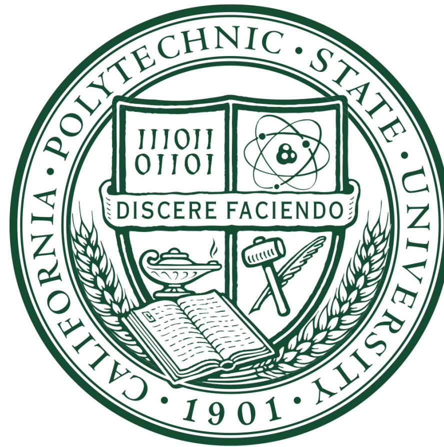
FOAMLAND SECURITY

DOMINIC DOTY & COLTON CRIVELLI



An Automated Nerf Targeting System

Mechanical Engineering
Dr. John Ridgely
California Polytechnic State University
San Luis Obispo

CONTENTS

i DESIGN 1
1 INTRODUCTION 2
1.1 Background Information 2
1.2 Design Motivation 2
1.3 Target Customer Description 3
2 SPECIFICATIONS 4
3 HARDWARE DESIGN 5
3.1 Brainstorming 5
3.2 Solid Modelling 6
3.3 Simulation 8
3.4 Optical Assembly 9
3.5 Electrical Hardware 11
3.5.1 Microcontroller 11
3.5.2 Encoder Front End 12
3.5.3 Motor Driver 13
3.6 Manufacturing 14
4 SOFTWARE DESIGN 16
4.1 Ideation 16
4.2 Task Diagram 16
4.2.1 Task Seekill Finite State Machine 17
4.2.2 Task Control FSM 20

ii RESULTS 22
5 RESULTS 23
5.1 Specification Evaluation 23
5.2 V2 23
5.2.1 Mechanical Design 24
5.2.2 Electrical Design 24
5.2.3 Sensor Design 25
5.2.4 Program Design 25

iii APPENDIX 26
A CODE 27
A.1 Task Seekill 27
A.2 Task Control 37
A.3 Class PID 42
A.4 Class Encoder 45
A.5 Class Motor 51
A.6 Class ADC 55

LIST OF FIGURES

Figure 1 Foamland Security 2
Figure 2 Brainstorming Design Sketch 6
Figure 3 Isometric Solid Model Rendering 7
Figure 4 Bottom Solid Model Rendering 8
Figure 5 Simulink Closed Loop Model 8
Figure 6 Simulink Closed Loop Step Response 9
Figure 7 Optical Assembly Attached to Nerf Gun 10
Figure 8 Line Scanner Phototransistor Assembly 10
Figure 9 Demonstration of the Optical Assembly 11
Figure 10 Optical Encoder Quadrature Head 12
Figure 11 Comparator Noise Acceptance 13
Figure 12 Encoder Signal Conditioning Circuitry 13
Figure 13 Laser Cutter Manufacturing 14
Figure 14 Completed Foamland Security Assembly 15
Figure 15 Ideation Task Diagram 17
Figure 16 Task Diagram for Foamland Security 17
Figure 17 Finite State Machine of Task Seekill 20
Figure 18 Finite State Machine of Task Control 21

ACRONYMS AND ABBREVIATIONS

IC	Integrated Circuit
IR	Infrared
FSM	Finite State Machine
PWM	Pulse Width Modulation
ALTI	Altitude Axis
AZIM	Azimuth Axis
RTOS	Real Time Operating System
ADC	Analog to Digital Converter
LPF	Low Pass Filter
CP	California Polytechnic

Part I

DESIGN

1 INTRODUCTION

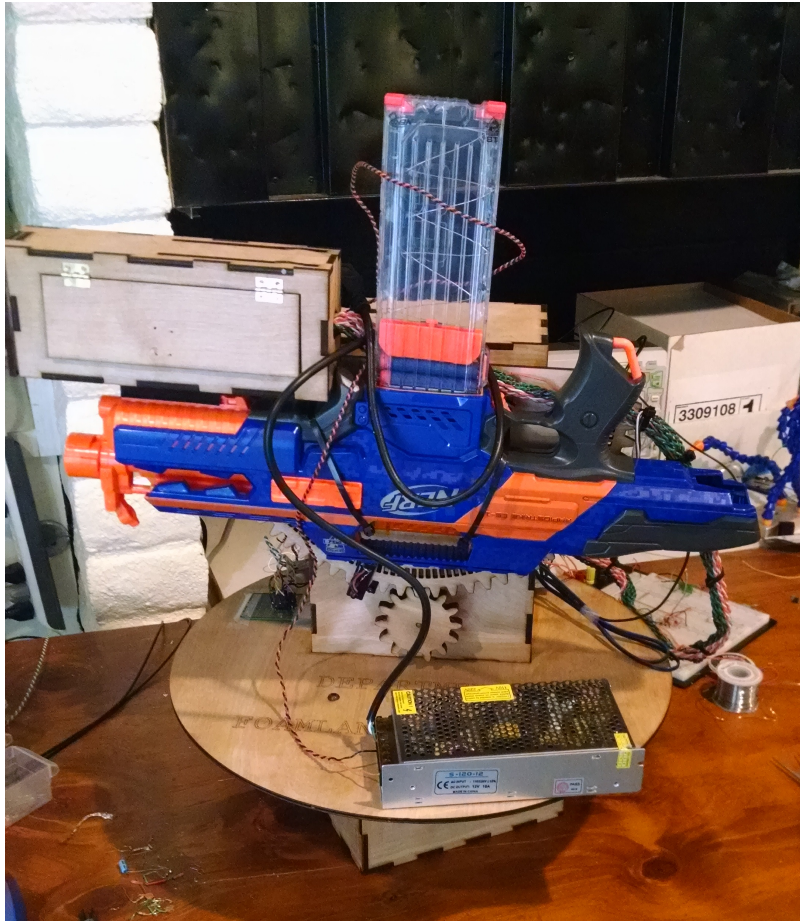


Figure 1: Foamland Security.

1.1 BACKGROUND INFORMATION

Foamland Security represents one creation out of many spawned from the Winter 2015 ME405 Learn By Dueling Challenge. The basic goal is to create an autonomously controlled, IR sensing, PID controlled Nerf turrent. The primary objective is to be able to hit an IR lamp quicker than any single opponent.

1.2 DESIGN MOTIVATION

A primary difference between Foamland Security and other dueling appartus is that Foamland Security was designed to be kept by its

creators. As a result, instead of using the standard ME405 board for brains, Foamland Security runs off an Atmega2560, as well as self-purchased brushed DC gear motors, custom encoders, and a 12V power supply, much of which can be seen in Figure 1.

1.3 TARGET CUSTOMER DESCRIPTION

The primary customers are ourselves because we are keeping Foamland Security and we are the individuals that benefited from its creation by learning. As a result much of what was done and the approach to how it was done was decided by what we felt we would benefit from learning how to do.

Dr. Ridgely represents a secondary customer because he gives us a grade, which is supposed to matter to us.

2 SPECIFICATIONS

The table below depicts the specifications that Foamland Security was intended to meet:

Specification	Justification
Must have PID control over two axes	Two axis control is necessary to successfully target the heat lamp
Must sense IR	IR sensing is necessary to be able find the heat lamp
Must use a lens	Specified by Dr. Ridgely to increase chances of success
Must use gears or belts	Specified by Dr. Ridgely
Must use a RTOS	RTOS is an appropriate tool for this task and a core concept of ME405
Must use tasks	Tasks are a core concept of ME405
Must operate autonomously	A primary objective of learn by dueling is to have a closed loop system
Must autonomously locate, shoot at, and hit target	Overall goal of Learn By Dueling

3 HARDWARE DESIGN

The mechanical design of "Foamland Security" was chosen with simplicity, robustness, and performance in mind. The apparatus achieves this through use of simple, laser cut plywood pieces and brushed DC gear motors.

3.1 BRAINSTORMING

Foamland Security needed to orient itself to point any direction in 3d space. This requirement immediately pointed the design development in the direction of telescopes. Telescopes are easily oriented to point at any area of the sky accurately. Most telescope designs feature an altitude mechanism, that moves the telescope from the horizon up towards the sky, which is mounted on an azimuth mechanism that sweeps the telescope across the horizon. To ease the loads on the mechanism, the center of gravity of the telescope is almost always located at the center of rotation of the altitude mechanism. Thus only requiring the mechanism to overcome the moment of inertia of the assembly. We considered this arrangement ideal for the Nerf turret.

Since the apparatus would only be required to rotate through less than 180° in each axis, high torque was considered to be of the utmost priority. To achieve this, the final drive on each axis is geared. Multiple options for gearing were considered, but only belts and spur gears were considered appropriate options. Spur gears require careful alignment, are noisy, and expensive. On the other hand, belts are less sensitive to misalignment, are quiet, and can use smaller diameter pulleys due to smaller tooth size. Belts are the ideal choice, but due to time considerations, we could not order closed timing belts and instead opted for spur gears.

The target for Foamland Security was a small automotive lamp. We selected infrared phototransistors to locate the lamp. In competition, the gun would start facing 180 degrees away from the lamp, while the lamp was at an unknown altitude. In order to sense and target the lamp as quickly as possible, our initial brainstorming yielded a line scanner that moved independently of the rest of the mechanism. The scanner would spin at around 60 rpm taking readings on a line of 16 phototransistors. After a calibration to background levels, the scanner would find any differences in its IR surroundings almost instantly. It would then transmit the location of the lamp to the motion control module, which would target the lamp as quickly as possible. A sketch of the original design is shown in Figure 2.

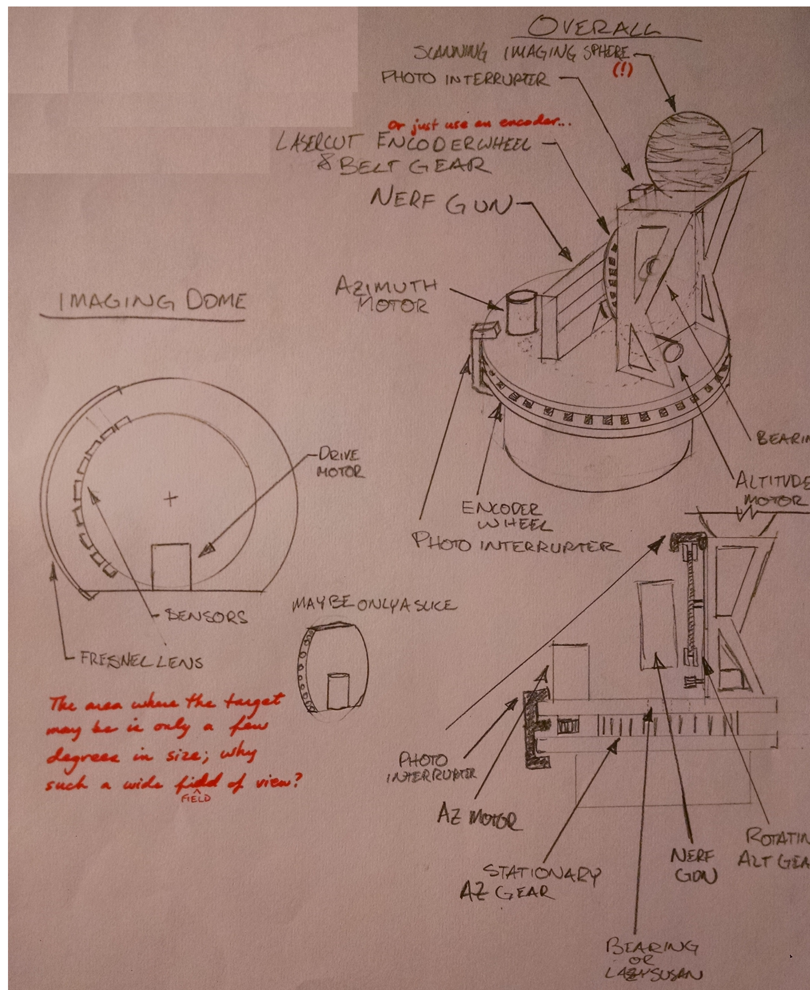


Figure 2: Brainstorming Design Sketch.

3.2 SOLID MODELLING

To fully understand the proportions and layout of the assembly, we made a solid model. The model includes all laser cut panels modelled as solid pine. This proves to be a good approximation of the 5mm plywood used to build the real turret. The Nerf gun was roughly modelled from a scaled picture, and its center of gravity and weight checked against the real gun to ensure accuracy. The turret assembly was laser cut from three sheets of 18" by 32" 5mm underlayment. This material was selected because of its consistent thickness, pleasing surface finish, and laser compatibility.

The assembly was designed to be assembled purely with wood glue using tab and slot construction. This yields a strong and low cost structure. Figure 3 shows a Solidworks rendering of the design. The altitude assembly of the gun was designed with the gun's center of gravity located at the center of rotation. The center of gravity of the entire gun and altitude assembly was aligned to the center of

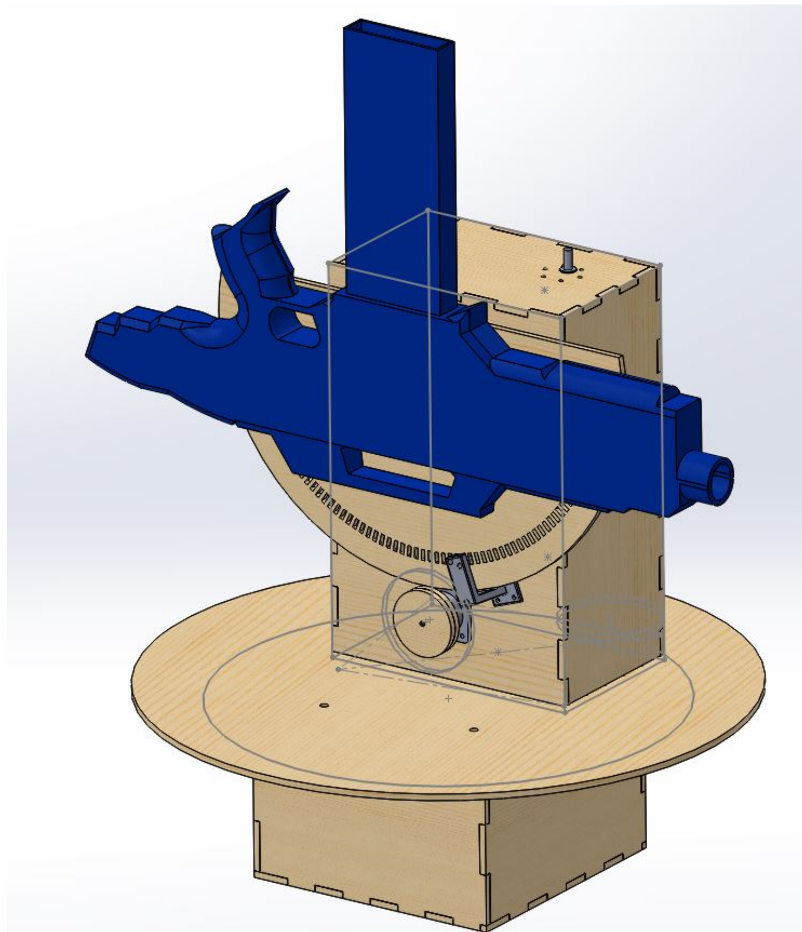


Figure 3: Isometric Solid Model Rendering.

rotation of the azimuth axis. All axis alignments were constructed in Solidworks such that they automatically updated, ensuring alignment through every part of the design. Because the center of gravity was aligned with the center of rotation, all motor torque is applied to overcoming the moment of inertia of the assembly, yielding the quickest angular acceleration possible for any motor-structure combination.

The structure also included built in optical encoders. The bottom encoder can be seen in Figure 4, along with the quadrature read head in grey. Note that the gear teeth are not rendered to expedite Solidworks rendering. Because the assembly did not require precise positioning, a coarse laser cut encoder was added to the altitude and azimuth gears. Laser cutting the encoder allowed for direct feedback from each axis with no possibility of backlash whatsoever, along with cost reductions. The cost reductions proved moot however, due to the processing required to make the encoder signals usable, which will be discussed in Section 3.5.2.

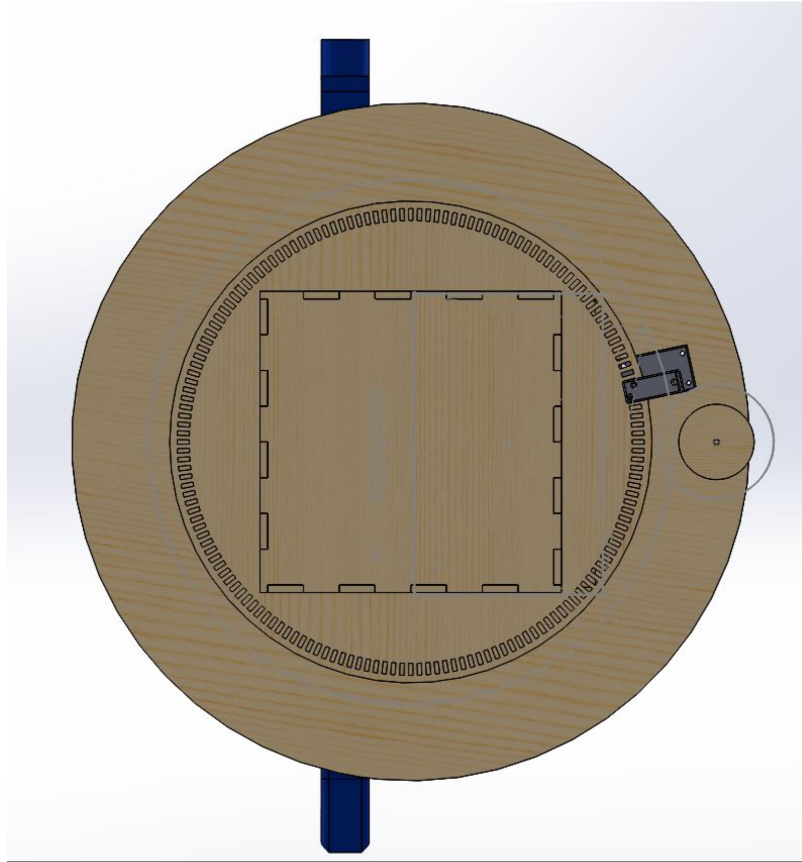


Figure 4: Bottom Solid Model Rendering.

3.3 SIMULATION

One of the main goals of making a solid model of the system was to find the moment of inertia of each of the rotating axes. This would allow the creation of a Simulink model to explore the effect of different motors and gearing combinations on the system.

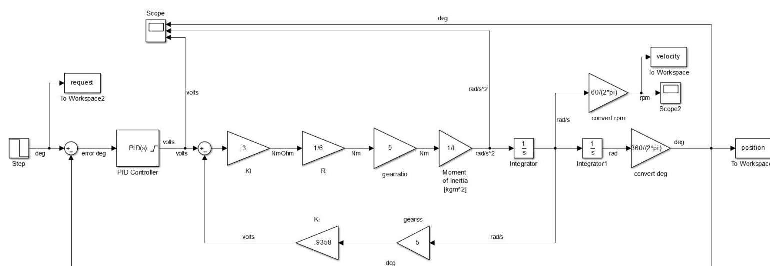


Figure 5: Simulink Closed Loop Model.

Figure 5 shows the Simulink model constructed to simulate the response of closed loop control on a single axis. This allowed the test of

multiple gear ratios and motors to determine what the ideal combination would be. The moment of inertia used in the Simulink model was calculated with Solidworks. The parameters used to simulate the electric motor were calculated from its rated stall torque and no load rpm, as well as its voltage and no load current. Using these, appropriate values to simulate the motor were found. A motor was tested to check that the predicted results were similar to the real motor performance. The results were accurate, within reason considering the tight development schedule of two weeks.

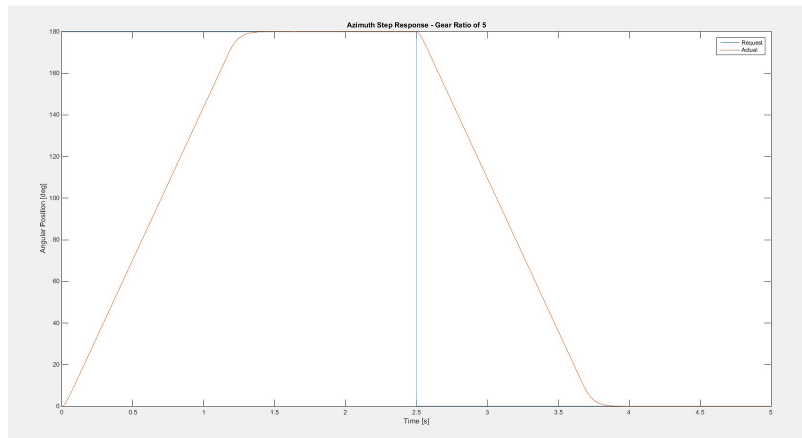


Figure 6: Simulink Closed Loop Step Response.

Figure 6 is a plot of two step inputs to a simple P only controller on the azimuth axis. The step input was from 0 to 180° and then back to 0. We chose this as an appropriate input to the simulation because it modelled the worst case scenario for the apparatus traversing at high speed. The desired response was for the axis to only just achieve its maximum angular rate before being required to decelerate to the set point. Unfortunately, the gear motors purchased already had an extremely high internal gear ratio, which almost removed the necessity of gearing altogether. At this point, the whole mechanism would have ideally been redesigned without gearing, but due to time constraints, we had to move ahead with the system as it stood. The lowest gear ratio we could fit in the current mechanical design was 1:5. The step response in Figure 6 shows the azimuth reaching its top speed extremely quickly, travelling at constant rate to the set point. The system still completes a 180° turn in just over a second, but the performance could be improved in future versions if a better gear ratio was implemented.

3.4 OPTICAL ASSEMBLY

The original independently rotating line scanner was abandoned due to a few unforeseen complications. The main obstacle was the focal length of the Fresnel lenses we selected. The focal length was much

longer than expected, and there was not enough room in the original mounting location for the entire rotating optical assembly. Another major obstacle was the transmission of power to the optical scanner. Power needed to be transmitted through the constantly rotating joint to the internal electronics. While not difficult, it proved to be too much added complexity in our short development schedule. The largest issue however, was the misalignment between the gun and the sensor. Because the gun and the optical scanner did not rotate about the same point, the angle at which the gun and the optical sensor would perceive the target was dependent on the distance to the target. The combination of all these issues made the design unacceptable.

Figure 7: Optical Assembly Attached to Nerf Gun.

The design was revised to mount the line scanner directly on the gun, as shown in Figure 7. The new design featured 14 phototransistors in a vertical line mounted inside a box with a Fresnel lens to focus the light. The box was mounted to the Nerf gun's dovetail accessory rail using a 3d printed part. This allowed the optical assembly to be easily attached and detached from the gun for manufacturing.

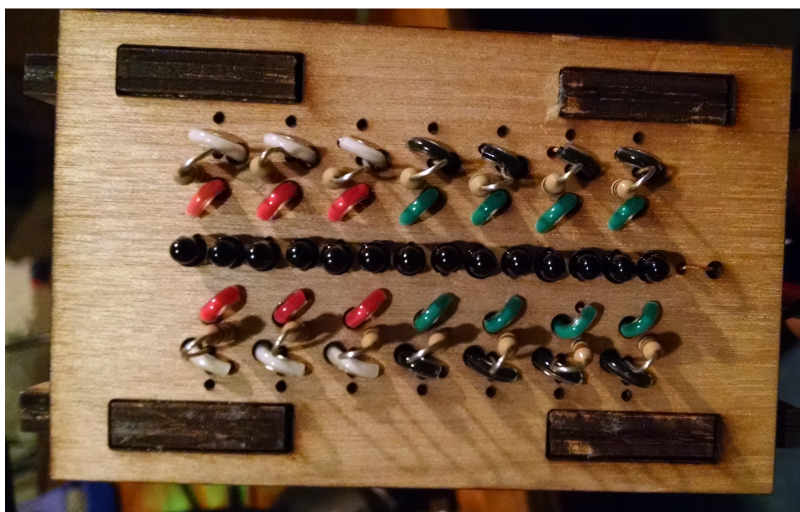


Figure 8: Line Scanner Phototransistor Assembly.

Figure 8 shows the line scanner itself and its 14 constituent phototransistors and their current limiting resistors. All transistors were powered from a single power wire, while each transistor had an independent sense and ground wire so that each transistor output was a twisted pair. The board also had allowances LPF to be installed on each transistor, but this was deemed unnecessary since the readings would largely be averaged, overcoming any small interference.

Figure 9: Demonstration of the Optical Assembly.

Figure 9 demonstrates the optical assembly focusing light on the phototransistors. The assembly was designed with the phototransistor board mounted in slots, allowing the focus to be fine tuned during testing. Once the optimal distance to the lens was determined, the phototransistor board was permanently attached to the optical assembly.

3.5 ELECTRICAL HARDWARE

3.5.1 *Microcontroller*

Most students in our class chose to use the ME405 microcontroller board, which features an Atmel Atmega1281, two motor drivers, and a variety of other supporting hardware. We decided that in order to maintain full functionality of Foamland Security after the course concluded, we would use a Arduino Mega, which utilizes an Atmega2560 also from Atmel. The differences between the two ICs are negligible, with the only serious difference being increased memory size. The board was configured to run FreeRTOS with the accompanying ME405 library.

3.5.2 *Encoder Front End*

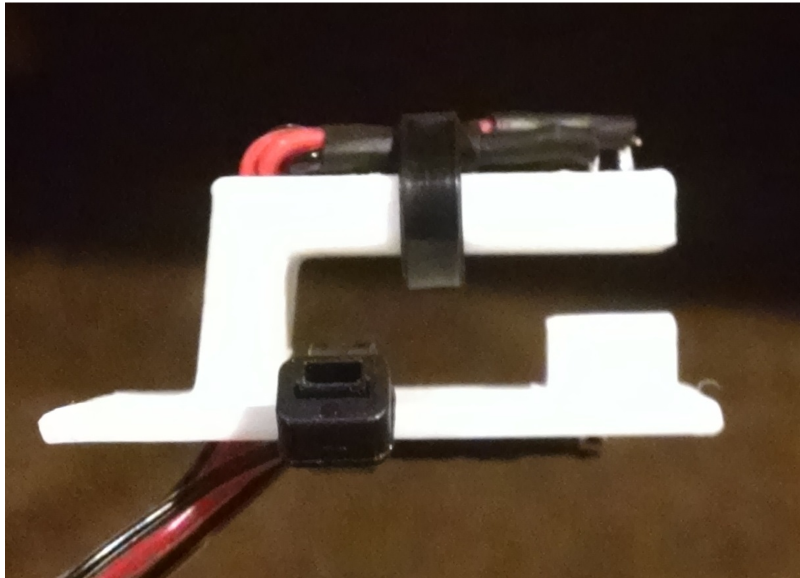


Figure 10: Optical Encoder Quadrature Head.

As mentioned in Section 3.2, we opted out of using commercial encoders, and instead we laser cut encoder grating into the gears and designed housings for an IR emitter and phototransistor to look through the encoder grating as shown in Figure 10. While this was well executed, various problems arose due to:

1. The chosen phototransistors
2. Noise from the H-bridge, motor, and power supply
3. The necessary signal integrity for a signal to be used as an interrupt

The selected phototransistors induced small currents because of their small viewing angle of 24° and limited spectral range from 880 to 1120nm. The small currents resulted in signals more susceptible to noise. This necessitated a LPF to clean the signal up. Because of the extremely small currents involved, it was necessary to use a buffer before any signal filtering could be attempted. After the buffer and LPF were implemented, the signal was relatively smooth. Unfortunately, the slowly varying analog signal from the phototransistor is not compatible with microcontroller interrupts. A comparator had to be added to create a clean square wave with low rise times. Due to the noise and slow transitions of the encoder signal, the comparator would often "bounce" as seen in Figure 11.

This issue was solved by switching the standard comparator with a Schmitt trigger. The Schmitt trigger is similar to a comparator with

Figure 11: Comparator Noise Acceptance.

unidirectional reference voltages. That is, a transition from low to high is compared against one voltage, while a high to low transition is compared against another, making it immune to lollygagging in the digital grey area of 2 to 4 volts. The low to high transition point was set at 4V and the high to low transition around 2V, solved this issue. The final circuit is shown in Figure 12.

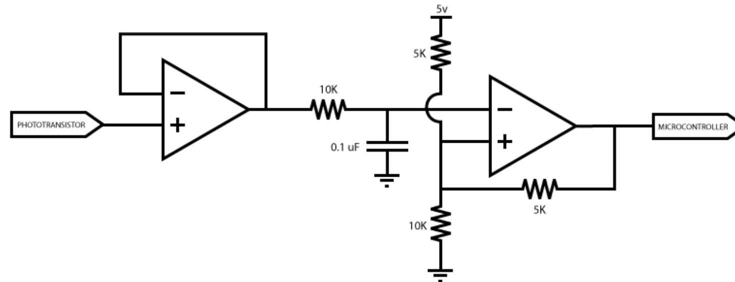


Figure 12: Encoder Signal Conditioning Circuitry.

3.5.3 *Motor Driver*

To drive the brushless DC gear motors, multiple H-bridges were considered. We first fabricated an H-bridge purely out of TO-220 MOSFETS. This was abandoned however because of the excessive size and cost of the circuit, especially since the circuit could handle far higher power than the motors demanded. We scaled back and instead used a SN754410 Quad Half H-bridge from Texas Instruments. The board can be used as two full H-bridges, capable of driving up to 36V and 1A continuously per channel from 5V logic levels. Each half H-bridge

was driven by a PWM pin, allowing 8 bit control of motor voltage in either direction.

3.6 MANUFACTURING

The whole assembly was manufactured either on the laser cutter in the CP Mustang '60 shop or on a Kossel 3d printer at our abode. Figure 13 depicts the process of cutting one of the three panels of parts that made up the device. Some small parts were also 3d printed, including the encoder read heads, small braces, motor mounts, and the optical mounting hardware. Overall, laser cutting the assembly is a much better way to go about manufacturing than 3d printing the entire thing. The geometry is almost entirely 2d, and the application hardly demands anything fancy like helical gears.

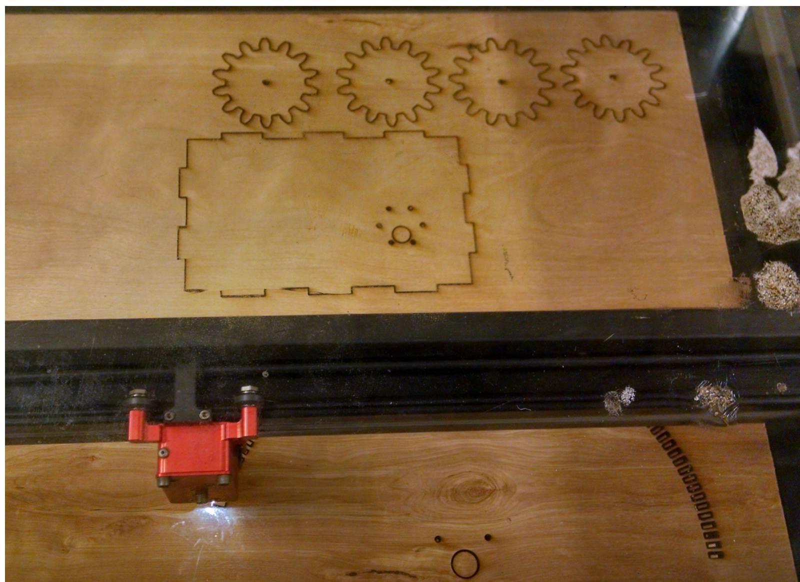


Figure 13: Laser Cutter Manufacturing Foamland Security Parts.

The gun was affixed to the altitude axis using two zip ties. This allowed for easy adjustments of the gun to ensure its alignment with the axis. It also allows for complete disassembly of the altitude axis for later adjustments and additions. Figure 14 shows a picture of Foamland Security in its completed state.

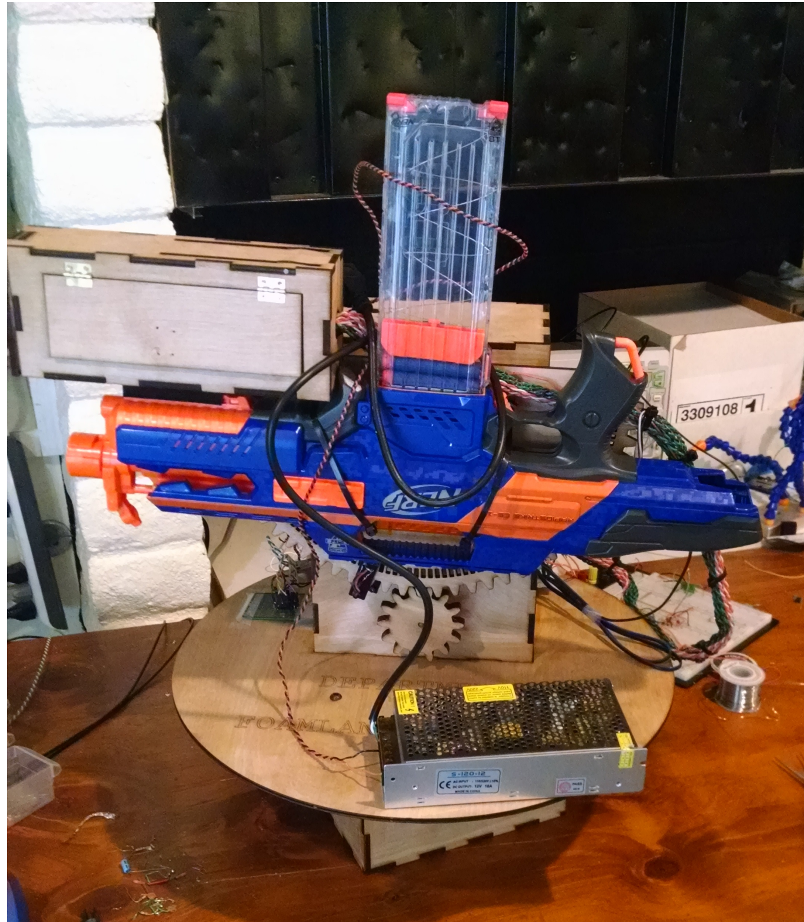


Figure 14: Completed Foamland Security Assembly.

4 SOFTWARE DESIGN

The software design of Foamland Security was chosen with efficiency, readability, and portablility in mind. This is achieved through the use of a RTOS implemented on an Atmega2560, the use of tasks, and by implementing FSMs within tasks.

4.1 IDEATION

The original sketch for sensing IR, as seen in Figure 2, depicts an appartus capable of moving independently of the apparatus, imaging the entire surrounding area. With this in mind we designed the task digram seen in Figure 15. The idea is that the sensor senses the exact position of the target, causing the azimuth and altitude motor controls to work simultaneously to point the Nerf gun at the target, while a task simultaneously checks for the moment that the target is acquired (in task Targeting). However, this sensing appartus was deemed too time intensive to implement so we opted to create the sensing ciruit shown in Figure 7. With an attached sensing circuit the azimuth and altitude postions must be determined sequentially. In other words, the design of this sensing circuitry elimanted the needs for separate altiude, azimuth, and targeting tasks in favor of a single task that achieved the same functionality sequentially. The User Task was also deemed superfluous as Foamland Security operates autonomously, with the excpetion of the switches 'power,' 'enable,' 'auto,' and 'zero.' In the final design these switches are how the user interfaces with Foamland Security, rendering a task for user interfacing unnecessary.

4.2 TASK DIAGRAM

Foamland Security only required two tasks— a control task and a search and fire task, as seen in Figure 16. As previously mentioned, Foamland Security only demands two tasks because much of the work needs to be done sequentially.

It should be noted that the Task Diagram only depicts two variables for intertask communication. While these are the only two variables communicated from task to task, the task Control utilizes other shared variables. For example, the encoder class is responsible for keeping track of the current positions of both the azimuth and altitude motors and storing them in shared variables. Similarly, the encoding class stores error and velocities as shared variables.

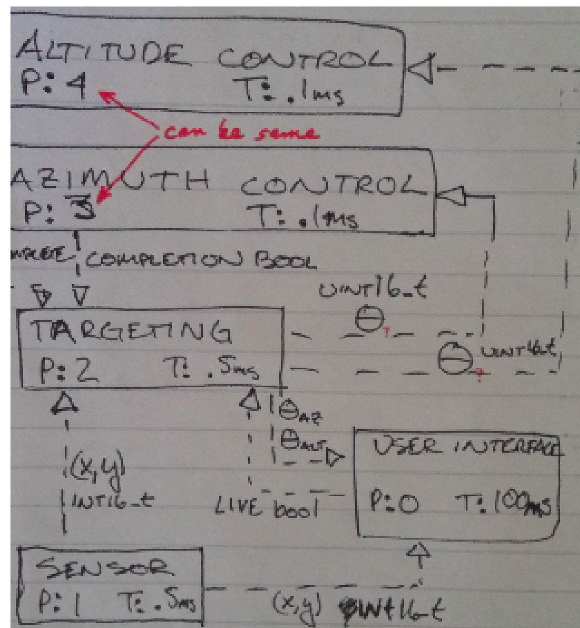


Figure 15: Ideation Task Diagram.

Each task has the same priority but task Control is run every 10ms whereas task Seekill is only run every 25ms. This was deemed desirable because when task Seekill tells task Control to move, nothing can be done until task Control moves the Nerf gun to the desired position.

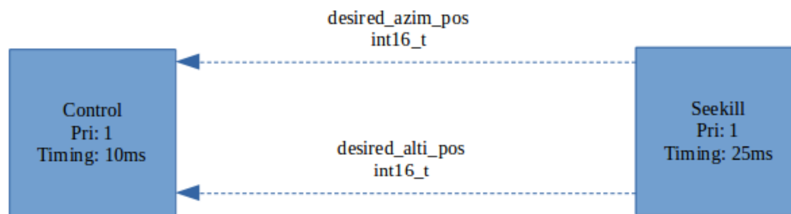


Figure 16: Task Diagram for Foamland Security.

4.2.1 *Task Seekill Finite State Machine*

The FSM for task Seekill, seen in Figure 17, details how Foamland Security searches for, and fires at, the target. The underlying basis is that the IR sensing circuitry is a vertical line of 14 sensors, the first state should seek to locate the target horizontally by averaging the analog values returned by all 14 sensors. Once the analog volages returned by the sensing circuity are averaged the state checks to see if the current average value is greater than the past averaged values and if it is, it stores it as the new peak value. Regardless of whether or not the current averaged value is greater than the peak average value, the next state that the FSM moves to is Move Azim, which moves

the apparatus horizontally and then returns to Sense Azim to process the new sensor readings. The FSM does this continuously until it reads an analog voltage that is significantly less than the peak analog voltage. This condition means that Foamland Security has passed its target horizontally. This condition transitions the FSM from state Sense Azim to state Sense Alti. The following snippet of code shows the two cases used for sensing and moving Foamland Security horizontally:

```
// state sense_azim reads sensors and compares to last readings
case (sense_azim):
{
        uint16_t azim_average = average_all_ADC ();      // calls
             a function to avg all channels

        // compares last sensor reading to current and stores the
             highest
        if (azim_average > peak_avg_a2d)
        {
                peak_avg_a2d = azim_average;
                transition_to (move_azim);
        }

        // checks if there's been a large drop off of IR because
            this means we've passed the target!
        else if (azim_average < (peak_avg_a2d - SIG_IR_DROP))
        {
                while (azim_average != peak_avg_a2d)      // get
                    us back to our target
                {
                        desired_azim_pos->put(current_azim_pos->
                            get() - 15);
                        azim_average = average_all_ADC();
                              // update ADC readings
                        delay_ms(10);
                                              // allows us to
                            switch tasks and actually move
                }
                transition_to (sense_alti);
        }

        // if the sensors haven't told us anything significant,
            we keep on keepin' on
        else
        {
                transition_to (move_azim);
        }
}
        break;  // end state sense_azim
```

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

// state move_azim moves the azimuth motor and then sends us back
     to sense_azim to keep checking IR
case (move_azim):
        desired_azim_pos->put(current_azim_pos->get() + 15);
        while ((current_azim_pos->get() < (desired_azim_pos->get
            () - 4)) || (current_azim_pos->get() > (
            desired_azim_pos->get() + 4)))
        {
                delay_ms(15);   // twiddle your thumbs
        }
        transition_to (sense_azim);
        break;  // end state move_azim
```

Once Foamland Security believes it has passed its target it backtracks until it detects a voltage similar to the peak. Then it transitions to Sense Alti where it scans all fourteen sensors to find the sensor sensing the highest concentration of IR. Once located, the task Control is told to move Foamland Security to the target, after which the FSM moves onto the firing sequence. The following sequence of code shows how Foamland Security searches and moves to the target in the altitude position:

```
// state sense_alti is entered when we've locked in horizontally
    and now we need to find the target vertically
case (sense_alti):
        desired_azim_pos->put(current_azim_pos->get()); // freeze
             horizontal
        alti_seek();
                                    // a function that lock us
            onto the target vertically
        desired_alti_pos->put(current_alti_pos->get()); // freeze
             vertically
        transition_to (initiate_firing);
                // begin shooting sequence
        break;  // end state sense_alti
```

The firing process spans three states to allow the flywheels in the Nerf gun to get up to speed before the trigger can be "pulled." The trigger must be "pulled" for a time long enough to eject projectiles. This sequence represents a challenge because as the RTOS idles in a state within the firing sequence, the RTOS should not leave the current task. In order to accomplish this a while loop was implemented that used the tick counts as a condition for delay, rather than using `delay_ms()`, which would have caused the RTOS to move to another task.

It should be noted that both the states for scanning horizontally and vertically use methods such as `average_all_ADC ()` and `alti_seek()`, which can be seen in Appendix A.

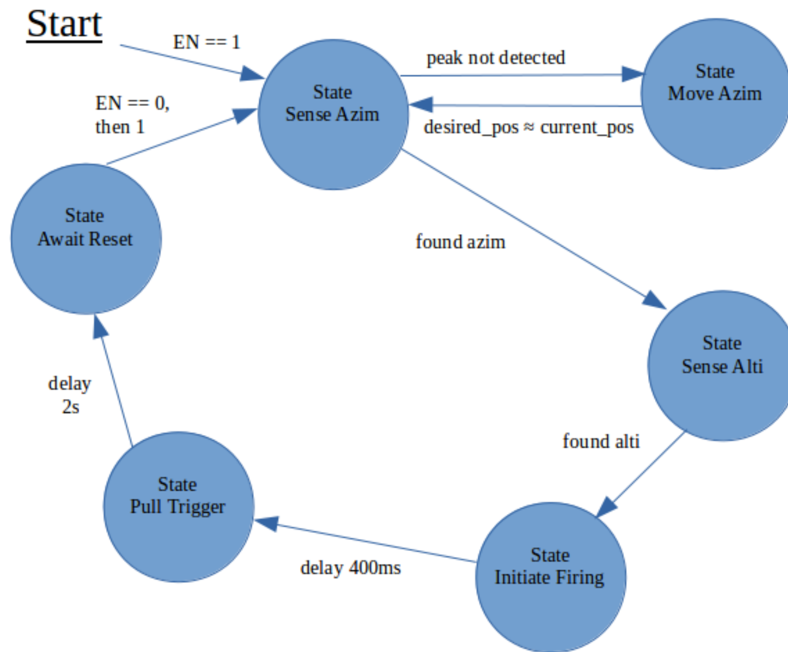


Figure 17: Finite State Machine of Task Seekill (Pronounced Task Seek-Kill).

4.2.2 *Task Control FSM*

The task Control, seen in Figure 18, consists of three states: Check Difference, Azim Power, and Alti Power. The state Check Difference checks if there is a difference between the desired and current azimuth position and if there is, then it transitions to state Azim Power where it sends the difference to an object of the PID class to determine the desired power to be sent to a method of an object of the motor class that sets the appropriate motor's power (by setting the duty cycle of the PWM). After the azimuth motor speed has been updated, the FSM transitions back to check difference. Once the azimuth position has been aligned with the target, state Check Difference will return an azimuth difference of 0 and then the altitude difference will be checked and the altitude power will be similarly found and set until Foamland Security finds the target. The source code for task Control can be found in Appendix A and the following snippet of code shows how the PID class works:

```
int16_t pid::action (int16_t diff)
{
3       int8_t isat = sat/Ki;
        int16_t proportional = (diff*Kp);// calc proportional
            factor

        if (((diff > imin_diff) || (diff < -imin_diff)) && (act >
             -sat) && (act < sat))
        {
```

```
8               iaccumulate += diff;        // only occurs if
                    diff is substantial and if act is not
                    saturated (mitigates windup)
                if (iaccumulate > isat)         // further
                    mitigates windup
                        iaccumulate = isat;
                else if (iaccumulate < -isat)
                        iaccumulate = -isat;
13      }

        int16_t integral = iaccumulate*Ki;
        int16_t derivative = (diff-prev_diff)*Kd;

18      act = proportional + integral + derivative; // calculate
            desired output power

        if (act > sat)
                        // ensures action isn't larger than the
            max power we can give the motor
                act = sat;
        else if (act < -sat)
23              act = -sat;
        prev_diff = diff;

        return act;
}
```

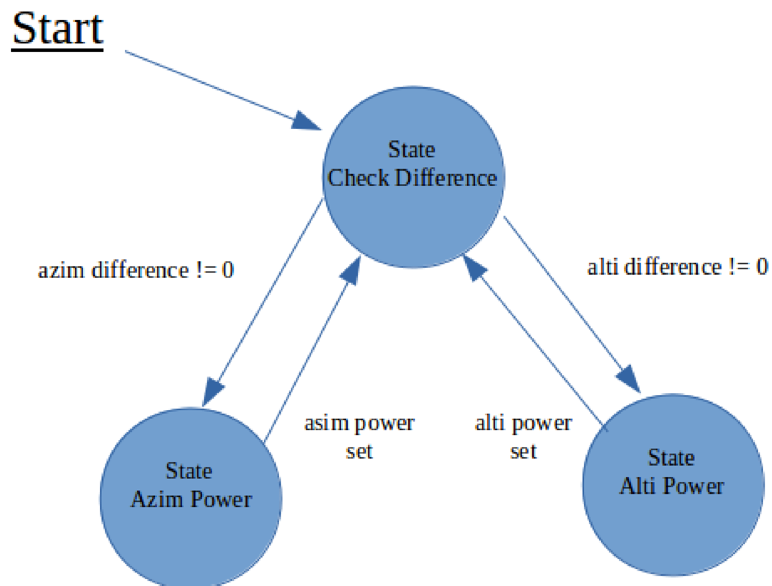


Figure 18: Finite State Machine of Task Control.

Part II

RESULTS

5 RESULTS

5.1 SPECIFICATION EVALUATION

The following table is a reiteration of the table from Chapter 2 with an added Pass/Fail column.

Specification	Justification	Pass/Fail
Must have PID control over two axes	Two axis control is necessary to successfully target the heat lamp	Pass
Must sense IR	IR sensing is necessary to be able find the heat lamp	Pass
Must use a lens	Specified by Dr. Ridgely to increase chances of success	Pass
Must use gears or belts	Specified by Dr. Ridgely	Pass
Must use a RTOS	RTOS is an appropriate tool for this task and a core concept of ME405	Pass
Must use tasks	Tasks are a core concept of ME405	Pass
Must operate autonomously	A primary objective of learn by dueling is to have a closed loop system	Pass
Must autonomously locate, shoot at, and hit target	Overall goal of Learn By Dueling	Fail

The final product theoretically fully functions but in the process of moving the encoder filtering circuit from a breadboard to protoboard, one of the filtering circuits was damaged. This means that we had PID control over either azimuth or altitude but we could not implement control over both simulateously to show our ability to "autonomously locate, shoot at, and hit the target." Despite this, Foamland Security can display all of the desired functionalities separately.

5.2 V2

During the development and immediately following the completion of Foamland Security, it became clear that many design decisions were not ideal in V1. As with most designs, Foamland Security must

go through another iteration to improve its utility. Since we bought all the components ouselves, iterating on the current design seems almost inevitable to get a fully functioning prototype and improve the shortcomings of V1. Here are the proposed changes.

5.2.1 *Mechanical Design*

The most obvious deficiency of the current mechanical design is in the selected gear ratios. It reaches its maximum angular speed almost instantly, wasting much of the torque of the motors. In V2, this would be corrected. The current design would have to be completely overhauled to allow for a lower gear ratio than 1:5 however, as gears with more similar diameters simply will not work with the design. To solve this issue on the altitude axis, the gearing will be moved to the internals of the altitude tower, and utilize a timing belt and pulleys. The altitude encoder will also be omitted and a potentiometer will be used in its place. In this application a potentiometer is a better solution since it provides absolute position feedback, eliminating the endstop switch required by the encoder. With the 10 bit ADC on the Atmega 2560, this provides a resolution of 0.3°. A potentiometer is not a limiting option as the axis is not required to rotate more than approximately 100°. The azimuth will continue with a custom laser cut optical encoder, but the grating will be moved internally, so that the read head will be completely hidden from view inside the lower base. New phototransistors will be selected with higher currents, with an amplifier placed immediatley adjacent to them to eliminate noise issues. The azimuth drive will also be switched to timing belts and pulleys, but mounted inside the encoder ring, so that the motor will be mounted within the altitute box, hiding it from view as well. The V2 assembly will be entirely more sleek than the V1 assembly, hiding all mechanical aspects inside enclosures, and eliminating all exposed wiring by running the Nerf gun control wires through the hollow altitude shaft.

5.2.2 *Electrical Design*

The principal issue of V1 was the custom lasercut encoders. The low currents of the phototransistors required somewhat complex filtering and conditioning to produce the required square wave. This was caused both by the low current of the phototransistors, as well as excessive noise from the brushed DC motors. To lessen both causes of this problem, we must address both the motors and the phototransistors. To improve the signals from the phototransistors, they will be replaced by more sensitive ones and supplied by a higher voltage. An amplifier will be placed as close as physically possible to the phototransistors to lessen the influence of any noise on the signal. To

decrease the noise produced, the motors will be switched to brushless DC motors, with their drivers PWM driven through Darlington transistors to lessen the current load on the Arduino linear regulator. V2 will also feature a power slip ring connection to transmit power through the azimuth joint. This allows the entire assembly to rotate unhindered in any direction.

5.2.3 *Sensor Design*

To ensure the applicablility of Foamland Security to our college apartment, the optical assembly will be abandoned in favor of machine vision. As of yet, the specific system has not been selected, but likely candidates are a Raspberry Pi or BeagleBone with a camera and OpenCV, a C++ library of computer vision functions, or a standalone module like OpenMV or Pixy. This will allow Foamland Security to target and shoot individuals that are not bright sources of IR light. In addition to machine vision control, a 2.4 GHz remote will also be added, allowing an individual to manually aim and fire the turret, as well as engage the automatic targeting mode.

Part III

APPENDIX

CODE

A.1 TASK SEEKILL

```
//
    ************************************************************************
/** @file task_seekill.cpp
 *    This file contains a task class that senses the highest IR
    light around, using 14
 *        ADCs, and lets the task class control know where it is
*/
//
    ************************************************************************
#include <avr/io.h>                              // Port I/O for SFR's
#include <avr/wdt.h>                             // Watchdog timer
    header
#include "task_user.h"                           // Header for this
    file
#include "shares.h"
#include "motordrive.h"
#include "task_control.h"                        // header for
    task_control
#include "task_seekill.h"                        // header for
    task_seekill

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

#define SIG_IR_DROP 400
    // value that represents a significant drop in IR for sensing
#define NUM_ADC_CH 14
    // number of ADCs our sensing circuitry uses

// enumerated type that allows descriptive names for states in
    FSM
enum state_names_t {sense_azim = 0, move_azim = 1, sense_alti =
    2, initiate_firing = 3, pull_trigger = 4, await_reset = 5};
//
    ------------------------------------------------------------------------


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

```
/** This constructor creates a task which initializes an enable
    pin and creates a tast
 *      for sensing and shooting at our target.
 *  The main job of this constructor is to call the constructor
     of parent class (\c frt_task );
 *  the parent's constructor does the work.
 *  @param a_name A character string which will be the name of
     this task
 *  @param a_priority The priority at which this task will
     initially run (default: 0)
 *  @param a_stack_size The size of this task's stack in bytes
 *                    (default: configMINIMAL_STACK_SIZE)
 *  @param p_ser_dev Pointer to a serial device (port, radio, SD
     card, etc.) which can
 *                    be used by this task to communicate (default
     : NULL)
 */

task_seekill::task_seekill (const char* a_name,
                                                        unsigned
                                                            portBASE_TYPE

                                                            a_priority
                                                            ,
                                                        size_t
                                                            a_stack_size
                                                            ,
                                                        emstream*

                                                            p_ser_dev
                                                            )
        : TaskBase (a_name, a_priority, a_stack_size, p_ser_dev)
{
        // pin H0 will be used to enable foamland security
        DDRH &= ~(1 << 0);
}

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

void task_seekill::run (void)
             // method called RTOS scheduler
{
        TickType_t previousTicks = xTaskGetTickCount (); // holds
             times to use for precise task scheduling
        uint16_t peak_avg_a2d = 400     ;
                             // creates a variable to store
            highest average

        motordrive* azim_motor = new motordrive(1, DDB5, DDB6, &
            OCR1B, &OCR1A, p_serial);       // pins 11 and 12
            PWMing
```

```
53          motordrive* alti_motor = new motordrive(0, DDE3, DDE4, &
                OCR3B, &OCR3A, p_serial);       // pins 5 and 2
                PWMing

            for (;;)
            {
                    bool EN = (PORTH & (1< PINH0));
                                // read the enable pin
58                  while (EN != 1)
                    {
                            // twiddle thumbs
                    }

63                  switch (state)      // run FSM, 'state' is kept
                        by parent class
                    {
                            //
                                ---------------------------------------------------------------

                            // state sense_azim reads sensors and
                                compares to last readings
                            case (sense_azim):
68                          {
                                    uint16_t azim_average =
                                        average_all_ADC ();      //
                                        calls a function to avg all
                                        channels

                                    // compares last sensor reading
                                        to current and stores the
                                        highest
                                    if (azim_average > peak_avg_a2d)
73                                  {
                                            peak_avg_a2d =
                                                azim_average;
                                            transition_to (move_azim)
                                                ;
                                    }

78                                  // checks if there's been a large
                                         drop off of IR because this
                                        means we've passed the target
                                        !
                                    else if (azim_average < (
                                        peak_avg_a2d - SIG_IR_DROP))
                                    {
                                            while (azim_average !=
                                                peak_avg_a2d)      //
                                                 get us back to our
                                                target
                                            {
```

```
83                              desired_azim_pos
                                    ->put(
                                    current_azim_pos
                                    ->get() - 15)
                                    ;
                                azim_average =
                                    average_all_ADC
                                    ();
                                    // update ADC
                                     readings
                                delay_ms(10);




                                                //
                                    allows us to
                                    switch tasks
                                    and actually
                                    move
                        }
                        transition_to (sense_alti
                            );
88              }

                // if the sensors haven't told us
                     anything significant, we
                    keep on keepin' on
                else
                {
93                      transition_to (move_azim)
                            ;
                }
        }
                break;  // end state sense_azim

98      //
            -----------------------------------------------------------

        // state move_azim moves the azimuth
            motor and then sends us back to
            sense_azim to keep checking IR
        case (move_azim):
                desired_azim_pos->put(
                    current_azim_pos->get() + 15)
                    ;
                while ((current_azim_pos->get() <
                     (desired_azim_pos->get() -
                    4)) || (current_azim_pos->get
                    () > (desired_azim_pos->get()
                     + 4)))
103             {
```

```
                        delay_ms(15);   //
                            twiddle your thumbs
                }
                transition_to (sense_azim);
                break;  // end state move_azim

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

        // state sense_alti is entered when we've
             locked in horizontally and now we
            need to find the target vertically
        case (sense_alti):
                desired_azim_pos->put(
                    current_azim_pos->get()); //
                    freeze horizontal
                alti_seek();

                                            // a
                    function that lock us onto
                    the target vertically
                desired_alti_pos->put(
                    current_alti_pos->get()); //
                    freeze vertically
                transition_to (initiate_firing);

                    // begin shooting sequence
                break;  // end state sense_alti

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

        // states initiate_firing through
            await_reset are a firing sequence,
            ending in an idle state
        case (initiate_firing):         // starts
             nerf motor at full speed
                firing (255, 0);
                while (xTaskGetTickCount() !=
                    previousTicks + 400)
                {
                                // build up motor
                                     speed
                }
                transition_to (pull_trigger);
                break;  // end state
                    initiate_firing

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

```
                    case (pull_trigger):    // prevents motor
                        from firing too fast and pulls
                        trigger
                        firing ((255/2), (255/2));
                        while (xTaskGetTickCount() !=
                            previousTicks + 2000)
133                     {
                                        // fire
                                            projectiles
                        }
                        firing (0, 0);  // shuts off nerf
                            motor and trigger
                        transition_to (await_reset);
138                     break; // end state pull_trigger

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

                    case (await_reset):        // the gun
                        idles in await_reset until enable is
                        pressed and then we start again
                        while (EN == 1)
143                     {
                                EN = (PORTH & (1< PINH0))
                                    ;      // idle
                                    victoriously until
                                    services are re-
                                    requested
                        }
                        while (EN != 1)
                        {
148                             EN = (PORTH & (1< PINH0))
                                    ;      // idle
                                    victoriously until
                                    services are re-
                                    requested
                        }
                        transition_to(sense_azim);
                        break; // end state await_reset
                }
153     }
        delay_from_for_ms (previousTicks, (25));
}

//
    ----------------------------------------------------------------------------------
158
uint16_t task_seekill:: average_all_ADC (void)
    // method to average all 16 ADCs
{
```

```
        adc* p_my_adc = new adc (p_serial);
                            // create an object of adc to use in
            this method

163     uint16_t average = 0;
                                    // stores average ADC value
        int8_t adc_channel = 0;
                                    // determines which ADC
            channel is read
        bool round_two = 0;

        // this for loop runs us through all 14 ADCs that our
            sensor utilizes
168     for (uint8_t scanner = NUM_ADC_CH; scanner > 0; scanner
            --)
        {
                if (adc_channel > 7)
                                            // allows use of 2nd
                    chunk of 8 ADCs
                {
                        round_two = 1;
173                     adc_channel = 0;
                }
                average += p_my_adc->read_once(adc_channel,
                    round_two); // accumulates readings
                adc_channel++;
        }
178     return (average/NUM_ADC_CH);
                    // returns accumulated value divided by
            number of readings
}

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

183 void task_seekill:: alti_seek(void)
{
        adc* p_my_adc = new adc (p_serial);

        int8_t adc_channel = 0;
188     bool round_two = 0;
        bool max_round = 0;
        uint16_t max_analog = 0;
        uint8_t max_adc_channel = 0;

193     for (uint8_t num_adc_channels = 14; num_adc_channels > 0;
             num_adc_channels--)
        {
                if (adc_channel > 7)
                {
                        round_two = 1;
```

```
                        adc_channel = 0;
                }

                // read the voltage on each phototranny, 1 by 1
                uint16_t analog = p_my_adc->read_once(adc_channel
                    , round_two);

                // allows us to store exactly which phototranny
                    saw the most IR
                if (analog > max_analog)
                {
                        max_analog = analog;
                        max_adc_channel = adc_channel;
                        max_round = round_two;
                }
                adc_channel++;
        }

        // a loop that runs until the barrel is pointing at the
            target
        while (p_my_adc->read_once(max_adc_channel, max_round) !=
             max_analog)
        {
                desired_alti_pos++;
                while (desired_alti_pos != current_alti_pos)
                {
                        delay_ms(10);   //twiddle your thumbs
                }
        }
//      trigger->put(1);                // "pulls" trigger
        return;
}

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


void task_seekill:: firing(uint8_t motor_speed, uint8_t
    fire_speed)
{
        /* configuring pins H3 and H4 to serve as PWM pins for
            the supply of the nerf
           gun motor and for the gate of the FET controlling the
               trigger*/

        DDRH |= (1 << 3);
                      // pins H3 and H4 configured as outputs
        DDRH |= (1 << 4);

        TCCR4A |= (1 << WGM40);                   // configure 8
            bit fast PWM
        TCCR4B |= (1 << WGM42);
```

```
        TCCR4B |= (1 << CS42) | (1 << CS40);      // set
            prescaler for timer/counter at /64

        TCCR4A |= (1 << COM4B1);
        TCCR4A &= ~(1 << COM4B0);
243     TCCR4A |= (1 << COM4A1);
        TCCR4A &= ~(1 << COM4A0);

        // pins H3 and H4 PWM at a duty cycle that correlates to
            the passed in variables
        OCR4A = motor_speed;
248     OCR4B = fire_speed;

        return;
}
```

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

/** @file task_seekill.h
 *    This file is the header for the task class sense.cpp that
     read the ADC values
4 *        from the sensing circuity and determines the location
     that is getting the largest
 *        amount of IR light
*/
//
    *************************************************************************


9 #ifndef _task_sense_h_
    // prevents multiple inclusions
#define _task_control_h_

#include <stdlib.h>                          // prototype
    declarations for I/O functions
#include <avr/io.h>                          // header for special
     function registers
14
#include "FreeRTOS.h"                        // primary header for
     FreeRTOS
#include "task.h"                            // header for
    FreeRTOS task functions
#include "queue.h"                           // FreeRTOS inter-
    task communication queues

19 #include "taskbase.h"                       // ME405/507 base
    task class
#include "time_stamp.h"                      // class to implement
     a microsecond timer
#include "taskqueue.h"                       // header of wrapper
    for FreeRTOS queues
```

```
#include "taskshare.h"                    // header for thread-
    safe shared data
#include "shares.h"                       // global ('extern')
    queue declarations
24
#include "rs232int.h"                     // ME405/507 library
    for serial comm.
#include "adc.h"                          // header for A/D
    converter driver class
#include "encoderdriver.h"
        // encoder initialization and interrupts
#include "pid.h"
        // pid control and creation of encoder objects
29 #include "motordrive.h"
    // motor initialization and power setting
#include "adc.h"
            // motor initialization and power setting
//
    ------------------------------------------------------------------------------------

/** @brief   This class senses IR
 *  @details This class uses readings from 14 ADCs to locate the
     largest source of IR lights
34 *                    in both the X and Y axis
 */

class task_seekill : public TaskBase
{
39 private:
        // nothing as of right now

protected:
        uint16_t average_all_ADC (void);        // method to
            average all 16 ADCs
44      void firing(uint8_t motor_speed, uint8_t fire_speed);
        void alti_seek(void);                           // find
            max analog altitude voltage and sets a bool

public:
        // This constructor creates a generic task of which many
            copies can be made
49      task_seekill (const char*, unsigned portBASE_TYPE, size_t
            , emstream*);

        // This method is called by the RTOS once to run the task
             loop for ever and ever.
        void run (void);
};
54
#endif // _task_sense_h_
```

A.2 TASK CONTROL

```
//
    ************************************************************************
/** @file task_control.cpp
 *    This file contains outlines a task that creates objects of
     motordriver, encoder,
 *    and PID classes. Once all of these objects have been
     created, it allows PID control
 *    of each motor that classes have been created for.
*/
//
    ************************************************************************

#include <avr/io.h>                          // port I/O for SFR's
#include <avr/wdt.h>                         // watchdog timer
    header
#include "task_user.h"                       // header for this
    file
#include "shares.h"
#include "motordrive.h"
#include "task_control.h"                    // header for
    task_control

//
    ------------------------------------------------------------------------
// enumerated type that allows descriptive names for states in
    FSM
enum state_names_t {check_diff= 0, azim_power = 1, alti_power =
    2};

//
    ------------------------------------------------------------------------
/** This constructor creates a task which handles PID control
 *  The main job of this constructor is to call the constructor
     of parent class (\c frt_task );
 *  the parent's constructor does the work.
 *  @param a_name A character string which will be the name of
     this task
 *  @param a_priority The priority at which this task will
     initially run (default: 0)
 *  @param a_stack_size The size of this task's stack in bytes
 *                      (default: configMINIMAL_STACK_SIZE)
 *  @param p_ser_dev Pointer to a serial device (port, radio, SD
     card, etc.) which can
 *                   be used by this task to communicate (default
     : NULL)
 */

```

```
task_control::task_control (const char* a_name,
                                                unsigned
                                                    portBASE_TYPE
                                                     a_priority,
                                                size_t
                                                    a_stack_size,
                                                emstream*
                                                    p_ser_dev)
35      : TaskBase (a_name, a_priority, a_stack_size, p_ser_dev)
{
        ptr_serial_ctrl = p_ser_dev;
                     // allows communication
}

40
//
    ---------------------------------------------------------------------

void task_control::run (void)
             // called by RTOS scheduler
{
45      TickType_t previousTicks = xTaskGetTickCount (); // holds
             times to use for precise task scheduling

        // create objects for encoder, motordrivers, and pid
        encoder* encoders = new encoder(p_serial);
        motordrive* alti_motor = new motordrive(0, DDE3, DDE4, &
            OCR3B, &OCR3A, p_serial);       // pins 5 and 2
            PWMing
50      motordrive* azim_motor = new motordrive(1, DDB5, DDB6, &
            OCR1B, &OCR1A, p_serial);       // pins 11 and 12
            PWMing
        pid* azim_control = new pid(p_serial, 20, 1, 0, 255);

                    // pass in Ki, Kp, Kd, and sat
        pid* alti_control = new pid(p_serial, 3, 0, 0, 255);

        // initialize
55      int16_t diff = 0;
        int16_t des_power = 0;

        for (;;)
        {
60              switch (state)      // run FSM, 'state' is kept
                    by parent class
                {
                        //
                            ----------------------------------------------
```

```
// state check_diff determines whether or
     not power needs to be set in either
    azim or alti
case (check_diff):
{
        diff = desired_azim_pos->get() -
            current_azim_pos->get();
                  // diff represents how
             far motor needs to turn
        if (diff > 0)
        {
                transition_to(azim_power)
                    ;
        }

        diff = desired_alti_pos->get() -
            current_alti_pos->get();

        else if (diff > 0)
        {
                transition_to(alti_power)
                    ;
        }
}
        break;  // end state check_diff

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

// state azim_power determines and sets
    power for azimuth motor
case (azim_power):
{
        des_power = azim_control->action(
            diff);
                                    //
            PID calculated power to turn
            motor efficiently
        azim_motor->set_power(des_power);


                   // sets the motor
        delay_ms(25);


                                    //
            breaks to do something else
            for a bit
        transition_to (check_diff);
}
        break;  // end state azim_power
```

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

                        // state alti_power determines and sets
                            power for altitude motor
                        case (alti_power):
95                              des_power = alti_control->action(
                                    diff);
                                alti_motor->set_power(des_power);
                                transition_to (check_diff);
                                break;  // end state alti_power
                }
100             // PID debugging//
//              *ptr_serial_ctrl << "Current azimuth pos: " <<
    current_azim_pos->get() << endl;
//              *ptr_serial_ctrl << "Desired azimuth pos: " <<
    desired_azim_pos->get() << endl;
//              *ptr_serial_ctrl << "Calculated difference: " <<
    diff << endl << "**********************" << endl << endl;

105             // pure encoder debugging
//              *ptr_serial_ctrl << "**********************" <<
    endl << "Power sent to motors: " << des_power << endl;
//              *ptr_serial_ctrl << "C " << current_azim_pos->get
    ();
//              *ptr_serial_ctrl << "D " << desired_azim_pos->get
    ();
//              *ptr_serial_ctrl << "e " << azim_error->get();
110
                delay_from_for_ms (previousTicks, (10));
        }
}
```

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

2 /** @file task_control.h
   *    This file contains the header for a task class that creates
        objects of motor,
   *        encoder, and PID classes and accomplishes PID control
       with these objects
  */
  //
    ************************************************************************

7
  // prevents mutiple inclusions
  #ifndef _task_control_H_
  #define _task_control_H_

12 #include <stdlib.h>                          // prototype
    declarations for I/O functions
```

```
#include <avr/io.h>                          // header for special
     function registers

#include "FreeRTOS.h"                        // primary header for
     FreeRTOS
#include "task.h"                            // header for
    FreeRTOS task functions
#include "queue.h"                           // FreeRTOS inter-
    task communication queues

#include "taskbase.h"                        // ME405/507 base
    task class
#include "time_stamp.h"                      // class to implement
     a microsecond timer
#include "taskqueue.h"                       // header of wrapper
    for FreeRTOS queues
#include "taskshare.h"                       // header for thread-
    safe shared data
#include "shares.h"                          // global ('extern')
    queue declarations

#include "rs232int.h"                        // ME405/507 library
    for serial comm.
#include "adc.h"                             // header for A/D
    converter driver class
#include "encoderdriver.h"
        // encoder initialization and interrupts
#include "pid.h"
        // pid control and creation of encoder objects
#include "motordrive.h"
    // motor initialization and power setting


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

/** @brief   This task controls the motors
 *  @details This task uses interrupts generated from an optical
     encoder to determine the
 *                    location of a motor and it uses PID
     control to move the motor to desired
 *                    positions in a timely manner.
 */

class task_control : public TaskBase
{
private:
        emstream* ptr_serial_ctrl;

protected:
        // No protected variables or methods for this class
```

```
public:
        // This constructor creates a generic task of which many
            copies can be made
        task_control (const char*, unsigned portBASE_TYPE, size_t
            , emstream*);

        // This method is called by the RTOS once to run the task
             loop for ever and ever.
        void run (void);
};

#endif // _task_control_H_
```

A.3 CLASS PID

```
//
    ********************************************************************************
/** @file pid.cpp
 *    This file outlines a class for PID control.
*/
//
    ********************************************************************************

#include <stdlib.h>                         // standard library
    header files
#include <avr/io.h>
#include "rs232int.h"                       // serial port class
#include "pid.h"                                // pid class

#define imin_diff 20
    // minimum difference to warrant integral accumulation
//
    --------------------------------------------------------------------------------

/** \brief This constructor constructs on object of the pid class
 *  \details This constructor saves the passed in proportional,
     integral, derivative,
 *                       and saturation limit constants to
     variables in the pid class, and initializes,
 *                       the integral variable to 0.
 *  @param Kp_in Proportional control constant
 *  @param Ki_in Intgeral control constant
 *  @param Kd_in Derivative control constant
 *  @param sat_in Power saturation limit constant
 */


pid::pid (emstream* p_serial_port, uint16_t Kp_in, uint16_t Ki_in
    , uint16_t Kd_in, uint16_t sat_in)
```

```
{
        ptr_serial_pid = p_serial_port;
        Kp = Kp_in;             // stores derivative gain
            constant
        Ki = Ki_in;             // stores integral gain constant
30      Kd = Kd_in;             // stores derivative gain
            constant
        prev_diff = 0;   // used for calculating differince in
            error
        sat = sat_in;    // stores saturation limit
        iaccumulate = 0; // initalize integral once
        act = 0;
35  }


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

    /** @brief   This method determines desired amount of power to
        send to motor
40   *  @details This method takes in a variable that represents how
         far away the desired
     *                    location is and then decides how much
         power (@act) to send to the motor of
     *                    this specific object to get to that
         desired location
     *  @param   diff This parameter represents how far away the
         desired position is
     */
45
    int16_t pid::action (int16_t diff)
    {
        int8_t isat = sat/Ki;
        int16_t proportional = (diff*Kp);              // calc
            proportional factor
50
        if (((diff > imin_diff) || (diff < -imin_diff)) && (act >
             -sat) && (act < sat))
        {
            iaccumulate += diff;       // only occurs if
                diff is substantial and if act is not
                saturated (mitigates windup)
            if (iaccumulate > isat) // further mitigates
                windup
55              iaccumulate = isat;
            else if (iaccumulate < -isat)
                iaccumulate = -isat;
        }

60      int16_t integral = iaccumulate*Ki;
        int16_t derivative = (diff-prev_diff)*Kd;
```

```
        act = proportional + integral + derivative; // calculate
            desired output power

        if (act > sat)
                                        // ensures action isn't
            larger than the max power we can give the motor
                act = sat;
        else if (act < -sat)
                act = -sat;
        prev_diff = diff;

        return act;
}
```

```
//
    ==========================================================================================

/** @file pid.h
 *    This file contains pid control
*/
//
    ==========================================================================================

// This define prevents this .H file from being included multiple
     times in a .CPP file
#ifndef _pid_h_
#define _pid_h_

#include "emstream.h"                            // Header for serial
    ports and devices
#include "FreeRTOS.h"                            // Header for the
    FreeRTOS RTOS
#include "task.h"                                // Header for
    FreeRTOS task functions
#include "queue.h"                               // Header for
    FreeRTOS queues
#include "semphr.h"                              // Header for
    FreeRTOS semaphores
//
    ----------------------------------------------------------------------------------------

/** @brief   This class runs handles pid control.
 *  @details This class takes in the difference between where we
     want to motor to be and
 *                       where it is. From this it calculates,
     combines and returns proportional, integral,
 *           and derivative values that are optimized to get the
     motor to the desired position
 *                       in a timely manner.
 */

class pid
```

```
{
        protected:
                emstream* ptr_serial_pid;      // pointer to
                    serial port for printing
28              uint16_t Kp;                               //
                    constant for proportional control
                uint16_t Ki;                               //
                    ditto for integral
                int16_t prev_diff;
                uint16_t Kd;                               //
                    ditto for derivative
                int16_t sat;                               //
                    saturation limit of power to motor
33              int16_t iaccumulate;
                int16_t act;

    public:
                // sets up the pid for use
38              pid (emstream* p_serial_port, uint16_t Kp_in,
                    uint16_t Ki_in, uint16_t Kd_in, uint16_t
                    sat_in);

                // determines action based on passed in
                    difference
        int16_t action (int16_t diff_in);
}; // end of class pid
43
#endif // _pid_h_
```

A.4 class encoder

```
1  //
    ************************************************************************************

/** @file encoderdriver.cpp
 *    This file contains code that initializes pins to interrupt
     for optical encoding
 *        and it includes two ISR for each encoder class that's
     created (this project will
 *        use two encoders so 4 ISR's are defined. The ISR
     contains the logic necessary
6  *        to determine which direction the motor is spinning
     whcih also enables the velocity
 *        to be calculated.
*/
//
    ************************************************************************************


11 #include <stdlib.h>                    // standard library header
     files
```

```
#include <avr/io.h>
#include "rs232int.h"                    // serial port class
#include "encoderdriver.h"               // encoder class

16 //
    ---------------------------------------------------------------------------------------

/** \brief   This constructor sets up the interrupt for the
    optical encoder
 *  \details This constructs passes in everything needed to
     initialize 2 pins to interrupt
 *                     based on signals from an optical encoder
     .
 *  @param   p_serial_encoder The serial port to output language
     to a terminal
21 */

encoder::encoder(emstream* p_serial_port_in)
{
        p_serial_encoder = p_serial_port_in;    // tedium
            incarnate
26
        // initializes current known positions of both encoders
            to 0
        current_azim_pos->put(0);
        current_alti_pos->put(0);

31      // initializes previous channel logic level of each
            encoder to the logic level of each respective pin
        prev_azim_ch1->put(PIND & (1<<PIND0));
        prev_azim_ch2->put(PIND & (1<<PIND1));
        prev_alti_ch1->put(PIND & (1<<PIND2));
        prev_alti_ch2->put(PIND & (1<<PIND3));
36
        // this code attempts to set up D0-3 for ext interrupts
            but it causes the entire fucking universe to implode
        EICRA |= ((1 << ISC00) | (1 << ISC10) | (1 << ISC20) | (1
             << ISC30));
        EIMSK |= ((1 << INT0) | (1 << INT1) | (1 << INT2) | (1 <<
             INT3));
        DDRD &= ~((1 << 0) | (1 << 1) | (1 << 2) | (1 << 3));
41 }


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

/** @brief   This method increments or decrements a shared
    variable depending on the direction
46 *                     the motor turns.
 *  @details This method compares prev_azim_ch1 and prev_azim_ch2
      to chA and chB for use in calculating
```

```
 *                        the velocity the motor is spinning at,
     and it also updates the shared variable
 *                        current_azim_pos, through a define, that
      allows the relative location of the motor to be
 *                        determined. Each two consequtive ISR's
     represent a respective motor, handeld by respective
51  *                        shared variables that are abstracted
     with respective defines. Comments on the logic will only
 *                        be made on the first ISR, to minimize
     repetitiveness.
 */

ISR (INT0_vect)
                     // interrupts on D0 for azimuth
56 {
        bool chA = (PIND & (1 << PIND0));
            // reads current encoder logic lvl on PIND0
        bool chB = (PIND & (1 << PIND1));
            // reads current encoder logic lvl on PIND1

        if ((chA != chB) | (prev_azim_ch1->ISR_get() ==
            prev_azim_ch2->ISR_get()))
61      {
                current_azim_pos->ISR_put(current_azim_pos->
                    ISR_get() + 1);  // motor spinning right
        }
        else if ((chA == chB) | (prev_azim_ch1->ISR_get() !=
            prev_azim_ch2->ISR_get()))
        {
66              current_azim_pos->ISR_put(current_azim_pos->
                    ISR_get() - 1);  // motor spinning left
        }
        if ((chA == prev_azim_ch1->ISR_get()) && (chB ==
            prev_azim_ch2->ISR_get()))
        {
                azim_error->ISR_put(azim_error->ISR_get() + 1);
                                             // error condition,
                    an encoder signal was missed
71      }
        else if ((chA != prev_azim_ch1->ISR_get()) && (chB !=
            prev_azim_ch2->ISR_get()))
        {
                azim_error->ISR_put(azim_error->ISR_get() + 1);
                                     // error condition, an
                    encoder signal was missed
        }
76      prev_azim_ch1->ISR_put(chA);
                                                     // updates
            shared variables for use in next ISR
        prev_azim_ch2->ISR_put(chB);
}
```

```
ISR (INT1_vect)
                    // interrupts on D1 for azimuth
{
        bool chA = (PIND & (1 << PIND0));
            // reads current encoder logic lvl on PIND0
        bool chB = (PIND & (1 << PIND1));
            // reads current encoder logic lvl on PIND1

        if ((chA == chB) | (prev_azim_ch1->ISR_get() !=
            prev_azim_ch2->ISR_get()))
        {
                current_azim_pos->ISR_put(current_azim_pos->
                    ISR_get() + 1);
        }
        else if ((chA != chB) | (prev_azim_ch1->ISR_get() ==
            prev_azim_ch2->ISR_get()))
        {
                current_azim_pos->ISR_put(current_azim_pos->
                    ISR_get() - 1);
        }
        if ((chA == prev_azim_ch1->ISR_get()) && (chB ==
            prev_azim_ch2->ISR_get()))
        {
                azim_error->put(azim_error->ISR_get() + 1);
        }
        else if ((chA != prev_azim_ch1->ISR_get()) && (chB !=
            prev_azim_ch2->ISR_get()))
        {
                azim_error->ISR_put(azim_error->ISR_get() + 1);
        }
        prev_azim_ch1->ISR_put(chA);
        prev_azim_ch2->ISR_put(chB);
}

ISR (INT2_vect)
                    // interrupts on D2 for altitude
{
        bool chA = (PIND & (1 << PIND2));
              // reads current encoder logic lvl on PIND2
        bool chB = (PIND & (1 << PIND3));
              // reads current encoder logic lvl on PIND3

        if ((chA != chB) | (prev_alti_ch1->ISR_get() ==
            prev_alti_ch2->ISR_get()))
        {
                current_alti_pos->ISR_put(current_alti_pos->
                    ISR_get() + 1);
        }
        else if ((chA == chB) | (prev_alti_ch1->ISR_get() !=
            prev_alti_ch2->ISR_get()))
        {
```

```
                current_alti_pos->ISR_put(current_alti_pos->
                    ISR_get() - 1);
        }
        if ((chA == prev_alti_ch1->ISR_get()) && (chB ==
            prev_alti_ch2->ISR_get()))
        {
                alti_error->ISR_put(alti_error->ISR_get() + 1);
        }
        else if ((chA != prev_alti_ch1->ISR_get()) && (chB !=
            prev_alti_ch2->ISR_get()))
        {
                alti_error->ISR_put(alti_error->ISR_get() + 1);
        }
        prev_alti_ch1->ISR_put(chA);
        prev_alti_ch2->ISR_put(chB);
}

ISR (INT3_vect)
                    // interrupts on D3 for altitude
{
        bool chA = (PIND & (1 << PIND2));
            // reads current encoder logic lvl on PIND2
        bool chB = (PIND & (1 << PIND3));
            // reads current encoder logic lvl on PIND3

        if ((chA == chB) | (prev_alti_ch1->ISR_get() !=
            prev_alti_ch2->ISR_get()))
        {
                current_alti_pos->ISR_put(current_alti_pos->
                    ISR_get() + 1);
        }
        else if ((chA != chB) | (prev_alti_ch1->ISR_get() ==
            prev_alti_ch2->ISR_get()))
        {
                current_alti_pos->ISR_put(current_alti_pos->
                    ISR_get() - 1);
        }
        if ((chA == prev_alti_ch1->ISR_get()) && (chB ==
            prev_alti_ch2->ISR_get()))
        {
                alti_error->put(alti_error->ISR_get() + 1);
        }
        else if ((chA != prev_alti_ch1->ISR_get()) && (chB !=
            prev_alti_ch2->ISR_get()))
        {
                alti_error->ISR_put(alti_error->ISR_get() + 1);
        }
        prev_alti_ch1->ISR_put(chA);
        prev_alti_ch2->ISR_put(chB);
}
```

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

/** @file encoderdriver.h
 *    This file contains the header for an optical encoder.
*/
//
    ********************************************************************************

// This define prevents this .h file from being included multiple
     times in a .cpp file
#ifndef _encoder_h_
#define _encoder_h_

#include "emstream.h"                           // Header for serial
    ports and devices
#include "FreeRTOS.h"                           // Header for the
    FreeRTOS RTOS
#include "task.h"                               // Header for
    FreeRTOS task functions
#include "queue.h"                              // Header for
    FreeRTOS queues
#include "semphr.h"                             // Header for
    FreeRTOS semaphores

#include <avr/interrupt.h>

#include "taskbase.h"                           // ME405/507 base
    task class
#include "time_stamp.h"                         // Class to implement
     a microsecond timer
#include "taskqueue.h"                          // Header of wrapper
    for FreeRTOS queues
#include "textqueue.h"                          // Header for a "<<"
    queue class
#include "taskshare.h"                          // Header for thread-
    safe shared data
#include "shares.h"

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

/** @brief   This class reads from an optical encoder.
 *  @details The class uses AVR chip ports as inputs from the
     optical motor encoder and
 *                    generates directional/positional record
     of motion, using ISR's
 */

class encoder
{
```

```
protected:
        emstream* p_serial_encoder;
            // pointer to allow communication

public:
        encoder (emstream* p_serial_port);
        void calc_vel(void);
            // calculates velocity
};
#endif // _encoder_h_
```

A.5 CLASS MOTOR

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

/** @file motordrive.cpp
 *    This file contains a motor driver
*/
//
    ************************************************************************************


#include <stdlib.h>                          // Include standard
    library header files
#include <avr/io.h>
#include "rs232int.h"                        // Include header for
     serial port class
#include "motordrive.h"                      // Include header for
     the motordrive class

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

/** \brief This constructor sets up a motor driver.
 *  \details The communications lines between the microcontroller
      and the motordriver
 *                         that allows the motor to make turns
 *  @param is_azim_motor_in This bool determines whether the
     first or third timer is configured
 *  @param tc_outpin_in Timer compare output for 1 pin
 *  @param tc_outpin_in2 Timer compare output for second pin
 *  @param p_OCR_in Address of firt PWM output compare register
 *  @param p_OCR_in2 Address of second PWM output compare
     register
 *  @param p_serial_port A pointer to the serial port which
     writes debugging info.
 */

```

```
motordrive::motordrive (bool is_azim_motor_in, uint8_t
    tc_outpin_in, uint8_t tc_outpin_in2, volatile uint16_t*
    p_OCR_in,
                                                volatile uint16_t
                                                    * p_OCR_in2,
                                                    emstream*
                                                    p_serial_port
                                                    )
{
        tc_outpin = tc_outpin_in;
              // pwm for "pos" motor direction
29      tc_outpin2 = tc_outpin_in2;               // pwm pin for
            "neg" motor direction
        p_OCR = p_OCR_in;                         // pointer to
            first output compare register
        p_OCR2 = p_OCR_in2;
                      // pointer to second output compare
            register
        is_azim_motor = is_azim_motor_in;                 //
            dependent on which motordriver to construct

34      ptr_to_serial = p_serial_port;            // inputted
            serial point pointer saved to permenant variable

        if (is_azim_motor == 1)
              // sets up pwm on pins 11 and 12
        {
        DDRB |= (1 << tc_outpin);                 // set output
            compareA/B for timer 1
39      DDRB |= (1 << tc_outpin2);                // set output
            compareA/B for timer 1

        TCCR1A |= (1 << WGM10);                   // set fast pwm
             in timer/counter
        TCCR1B |= (1 << WGM12);
        TCCR1B |= (1 << CS11) | (1 << CS10);      // set
            prescaler for timer/counter at /64
44
        TCCR1A |= (1 << COM1B1);
        TCCR1A &= ~(1 << COM1B0);
        TCCR1A |= (1 << COM1A1);
        TCCR1A &= ~(1 << COM1A0);
49      }

        else
                              // sets up pwm on pins 2 and 5
        {
        DDRE |= (1 << tc_outpin);                 // set output
            compareA/B for timer 3
54      DDRE |= (1 << tc_outpin2);
```

```
        TCCR3A |= (1 << WGM30);                    // set fast pwm
             in timer/counter
        TCCR3B |= (1 << WGM32);
        TCCR3B |= (1 << CS31) | (1 << CS30);       // set
            prescaler for timer/counter at /64

        TCCR3A |= (1 << COM3B1);                   // sets pwm so
            0 is "on" and 1 is "off" for motor 1
        TCCR3A &= ~(1 << COM3B0);
        TCCR3A |= (1 << COM3A1);
        TCCR3A &= ~(1 << COM3A0);
        }

        brake();                                   // sets pwm to
            0 for start-up
}


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

/** @brief   This method sets the direction and PWM power of a
    motor
 *  @details Takes a 16 bit signed number and uses it as power
     PWM of the motor
 *  @param   power Signed motor power input
 */

void motordrive::set_power (int16_t power)
{
        if(power >= 0)
        {
                *p_OCR2 = 0;                               // turn
                     off half the H-brdige
                *p_OCR = power;                            // PWM
                    the other half at power
        }
        else if(power < 0)
        {
                *p_OCR = 0;                                    //
                    turn off other half (direction control)
                *p_OCR2 = power*(-1);         // PWM the other
                    other half at power
        }
        return;
}


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

/** @brief   This method brakes the motor
```

```
94   *  \details Brakes the motor by setting PWM to 0

     */

    void motordrive::brake ()
99  {
            // sets power to 0
            *p_OCR = 0;
            *p_OCR2 = 0;
            return;
104 }
```

```
1   //
        ======================================================================================
    /** @file motordrive.h
     *    This file contains a motor driver for the ME 405 board and
         allows
     *    the control of up to two motors by the onboard drivers.
    */
6   //
        ======================================================================================

    // This define prevents this .H file from being included multiple
         times in a .CPP file
    #ifndef _motor_h_
    #define _motor_h_
11
    #include "emstream.h"                        // Header for serial
        ports and devices
    #include "FreeRTOS.h"                        // Header for the
        FreeRTOS RTOS
    #include "task.h"                            // Header for
        FreeRTOS task functions
    #include "queue.h"                           // Header for
        FreeRTOS queues
16  #include "semphr.h"                          // Header for
        FreeRTOS semaphores
    //
        -------------------------------------------------------------------------------------
    /** @brief   This class runs the onboard motor drivers on the ME
        405 board.
     *  @details Motor driver interface. Allows control of a motor on
          each of the
     *              two drivers on board. Setting power in an 16 bit
          signed number or braking
21   *                      in an 8 bit unsigned number.
     */

    class motordrive
    {
```

```
        protected:
                emstream* ptr_to_serial;       // pointer to
                    serial port for printing
                volatile uint16_t* p_OCR;      // address of pwm
                    output to chip (duty cycle)
                volatile uint16_t* p_OCR2;     // address of pwm
                    output to chip (duty cycle)
                uint8_t tc_outpin;             // timer control
                    output
                uint8_t tc_outpin2;            // timer control
                    output
                bool is_azim_motor;

    public:
                // The constructor sets up the motordrive for use
                    .
                // emstream* parameter is used to specify which
                    serial to print debugging
                // info to
                motordrive (bool is_azim_motor_in, uint8_t
                    tc_outpin_in, uint8_t tc_outpin2, volatile
                    uint16_t* p_OCR_in,
                                        volatile uint16_t*
                                            p_OCR_in2, emstream*
                                            p_serial_port);

                // This function sets the power of a specified
                    motor, as a signed 16 bit number
        void set_power (volatile int16_t power);

                // This function sets the braking of a specified
                    motor, as a signed 8 bit number
                void brake ();
}; // end of class motordrive

#endif // _motor_h_
```

A.6 CLASS ADC

```
//
    *************************************************************************************
/** @file adc.cpp
 *    This file contains a very simple A/D converter driver.
*/
//
    *************************************************************************************

#include <stdlib.h>                         // Include standard
    library header files
```

```
#include <avr/io.h>
#include "rs232int.h"                          // Include header for
     serial port class
#include "adc.h"                               // Include header for
     the A/D class

12 //
    ----------------------------------------------------------------------------

/** \brief Constuctor sets up an A/D converter.
 *  \details The A/D is made ready so that when a  method such as
      @c read_once() is
 *                called, correct A/D conversions can be performed
     .
 *  @param p_serial_port A pointer to the serial port which
     writes debugging info.
17 */

adc::adc (emstream* p_serial_port)
{
        ptr_to_serial = p_serial_port;
22
        ADCSRA |= (1<<ADEN);  // set the ADC enable bit, ADEN
        ADCSRA &= 0b11111000; // clear the last three bits of
            ADCSRA before setting the prescaler
        ADCSRA |= 0b00000101; // set the clock prescaler to a
            division of 32
        ADMUX |= (1<<REFS0);  // set the reference as AVCC with
            external cap at AREF pin
27 }


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

/** @brief   This method takes one A/D reading from the given
    channel and returns it.
32 *  @details This reads and concatenates the values in the high
     and low registers of the A/D
 *  @param   ch The A/D channel which is being read must be from
     0 to 15
 *  @param   even_more_adc A bool that allows ADCs 7 through 15
     to be read
 *  @return  The result of the A/D conversion
 */
37
uint16_t adc::read_once (uint8_t ch, bool even_more_adc)
{
        uint8_t timeout = 200;                          //
            appropiate timeout value that allows standard
            conversions
```

```
        ADMUX &= 0b11000000;                            // clears
             the ADMUX channel selection bits
42      ADCSRB &= (1 << MUX5);
        ADCSRB |= (even_more_adc << MUX5);      // set for ADCs
            7-15
        ADMUX |= ch;                                    // sets
            the ADMUX channel selection bits
        ADCSRA |= (1<<ADSC);                        // starts ADC
             conversion

47      while (ADCSRA&(1<<ADSC) && timeout != 0)
        {
                timeout--;              // waits for end of
                    conversion or timeout
        }

52      if (timeout == 0)           // tells user if timed out
                DBG (ptr_to_serial, "ADC time out"<< endl);

        return ADCL + (ADCH << 8); // returns a concatenated
            conversion value
}
57

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

/** @brief   This averages the A/D channel output.
 *  \details This calls read_once and averages them over the
     number of times specied by samples
62 *  @param   channel Specifies channel number
 *  @param   samples Specifies the number of samples to take
 *      @param   even_more_adc Decides whether first 8 or last 8
     ADCs are initialized in read_once
 *  @return  An average of the A/D samples
 */
67
uint16_t adc::read_oversampled (uint8_t channel, uint8_t samples,
     bool even_more_adc)
{
        uint16_t average = 0;

72      if (samples > 64)                           // limits sample
            size to 64 and gives warning
        {
                DBG (ptr_to_serial, "limited to 64 samples"<<
                    endl);
                samples = 64;
        }
77
        for (uint8_t counter = samples; counter > 0; counter--)
        {
```

```
                average += adc::read_once(channel, even_more_adc)
                    ; // adds up the requested samples
        }
82
        average = average/samples;              // divides by
            number of samples
        return (average);
}
```

```
//
    ==========================================================================================
/** @file adc.h
 *    This file contains a very simple A/D converter driver. The
     driver is hopefully
 *    thread safe in FreeRTOS due to the use of a mutex to
     prevent its use by multiple
5  *    tasks at the same time. There is no protection from
     priority inversion, however,
 *    except for the priority elevation in the mutex.
*/
//
    ==========================================================================================

10 #ifndef _adc_h_
            // prevents multiple inclusions
#define _adc_h_

#include "emstream.h"                       // Header for serial
    ports and devices
#include "FreeRTOS.h"                       // Header for the
    FreeRTOS RTOS
15 #include "task.h"                         // Header for
    FreeRTOS task functions
#include "queue.h"                          // Header for
    FreeRTOS queues
#include "semphr.h"                         // Header for
    FreeRTOS semaphores

//
    ----------------------------------------------------------------------------------------
20 /** @brief   This class runs the A/D converter on an AVR
    processor.
 *  @details This is the header for the class that runs an A/D
     converter and it passes
 *                     a means of communicating and some method
      prototypes for reading the ADC
 *                     once and more than once.
 */
25
class adc
```

```
{
        protected:
                emstream* ptr_to_serial;        // pointer that
                    allows printing
30
    public:
                // The constructor sets up the A/D converter for
                    use. The "= NULL" part is a
                // default parameter, meaning that if that
                    parameter isn't given on the line
                // where this constructor is called, the compiler
                     will just fill in "NULL".
35              // In this case that has the effect of turning
                    off diagnostic printouts
                adc (emstream* = NULL);

                // This function reads one channel once,
                    returning the result as an unsigned
                // integer; it should be called from within a
                    normal task, not an ISR
40      uint16_t read_once (uint8_t, bool);

                // This function reads the A/D lots of times and
                    returns the average. Doing so
                // implements a crude sort of low-pass filtering
                    that can help reduce noise
                uint16_t read_oversampled (uint8_t, uint8_t, bool
                    );
45
};

#endif // _adc_h_
```