if(highRoll)
```

```
                        {
                                roll = midRoll - offset;
                                highRoll --;
                        }
                        else if(lowRoll)
                        {
                                roll = midRoll + offset;
                                lowRoll --;
                        }
                        else
                        {
                                roll = midRoll;
                        }

                        pitch = midPitch + offset;
                        highPitch = count;
                break;

                case 0x0F:
                        if(highRoll)
                        {
                                roll = midRoll - offset;
                                highRoll --;
                        }
                        else if(lowRoll)
                        {
                                roll = midRoll + offset;
                                lowRoll --;
                        }
                        else
                        {
                                roll = midRoll;
                        }

                        if(highPitch)
                        {
                                pitch = midPitch - offset;
                                highPitch --;
                        }
                        else if(lowPitch)
                        {
                                pitch = midPitch + offset;
                                lowPitch --;
                        }
                        else
                        {
                                pitch = midPitch;
                        }
                break;
        }
```

```
//**********************************************************
//*******************Write PWM Values*********************
//**********************************************************
TCCR0 |= (1 << CS00);

PORTB |= 0b00000001;
for(int i = 1; i <= ((throttle+calib) >> 4); i++)     //Throttle
{
       TCNT0 = 0x00;
       while(TCNT0 <= 3)
       {
       }
}
PORTB &= 0b11111110;


PORTB |= 0b00000010;
for(int i = 1; i <= ((roll+calib) >> 4); i++)   //Roll
{
       TCNT0 = 0x00;
       while(TCNT0 <= 3)
       {
       }
}
PORTB &= 0b11111101;


PORTB |= 0b00000100;
for(int i = 1; i <= ((pitch+calib) >> 4); i++)  //Pitch
{
       TCNT0 = 0x00;
       while(TCNT0 <= 3)
       {
       }
}
PORTB &= 0b11111011;


PORTB |= 0b00001000;
for(int i = 1; i <= ((yaw+calib) >> 4); i++)    //Yaw
{
       TCNT0 = 0x00;
       while(TCNT0 <= 3)
       {
       }
}
PORTB &= 0b11110111;


PORTB |= 0b00010000;
for(int i = 1; i <= ((aux1+calib) >> 4); i++)   //Aux1
{
       TCNT0 = 0x00;
       while(TCNT0 <= 3)
       {
       }
}
```

```
            PORTB &= 0b11101111;
            //**********************************************************
            //**********************End PWM Write**********************
            //**********************************************************
      }
}
```

main.h

```
#ifndef main_H_
#define main_H_

#define minPWM                    1000
#define midPWM                    1500
#define maxPWM                    2000

#define minThrottle               1200
#define maxThrottle               1400

#define minRoll                   1370
#define midRoll                   1478
#define maxRoll                   1610
#define minPitch                  1306
#define midPitch                  1398
#define maxPitch                  1546
#define minYaw                    1000
#define midYaw                    1500
#define maxYaw                    2000
#define minAux1                   1000
#define midAux1                   1500
#define maxAux1                   2000

#define offset                    170
#define count                     1

#define calib                     172

#define setAltitude               76
#endif
```

UART.c

```
#include <asf.h>
#include <avr/io.h>
#define __DELAY_BACKWARD_COMPATIBLE__
#include <util/delay.h>

#include "UART.h"

void UART_init(int baud)
```

```
{
	UBRRH = (uint8_t)(baud >> 8);
	UBRRL = (uint8_t)(baud);

	UCSRB = ((1<<RXEN) | (1<<TXEN) | (1<<RXCIE));
	UCSRC = ((1<<URSEL) | (1<<UCSZ1) | (1<<UCSZ0));

	UCSRA |= 0x02;
}

void UART_PutChar(unsigned char c)
{
	while(!(UCSRA & (1 << UDRE)));
	UDR = c;
	//while(!(UCSRA & (1 << TXC)));
}

unsigned char UART_ReadChar(void)
{
	while(!(UCSRA & (1 << RXC)));
	return UDR;
}
```

UART.h

```
#ifndef UART_h_
#define UART_h_

void UART_init(int baud);
void UART_PutChar(unsigned char c);
unsigned char UART_ReadChar(void);

#endif
```

I2C.c

```
/************************************************************************
******************************I2C Control*********************************
*************************************************************************/
#include <inttypes.h>
#include <compat/twi.h>

#include "I2C.h"


/* define CPU frequency in Mhz here if not defined in Makefile *ifconfig
#ifndef F_CPU
#define F_CPU 4000000UL
#endif
```

```
*/
/* I2C clock in Hz */
#define SCL_CLOCK  10000L


/*************************************************************************
 Initialization of the I2C bus interface. Need to be called only once
*************************************************************************/
void I2C_init(void)
{
  /* initialize TWI clock: 100 kHz clock, TWPS = 0 => prescaler = 1 */

  TWSR = 0;                         /* no prescaler */
  TWBR = ((1000000/SCL_CLOCK)-16)/2;  /* must be > 10 for stable operation */

}/* I2C_init */


/*************************************************************************
  Issues a start condition and sends address and transfer direction.
  return 0 = device accessible, 1= failed to access device
*************************************************************************/
unsigned char I2C_Start(unsigned char address)
{
    uint8_t   twst;

       // send START condition
       TWCR = (1<<TWINT) | (1<<TWSTA) | (1<<TWEN);
       //TWCR = 0b10100100;
       PORTC |= 0x03;       //Set internal pull-up resistors
       // wait until transmission completed
       while(!(TWCR & (1<<TWINT)));

       // check value of TWI Status Register. Mask prescaler bits.
       twst = TW_STATUS & 0xF8;
       if ( (twst != TW_START) && (twst != TW_REP_START)) return 1;

       // send device address
       TWDR = address;
       TWCR = (1<<TWINT) | (1<<TWEN);

       // wail until transmission completed and ACK/NACK has been received
       while(!(TWCR & (1<<TWINT)));

       // check value of TWI Status Register. Mask prescaler bits.
       twst = TW_STATUS & 0xF8;
       if ( (twst != TW_MT_SLA_ACK) && (twst != TW_MR_SLA_ACK) ) return 1;

       return 0;

}/* I2C_Start */


/*************************************************************************
 Issues a start condition and sends address and transfer direction.
 If device is busy, use ack polling to wait until device is ready
 Input:   address and transfer direction of I2C device
*************************************************************************/
unsigned char I2C_Start_Wait(unsigned char address)
{
    uint8_t   twst;


    while ( 1 )
    {
           // send START condition
           TWCR = (1<<TWINT) | (1<<TWSTA) | (1<<TWEN);

       // wait until transmission completed
       while(!(TWCR & (1<<TWINT)));

       // check value of TWI Status Register. Mask prescaler bits.
       twst = TW_STATUS & 0xF8;
       if ( (twst != TW_START) && (twst != TW_REP_START)) continue;

       // send device address
       TWDR = address;
       TWCR = (1<<TWINT) | (1<<TWEN);

       // wail until transmission completed
       while(!(TWCR & (1<<TWINT)));

       // check value of TWI Status Register. Mask prescaler bits.
       twst = TW_STATUS & 0xF8;
       if ( (twst == TW_MT_SLA_NACK )||(twst ==TW_MR_DATA_NACK) )
       {
           /* device busy, send stop condition to terminate write operation */
               TWCR = (1<<TWINT) | (1<<TWEN) | (1<<TWSTO);

               // wait until stop condition is executed and bus released
               while(TWCR & (1<<TWSTO));

           continue;
       }
       if( twst != TW_MT_SLA_ACK) return 1;
       break;
     }
return 0;
}/* I2C_Start_Wait */


/*************************************************************************
 Issues a repeated start condition and sends address and transfer direction

 Input:   address and transfer direction of I2C device

 Return:  0 device accessible
          1 failed to access device
*************************************************************************/
unsigned char I2C_Rep_Start(unsigned char address)
{
    return I2C_Start( address );
}/* I2C_Rep_Start */


/*************************************************************************
 Terminates the data transfer and releases the I2C bus
*************************************************************************/
void I2C_Stop(void)
{
    /* send stop condition */
       TWCR = (1<<TWINT) | (1<<TWEN) | (1<<TWSTO);

       // wait until stop condition is executed and bus released
       while(TWCR & (1<<TWSTO));

}/* I2C_Stop */


/*************************************************************************
  Send one byte to I2C device

  Input:    byte to be transfered
  Return:   0 write successful
            1 write failed
*************************************************************************/
unsigned char I2C_Write(unsigned char data)
{
    uint8_t   twst;
       // send data to the previously addressed device
       TWDR = data;
       TWCR = (1<<TWINT) | (1<<TWEN);

       // wait until transmission completed
       while(!(TWCR & (1<<TWINT)));

       // check value of TWI Status Register. Mask prescaler bits
       twst = TW_STATUS & 0xF8;
       if( twst != TW_MT_DATA_ACK) return 1;
       return 0;

}/* I2C_Write */


/*************************************************************************
 Read one byte from the I2C device, request more data from device

 Return:  byte read from I2C device
*************************************************************************/
unsigned char I2C_ReadAck(void)
{
       TWCR = (1<<TWINT) | (1<<TWEN) | (1<<TWEA);
       while(!(TWCR & (1<<TWINT)));

    return TWDR;

}/* I2C_ReadAck */
```

```
/*************************************************************************
 Read one byte from the I2C device, read is followed by a stop condition

 Return:  byte read from I2C device
*************************************************************************/
unsigned char I2C_ReadNak(void)
{
       TWCR = (1<<TWINT) | (1<<TWEN);
       while(!(TWCR & (1<<TWINT)));

    return TWDR;

}/* I2C_ReadNak */
```

I2C.h

```
#ifndef I2C_H
#define I2C_H

#include <avr/io.h>

//#define I2C_READ    1
//#define I2C_Write   0


extern void I2C_init(void);
extern void I2C_Stop(void);
extern unsigned char I2C_Start(unsigned char addr);
extern unsigned char I2C_Rep_Start(unsigned char addr);
extern unsigned char I2C_Start_Wait(unsigned char addr);


/**
 @brief Send one byte to I2C device
 @param    data  byte to be transfered
 @retval   0 write successful
 @retval   1 write failed
 */
extern unsigned char I2C_Write(unsigned char data);


/**
 @brief    read one byte from the I2C device, request more data from device
 @return   byte read from I2C device
 */
extern unsigned char I2C_ReadAck(void);

/**
 @brief    read one byte from the I2C device, read is followed by a stop condition
 @return   byte read from I2C device
 */
extern unsigned char I2C_ReadNak(void);
```

```
/**
 @brief    read one byte from the I2C device

 Implemented as a macro, which calls either I2C_ReadAck or I2C_ReadNak

 @param    ack 1 send ack, request more data from device<br>
               0 send nak, read is followed by a stop condition
 @return   byte read from I2C device
 */
extern unsigned char I2C_read(unsigned char ack);
#define I2C_read(ack)  (ack) ? I2C_ReadAck() : I2C_ReadNak();


/**@}*/
#endif
```

LCD_Disp.c

```
/****************************************
       Control of 20x4 I2C LCD
****************************************/

#include <asf.h>
#include <avr/io.h>
#define __DELAY_BACKWARD_COMPATIBLE__
#include <util/delay.h>
#include "LCD_Disp.h"
#include "I2C.h"


void LCD_init()
{
       I2C_init();   //Initialize I2C protocol

       //*****Init Three Times*****
       LCD_Write(0x03, 0);
       _delay_ms(5);
       LCD_Write(0x03, 0);
       _delay_us(150);
       LCD_Write(0x03, 0);
       _delay_us(150);

       LCD_Write(0x02, 0);

       LCD_Write(0b00101000, 0);

       LCD_Write(0b00001000, 0);

       LCD_Write(0b00000001, 0);

       LCD_Write(0b00001100, 0);  //Set cursor options
}

void LCD_Write(unsigned char data, uint8_t RS_En)
{
	I2C_Start(LCD_Address);

	//*****Write Upper Four Bits*****
	I2C_Write((data & 0xF0) | Backlight | RS_En);
	togle_EN(data & 0xF0, RS_En);

	//*****Write Lower Four Bits*****
	I2C_Write(((data << 4) & 0xF0) | Backlight | RS_En);
	togle_EN((data << 4) & 0xF0, RS_En);

	I2C_Stop();
}

void togle_EN(uint8_t data, uint8_t RS_En)
{
	I2C_Write(data | Backlight | En | RS_En);
	_delay_us(1);
	I2C_Write((data | Backlight | RS_En) & ~En);
	_delay_us(50);
}

void LCD_Clear()
{
	LCD_Write(0x01, 0);
}

void LCD_Home()
{
	LCD_Write(0x02, 0);
	_delay_ms(2);
}

void LCD_Shift(uint8_t shift, uint8_t direction, uint8_t quantity)
{
	while(quantity)
	{
		LCD_Write(0x10 | shift | direction, 0);
		_delay_us(50);		//required 37usec pause
		quantity--;
	}
}

void LCD_On(uint8_t valid)
{
	LCD_Write(((valid << 2) | 0x08) & 0x0C, 0);
	_delay_us(50);		//required 37usec pause
}
```

LCD_Disp.h

```
#ifndef LCD_Disp_h_
#define LCD_Disp_h_

#define LCD_Address                         0x4E   //I2C address location

#define En                                  0x04
#define Backlight                           0x08
#define RS                                  0x01


//******LCD Shift*****
#define Right_Cur                           0x04
#define Left_Cur                            0x00
#define Disp_Shift                          0x08
#define Cur_Shift                           0x00


void LCD_init(void);
void LCD_Write(unsigned char data, uint8_t RS_En);
void togle_EN(uint8_t data, uint8_t RS_En);
void LCD_Clear(void);
void LCD_Home(void);
void LCD_Shift(uint8_t shift, uint8_t direction, uint8_t quantity);
void LCD_On(uint8_t valid);


#endif
```

Ultrasonic.c

```
#include <asf.h>
#include <avr/io.h>
#define __DELAY_BACKWARD_COMPATIBLE__
#include <util/delay.h>

#include "Ultrasonic.h"
#include "main.h"
#include "UART.h"
#include "LCD_Disp.h"


ISR(USART_RXC_vect)
{
	char US_Data[3];
	while(UART_ReadChar() != 'R'){}

	US_Data[0] = UART_ReadChar();
	US_Data[1] = UART_ReadChar();
	US_Data[2] = UART_ReadChar();

	/*for(int i = 0; i <= 2; i++)
	{
		LCD_Write(US_Data[i], RS);
```

```
        }
        LCD_Shift(Cur_Shift, Left_Cur, 3);*/

        //altitude = (US_Data[2] - '0') + (10 * (US_Data[1] - '0')) + (100 *
(US_Data[0] - '0'));
}

char US_Read()
{
        //PORTD |= 0b00000100;
        _delay_ms(500);

        char US_Data[3];
        while(UART_ReadChar() != 'R'){}

        US_Data[0] = UART_ReadChar();
        US_Data[1] = UART_ReadChar();
        US_Data[2] = UART_ReadChar();

        for(int i = 0; i <= 2; i++)
        {
                LCD_Write(US_Data[i], RS);
        }
        LCD_Shift(Cur_Shift, Left_Cur, 3);

        //PORTD &= 0b11111011;

        return (US_Data[2] - '0') + (10 * (US_Data[1] - '0')) + (100 * (US_Data[0] -
'0'));
}

int US_Read_ADC()
{
        int poll_factor = 3;
        int adc_result = 0;

        ADMUX = (ADMUX & 0xF8)|0x05; //Define ADC channel (maybe 0xF8?)

        for(int i=1;i<=poll_factor;i++)
        {
                ADCSRA |= (1<<ADSC); //Set ADC flag
                while(ADCSRA & (1<<ADSC)); //Wait for ADC flag
                adc_result += ADC;
        }
        adc_result = (adc_result/poll_factor);
        return(adc_result);
}
```

Ultrasonic.h

```
#ifndef Ultrasonic_h_
#define Ultrasonic_h_

```

```
char US_Read(void);
int US_Read_ADC(void);

#endif
```

Sharp_IR.c

```
#include <asf.h>
#include <avr/io.h>
#define __DELAY_BACKWARD_COMPATIBLE__
#include <util/delay.h>
#include <avr/interrupt.h>
#include "Sharp_IR.h"


void IR_init(void)
{
	ADMUX = 0x40; //Use AVCC (5V ref.) w/external AREF cap
	ADCSRA = 0x87;		//Define ADC register
}

uint16_t IR_Read(uint8_t ch)
{
	int poll_factor = 5;
	uint16_t adc_result = 0;
	
	ch &= 0x07;   //Force channel to be 0-7 (safety)
	ADMUX = (ADMUX & 0xF8)|ch; //Define ADC channel (maybe 0xF8?)
	
	for(int i=1;i<=poll_factor;i++)
	{
		ADCSRA |= (1<<ADSC);//Set ADC flag
		while(ADCSRA & (1<<ADSC)); //Wait for ADC flag
		adc_result += ADC;
	}
	return(adc_result /= poll_factor);
}

char Obstacle_Detect(void)
{
	int poll_factor = 3;
	uint16_t adc_result[4] = {0,0,0,0};
	uint8_t result = 0;
	for(int i = 0; i <= 3; i++)
	{
		ADMUX = (ADMUX & 0xF8) | i;
		for(int j = 1; j <= poll_factor; j++)
		{
			ADCSRA |= (1<<ADSC);
			while(ADCSRA & (1<<ADSC));
			adc_result[i] += ADC;
		}
		adc_result[i] /= poll_factor;
	}
	for(int i = 0; i <= 3; i++)
	{
		if(adc_result[i] >= maxDist)
		{
			result |= 1 << i;
		}
	}
	return result;
}
```

Sharp_IR.h

```
#ifndef Sharp_IR_h_
#define Sharp_IR_h_

//*******Define sensor channels*******
#define Forward_IR                 0
#define Rear_IR                    1
#define Left_IR                    2
#define Right_IR                   3
#define Top_IR                     4
#define Bottom_IR                  5

#define maxDist                    115    //130 = 75cm
#define minDist                    268

void IR_init(void);
uint16_t IR_Read(uint8_t ch);
char Obstacle_Detect(void);

#endif
```