**Embedded Control Car/Blimp Report**

|  |  |
| --- | --- |
| Michael Stark | Michael Wilkins |
| David Melecio-Vázquez | Alan Schimmel |

Section 4, Side A (TA: Xuemei Gao)

Due: 3 October 2010

Table of Contents

[Introduction 2](#_Toc279157564)

[Hardware Description 3](#_Toc279157565)

[I2c Bus 3](#_Toc279157566)

[Servo Motor 3](#_Toc279157567)

[Electronic Compass 4](#_Toc279157568)

[Ranger 4](#_Toc279157569)

[Buffer Chip 4](#_Toc279157570)

[Digital Peripherals 5](#_Toc279157571)

[Software Description 6](#_Toc279157572)

[Initialization 6](#_Toc279157573)

[Use of the PCA 8](#_Toc279157574)

[Pulse Streams 8](#_Toc279157575)

[Use of SMBus 9](#_Toc279157576)

[Reading Analog Input 10](#_Toc279157577)

[Control Loops 11](#_Toc279157578)

[Timing 12](#_Toc279157579)

[Results & Conclusions 13](#_Toc279157580)

[Discussion 13](#_Toc279157581)

[Optimal Gains 15](#_Toc279157582)

[Other Experimental Gains 16](#_Toc279157583)

[Appendix A: References 19](#_Toc279157584)

[Appendix B: Circuit Schematic 20](#_Toc279157585)

[Appendix C: Flowchart 22](#_Toc279157586)

[Appendix D: C Code 24](#_Toc279157587)

[Appendix E: Division of Labor 41](#_Toc279157588)

# Introduction

In order control any embedded control system the necessary hardware components and software configurations must be set so that the wanted system behaves as desired. For this lab the code for the blimp was designed first by testing it on the car. The hardware had to be manually wired in for the car (the blimp already has the wires and components built into the system). The purpose of this lab is to achieve control of the blimp. While it is in the air it will hover over its neutral range reading and correct any errors while also maintaining a desired heading. To correct the blimp orientation three fans will be used: one steering fan to control the heading, and two thrust fans to control the height (the range reading) from the ground.

In order to achieve control of this system several gains have to be adjusted to get the desired response in the shortest amount of time as possible. Before attempting anything with the blimp an understanding of the wiring and software configurations had to be built. The components (hardware and software) where all first made to control the car. The controllers in this lab map the envelope of flight for the blimp by using pulse width modulation signals. This variation in pulse width (eventually) changes the response of our system. This response is then further modified by control algorithms in the code until a desired and stable response is achieved.

# Hardware Description

The hardware is essentially the collection of tools that carry out the bidding of the software. The hardware consists of the speed controller, servo motor, compass, ranger, serial bus, and the buffer chip. By controlling the fans/wheels and being able to gauge the surroundings, these components collectively are able to control the movement of the car and the blimp.

## I2c Bus

The serial bus used in this system is the i2c bus. There are four wires used for the i2c bus: power, ground, SDA, and SCL. The i2c bus runs synchronous communication with one master and multiple slave devices at a time. The clock is sent through the SCL wire and used to synchronize the slaves to the master device. When the clock is low, the data line (SDA) gets set high or low. When the clock is high, the slave devices read this high/low bit. This is a write operation. Data can also be sent from the slave devices to the master in the same way; this is called a read operation. In this manner, data is transferred through the SDA wire of the bus.

## Servo Motor

The servo motor is simply a device that takes a pulse width and rotates accordingly. It is sensitive to the amount of time the pulse is high. A certain pulse width will correspond with a certain rotation angle on the motor. Because only a limited number of pulse widths are useful, a range was determined that the pulse widths sent to the servo motor must be within. For the gondola, the servo motor is used to adjust the angle of the thrust fans. The speed controller, like the servo motor, takes a pulse width and spins accordingly. Different pulse widths correspond to different rotational velocities of the motor. There is a separate pulse width range for the speed controller. For the car, the speed controller takes a pulse width, and adjusts the speed the wheels are rotating, thus determining the speed of the car.

## Electronic Compass

The job of the electronic compass is the same as that of a normal compass, except the electronic compass can convert the heading to a digital signal to be read by the microcontroller. This is useful for determining how far off of the desired heading the vehicle is. Once the difference between the desired heading and the actual heading has been determined, equations can be used to determine the pulse width needed for the optimal angle of whatever is controlling the direction of the vehicle. With the car, the compass works with the servo motor to control the steering of the wheels. With the gondola, the compass works with one of the speed controllers to control the tail fan, which controls the heading of the blimp.

## Ranger

The ranger’s function is to determine the distance the vehicle is from something in a certain direction. It sends out a sound wave, and then determines the time it took to return to it. With this time, the ranger can use the velocity of the wave to determine the distance from the ranger to whatever it rebounded off of. This is useful for figuring out the altitude of the blimp. With this knowledge, blimp can be moved to its desired height. For the car, the ranger works with the speed controller to set the wheels to the appropriate speed. For the blimp, the ranger works with the speed controller to adjust the power of the thrust fans until it reaches it’s desired altitude.

## Buffer Chip

The buffer chip protects components that may potentially be ruined if an improper voltage is directed to them. For example, if a component was given too much voltage, it might blow, burn out, or be ruined in some other way. With a buffer chip in place, the component will not be ruined. Also, the buffer chip is a nice way to organize all the wires leading into and out of all the different components.

## Digital Peripherals

A number of digital peripherals are used in this setup, like SDA and CEX0, that do not have a preset I/O pin. Because of this, a crossbar is used to assign a port and a pin number to each peripheral that will be used. The software must initialize each group of peripherals for the crossbar to know which peripherals it will need to assign port pins for. Once the crossbar knows which peripherals are being used, it then assigns them a port pin based on a priority table. There is a list of all the peripherals available that is ordered from the highest priority to the lowest. The crossbar will go through this list, and for every peripheral that is initialized it will be assigned a port pin, starting at port 0 pin 0. The next peripheral would then be assigned P0.1, then P0.2, etc. Once bit 7 of port 0 has been used, the next peripheral will be assigned P1.0, then P1.1, and so on.

# Software Description

The software for the car handled the control systems for steering the car and powering the motor, based on sensor readings from the ultrasonic ranger and the electronic compass. Input and output was performed through an attached LCD screen and numerical keypad. The system increased forward speed if the ranger read a relatively close reading (lower than 40 cm), and increased reverse speed if the ranger read a relatively far reading (larger than 50 cm). The system kept the car immobile (i.e. set the motor in neutral) for ranges between 40 and 50 cm. Steering was also achieved via software - after a desired heading was input, the compass would read a current heading. Depending on the difference between these two measurements, the car would steer too the left or right, either fully turning the wheels (if the desired heading was far away - say, 90 degrees of error or more) or only partially (if the desired heading was close - say, within 90 degrees of error).

The blimp ran almost identically to the car. Instead of speed control, the code for the blimp controlled the power to the thrust fans, or simply thrust. Again, a small reading of the ranger (less than 50 cm) resulted in ‘forward’ power (pushing the blimp up), and a large reading of the ranger (more than 50 cm) resulted in ‘reverse’ power (pushing the blimp down). This keeps the blimp hovering at a desired height of 50 cm. Steering was identical - instead of turning wheels left and right, the steering pulse width powered a tail fin in either forward or reverse to turn left or right.

## Initialization

Several components of the system had to be initialized before they could be used. In particular, several devices were used for output, and one for input - these had to be properly configured. A single pin on Port 1 was designated for analog input - this wire was connected to the battery for the car and blimp, in order to read the current voltage level of the battery. Two slide switches, connected to pins on Port 3, toggled the motor/thrust and steering on and off. These port pins had to be initialized as digital inputs. Finally, the pulse widths for the motor/thrust and steering had to be designated as output ports.

In order to generate varying pulse widths, the Programmable Counter Array (PCA) was used (PCA0, to be precise). In the PCA initialization function, PCA0 was enabled, set to use the internal system clock divided by 12 (SYSCLK/12) as a counter reference, and set to suspend if the microcontroller enters idle mode. PCA0 Counter/Timer Overflow interrupt requests were also enabled here. Finally, two capture control modules (CPM0 for power/thrust, CPM2 for steering) were enabled, set the use 16-bit pulse-width modulation (PWM), and had their associated comparator function enabled.

Timing is achieved through the use of interrupts. An interrupt occurs every time some significant event happened. In this project, an interrupt was triggered when the PCA counter overflowed from 216 to 0. In the interrupt initialization function, PCA0 interrupts were enabled, and interrupts in general were enabled.

The crossbar (XBR) controls which devices/protocols get which ports and pins for their input/output. For example, the communication protocol I2C/device System Bus (SMBus, or SMB) needs two pins to operate. Which specific pins, however, are determined by the XBR setting. In the XBR initialization function, UART0, SPI0, SMB0, and 3 capture control modules were enabled, leaving CP0 and ECI0 disabled.

SMB also had to be initialized - first, the SMB clock was set to use 100 kHz as its operating frequency. The special flag ENSMB was also set, enabling SMB globally.

Similar to the game project, analog/digital conversion (ADC) took place. The analog input of the variable battery voltage was converted to a digital value for the code to interpret and output. In the ADC initialization function, we enabled ADC1, set a gain of 1, set the reference voltage to Vref (2.4 volts), and turned on the internal bias generator and reference buffer.

## Use of the PCA

The PCA is a 16-bit counter, triggered in this lab by the system clock (22.1184 MHz) divided by 12 (SYSCLK/12, or a frequency of 1.8432 MHz). The PCA normally counts from 0 to 65,535 before overflowing back to 0 and triggering an interrupt request. This gives a period of approximately 35.556 ms (28.1250439 Hz) per overflow. However, the steering servo and drive motor of the car (and analogous systems on the blimp) operate at a 20 ms (50 Hz). In order to achieve this timing, the PCA counter was reset to a value larger than 0 (28,672, to be exact). This means that the PCA was counting only 36,864 counts, rather than the full 65,535 counts.

## Pulse Streams

The steering servo, drive motor, and analogous systems on the blimp are all controlled by varying an output between high and low in a regular pattern. The high and low pattern is called a pulse stream. The pulse width of the stream is the amount of time the pulse is high in each cycle. For the steering servo, a high pulse width meant a hard turn to the right, while a low pulse width meant a hard turn to the left. For the drive motor/thrust fans, a high pulse width meant forward/upward power, while a low pulse width meant reverse/downward power.

In software, pulse streams are generated by the capture control modules (CCM) in the following manner. Each CCM holds a 16 bit number. When the PCA counter reaches this number, the CCM changes its output signal from low to high. When the PCA counter overflows, all CCMs change their output signal to low. Thus, by increasing the number help in a CCM, the pulse stream is high less often, and the pulse width decreases. By lowering the number held in a CCM, the pulse width is decreased.

## Use of SMBus

The System Bus is a device allowing communication between a master device and one or more slave devices. In this project, the C8051 Microcontroller was the master device. Slave devices included the ultrasonic ranger, the electronic compass, the LCD Display, and the keypad input device. The communication is serially, meaning one bit at a time, and asynchronously, meaning only one way at a time. Thus, only one device can be communicating at once. The master device controls which device will be communicating.

For a write operation (the master sending information to a slave device), the master sends a STOP signal to halt all current communication. The master then sends the address of the device to which it is writing, and a signal indicating that a write operation is desired. The device at that address then listens for the register number it must write the information to, and finally the information itself. After every byte of information, the slave device must send an acknowledgement (ACK) signal, indicating that it properly read the information. The master device waits for this ACK before writing the next piece of data. Once all information has been sent, the communication is terminated.

For a read operation (a slave device sending information to the master), the master still sends a STOP signal to halt all current communication. The master then sends the address of the device from which it wishes to read, along with a signal for a write operation. The master then writes which register it desires to read from. Once this information is written to the slave device, the master STOPs communication again, then calls out the devices address again. This time, however, the signal is given for a read operation. The slave transmits the requested data to the master until finished and communication is terminated. The master must send an ACK signal to the slave indicating a successful read of each byte of data.

An example of a read operation is when the system reads the current heading from the compass. An example of a write operation is when the system tells the ranger to start a new sonic ‘ping’.

## Reading Analog Input

In this project, we desired to know the current battery level in the car/blimp. The battery’s voltage level varied between 12 V and 0 V. A voltage divider circuit was designed in hardware to limit this range to between 2.4 V and 0 V, keeping the input proportionally correct (i.e. if the battery level was 6 V, our divider circuit would limit the voltage to 1.2 V). The output of this divider circuit was our analog input. A simple formula is applied to the input voltage to generate a number between 0 and 255 (an unsigned character). Again, proportions were maintained, so an input reading of 1.2 V would result in a digital value of about 127. In code, we then reversed the process to determine the exact voltage level of the battery between 0 V and 12 V.

Note: for safety, the voltage divider circuit was designed for a maximum input voltage of 15 V. This way, the analog/digital converter would not be overloaded with a voltage higher than 2.4 V.

## Control Loops

For the steering, it was necessary to vary the pulse width based on the current heading read and the desired heading. For power/thrust, it was necessary to vary the pulse width based on the range read and the desired height. In each case, an error was calculated (desired - actual) and used in a formula to determine a correct pulse width. The simplest form was a proportional control loop: PW = PWNeutral +Kp \* error. For the car system, because friction between the ground and the wheels provided natural damping, this was sufficient. However, for the blimp, there was no contact with the ground, and there was significantly less friction (and thus, less damping). In this case, a simple proportional control loop resulted in endless oscillations. For steering, the blimp would overshoot its desired heading and reverse the tail fan, only the overshoot again and start all over. For thrust, the blimp would move higher than its desired height and reverse thrust, only to fall below its desired height again and start over.

In order to account for this, a derivative term is added to the equation. The proportional plus derivative control loop (PD control) looks like: PW = PWNeutral+Kp\*error +Kd\*derrordt, where the de/dt term was approximated by error - previous error. This allowed the system to make changes less rapidly as the desired result was approached. For example, as the blimp lifted up closer to 50 cm height, the thrust fans would become less powerful, and even begun reverse thrust, in order to prevent the blimp from overshooting the 50 cm height mark by too much. This stabilized the system around 50 cm fairly quickly, rather than resulting in endless oscillations.

## Timing

The ultrasonic ranger operates by sending out a short burst of sound and using simple echolocation to determine the distance from the nearest object based on the time it takes for the ‘ping’ to bounce back to the sensor. Each ping takes about 65 ms to complete. Since our system was designed around a 20 ms period, we signaled for a ping to start every 80 ms, or every 4 PCA overflows.

The electronic compass should only be read every 40 ms in order to get an accurate compass reading. We read from the compass every 40 ms, or every 2 PCA overflows.

This timing was achieved by having two separate counters - one for the ranger, and one for the compass. Each PCA overflow incremented these counters, but a corresponding flag (new heading or new range) was set based on their values (if the ranger counter is greater than 4, signal for a new range and reset the counter. If the compass counter is greater than 2, signal for a new reading and reset the counter.)

Debug information was also printed. In order to avoid having too much data, this information was only outputted every 400 ms. A third counter was used to keep track of this - if this counter was greater than 20, we output data and reset the counter.

# Results & Conclusions

## Discussion

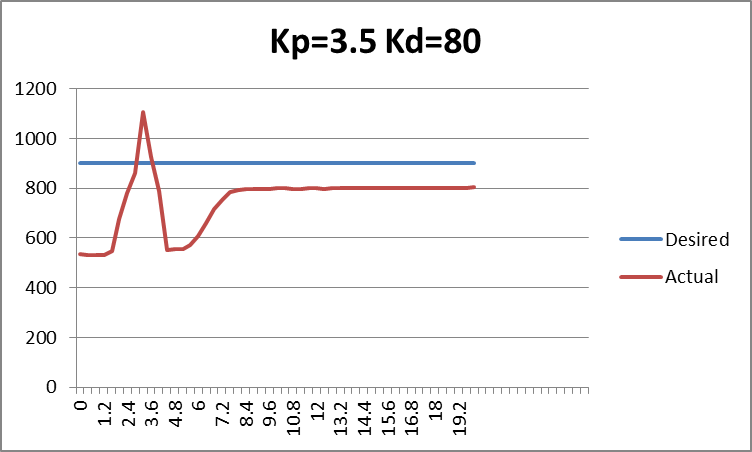
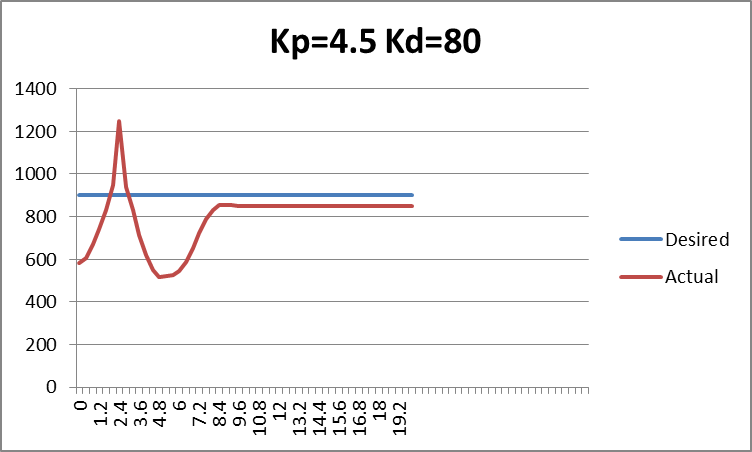
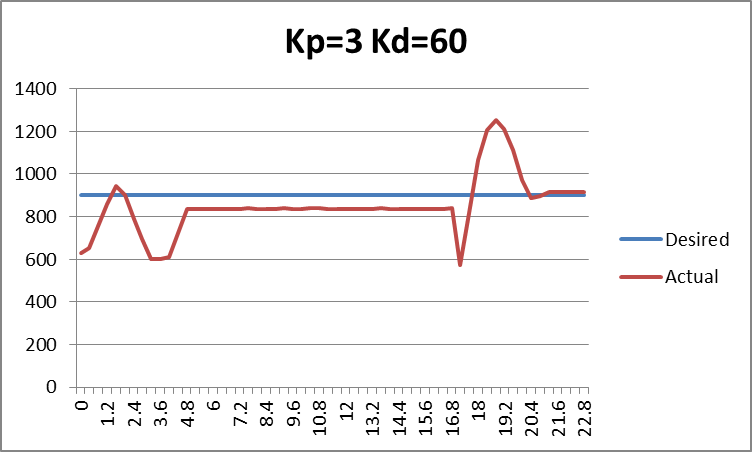
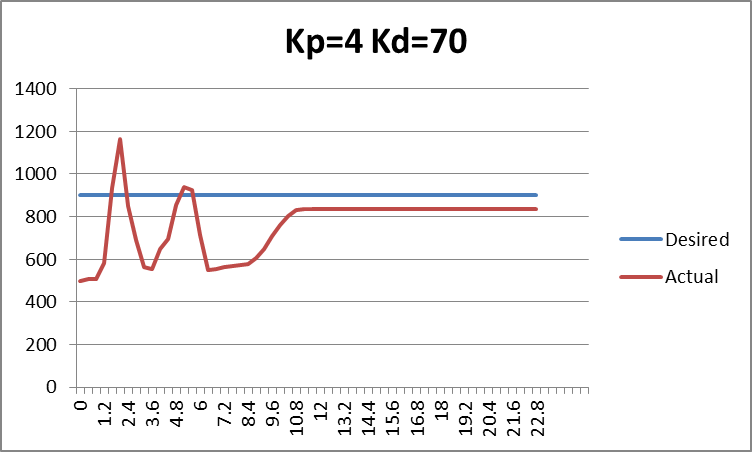
The following observations were made during this laboratory using the help of these graphs below. It was noticed that when proportional gain was set at a low value the gondola didn’t have enough power to actually rotate itself around. When the value was raised a high point was found in which there would be too much power and the gondola would spin around wildly and overshoot the target. After tuning Proportional gain, it was found that with a low derivative gain, error would not be corrected thus leaving the gondola off course. When derivative gain was increased it was found that it would start to oscillate if it was gained too high because of the inertia of the system and it trying to correct for error in location.

A problem encountered was one that is not so obvious, it was noticed that the performance of each gondola varied (when testing the same gains, some gondolas worked Just as we desired it to and hit the target direction, while others overshot or undershot the desired target. This can be attributed to a multitude of factors. First at the most simple level, the tables may not have all been level thus gravity was acting against each gondola differently. The second factor is the motors; cheap manufactured motors do not perform the same and can vary greatly in max speed thusly, given the same voltage 2 different motors might not spin at the same speed thus effecting thrust. Third, the propellers are not balanced thus each propeller is imperfect in its own way which can effect thrust. Fourth, the motor controllers AKA Electronic Speed Controllers, each one is a little different based on wire length to the motors quality of mosfets used and calibration on the controller itself. Although all of these factors may have only influenced the thrust by just a small amount, all together the effect was very apparent.

Many things were learned about hardware, and the imperfections and lack of consistency between multiple of the same products i.e. motors, speed controllers, propellers. Never the less with a well-tuned PID loop these inconsistencies can be accounted for at least to some extent. In closing, the best solution to all of these problems is to have proportional gain high enough so even in a slower less powerful motor, it will still have the power to move the gondola to the desired position, and a derivative high enough to account for error based on underpowered motors or propellers or gravity from uneven tables, while still being below the threshold of an oscillating effect.

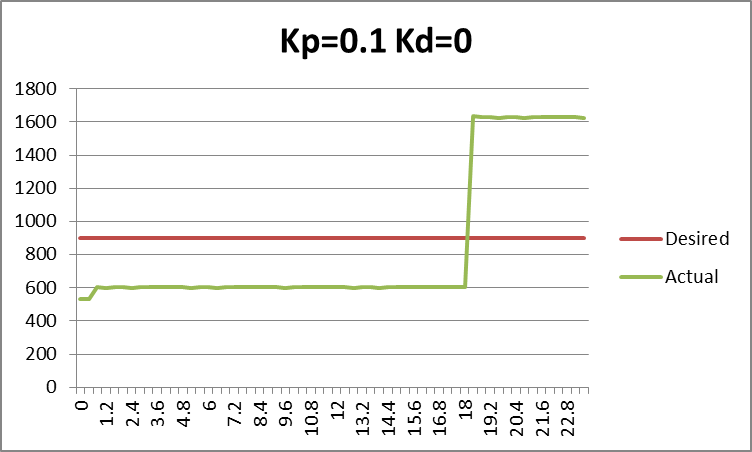
Through trial and error, the cases in figures 1-4 below were deemed to be the optimal choices for derivative and proportional gain.

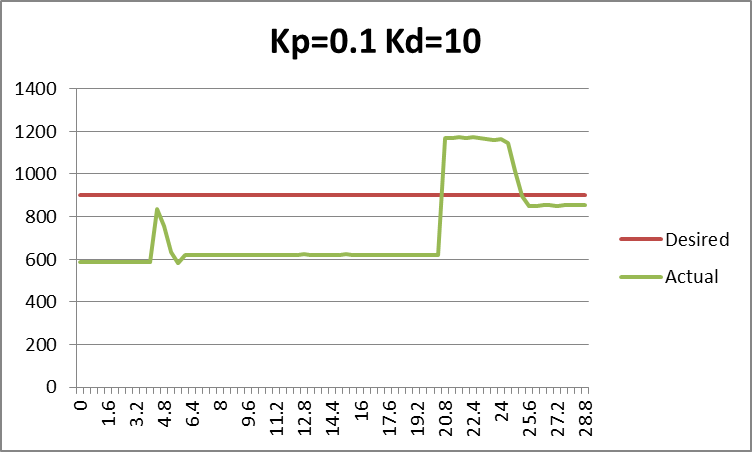
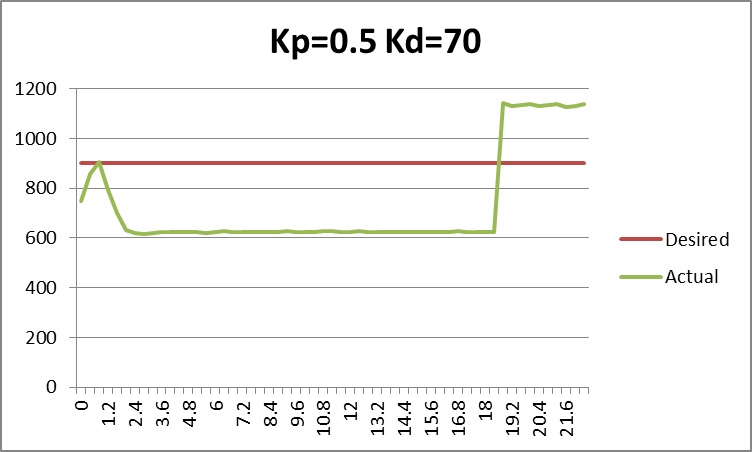
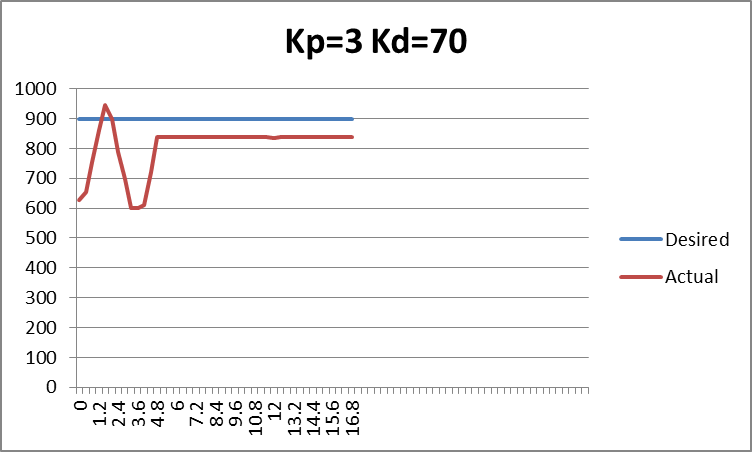
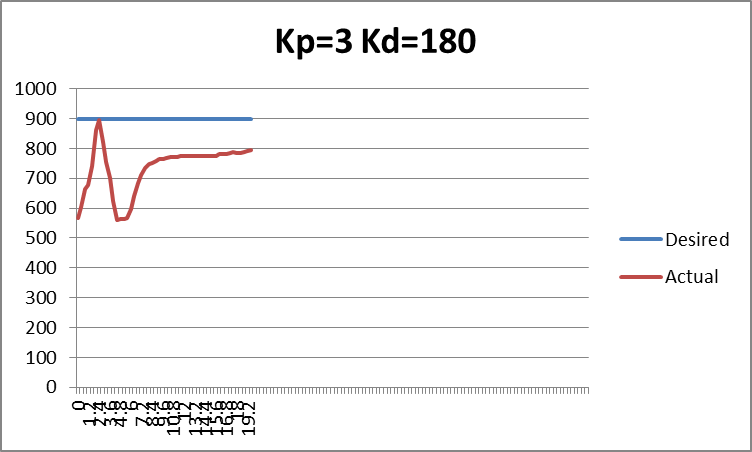
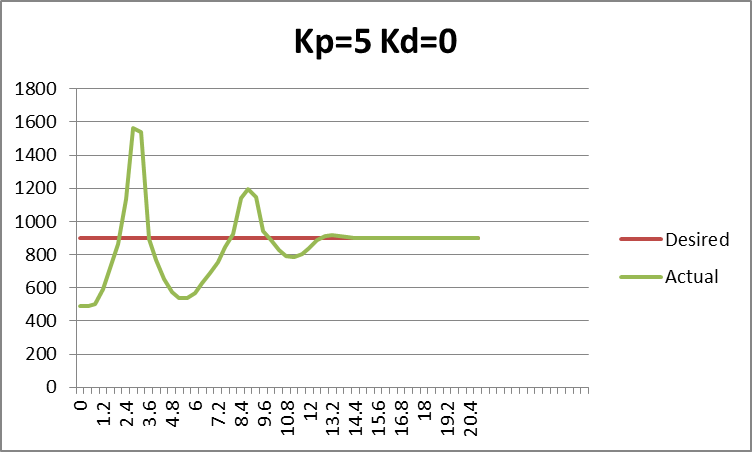
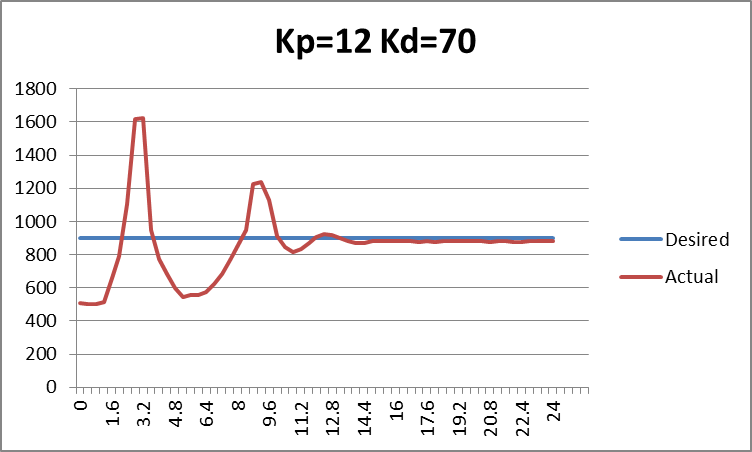
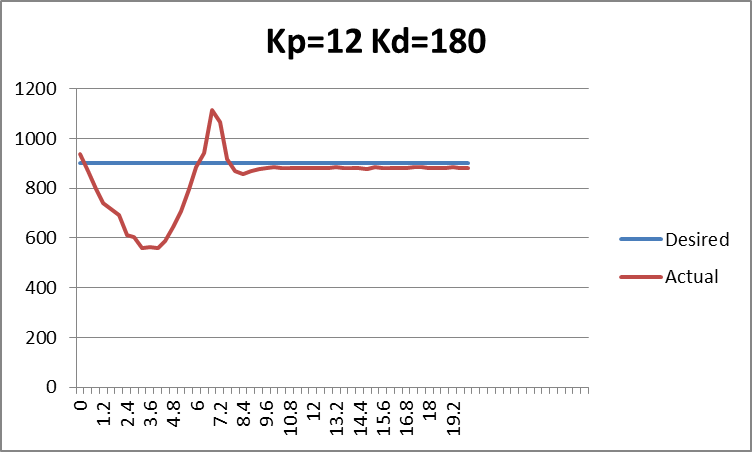
## Optimal Gains



Figures 1-4 (clockwise)

## Other Experimental Gains

  
Figure 5

  
Figure 6  
  
Figure 7  
  
Figure 8  
  
Figure 9  
  
Figure 10  
  
Figure 11  
  
Figure 12

# Appendix A: References

* Laboratory Introduction to Embedded Control Lab Manual v13.2

# Appendix B: Circuit Schematic

Switch schematic with this page

# Appendix C: Flowchart

Switch flowchart with this page

# Appendix D: C Code

/\*

\* Names: Michael Stark, David Melecio-Vazquez, Mike Wilkins, and Alan Schimmel

\* Section: 4 (Side A)

\* Date: 16 November 2010

\* File name: lab5.c

\* Program description:

\*/

#include <stdio.h>

#include <c8051\_SDCC.h>

#include <i2c.h>

#define THRUST\_PW\_MIN 2027 // 1.1 ms pulsewidth

#define THRUST\_PW\_NEUT 2764 // 1.5 ms pulsewidth

#define THRUST\_PW\_MAX 3502 // 1.9 ms pulsewidth

#define STEER\_PW\_MIN 2000

#define STEER\_PW\_NEUT 2750

#define STEER\_PW\_MAX 3500

#define THRUST\_ANGLE\_NEUTRAL 3300

#define DESIRED\_HEIGHT 50 // Desire a height of 50 cm.

#define MAX\_LEN 4

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

// 8051 Initialization Functions

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

void Port\_Init(void); // Initialize input/output ports.

void Interrupt\_Init(void); // Set up PCA0 interrupts.

void XBR0\_Init(void); // Initialize crossbar.

void SMB\_Init(void); // Initialize system bus.

void PCA\_Init (void); // Initialize the PCA counter.

void ADC\_Init(void); // Initialize A/D Conversion

void angle(void); // Set angle of thrust fans

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

// Thrust functions

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

int Read\_Ranger(void); // Read the ultrasonic ranger.

// Vary the thrust fans PW based on the range in cm.

int Thrust\_Fans(int range, unsigned int kp, unsigned int kd,

signed int prev\_error);

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

// Steering functions

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

int ReadCompass(void); // Read the electronic compass

// Vary the steering PW based on the heading in degrees and the gain constants.

signed int Steer(int current\_heading, unsigned int kp, unsigned int kd,

signed int prev\_error);

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

// Other functions

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

unsigned char Read\_Port\_1(void); // Performs A/D Conversion

float ConvertToVoltage(unsigned char battery);

void LCD\_Display(unsigned int current\_heading, int range);

unsigned int GetHeadingPGain(void); // Retrieve the user's input for the

// proportional steering gain

unsigned int GetHeadingDGain(void); // Retrieve the user's input for the

// derivative steering gain

unsigned int GetPowerPGain(void); // Retrieve the user's input for the

// proportional power gain

unsigned int GetPowerDGain(void); // Retrieve the user's input for the

// derivative power gain

unsigned int GetDesiredHeading(void); // Retrieve the user's input for the

// desired heading

int atoi(char \*buf); // Converts a string of characters to the equivalent

// integer, or -1 if invalid.

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

// Global Variables

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

unsigned int MOTOR\_PW = 0; // Pulsewidth to use for the drive motor.

unsigned int STEER\_PW = 0; // Pulsewidth to use for the steering motor

unsigned char D\_Counts = 0; // Number of overflows, used for setting new\_range

unsigned char S\_Counts = 0; // Number of overflows, used for setting new\_heading

unsigned char Overflows = 0; // Number of overflows, used for waiting 1 second

unsigned char new\_range = 0; // flag to start new range reading

unsigned char new\_heading = 0; // flag to start new direction reading

sbit at 0xB6 DSS; // Slide switch controlling the Thrust\_Fans function

sbit at 0xB7 SSS; // Slide switch controlling the Steer function.

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

// XData Constants

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

xdata unsigned int desired\_heading; // Desired direction

xdata unsigned int heading\_p\_gain; // Proportional gain constant for steering.

xdata unsigned int heading\_d\_gain; // Derivative gain constant for steering.

xdata unsigned int thrust\_p\_gain; // Proportional gain constant for power.

xdata unsigned int thrust\_d\_gain; // Derivative gain constant for power.

xdata unsigned int PCA\_COUNTS = 36864; // number of counts in 20 ms. Constant.

xdata unsigned char info[1] = { 0x51 };

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

// Main Function

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

void main(void) {

int range = 0; // Result of the read operation

int current\_heading = 0; // Heading read by the electronic compass

signed int steer\_prev\_error = 0;

signed int thrust\_prev\_error = 0;

// System initialization

Sys\_Init();

putchar(' ');

Port\_Init();

Interrupt\_Init();

XBR0\_Init();

SMB\_Init();

PCA\_Init();

ADC\_Init();

// print beginning message

printf("\rEmbedded Control Gondola Control\r\n");

lcd\_clear();

// set initial value

MOTOR\_PW = THRUST\_PW\_NEUT;

PCA0CPL2 = 0xFFFF - MOTOR\_PW;

PCA0CPH2 = (0xFFFF - MOTOR\_PW) >> 8;

PCA0CPL3 = 0xFFFF - MOTOR\_PW;

PCA0CPH3 = (0xFFFF - MOTOR\_PW) >> 8;

PCA0CPL3 = 0xFFFF - THRUST\_ANGLE\_NEUTRAL;

PCA0CPH3 = (0xFFFF - THRUST\_ANGLE\_NEUTRAL) >> 8;

STEER\_PW = STEER\_PW\_NEUT;

PCA0CPL0 = 0xFFFF - STEER\_PW;

PCA0CPH0 = (0xFFFF - STEER\_PW) >> 8;

Overflows = 0; // Overflows is incremented once every 20 ms. 1 s = 1000 ms.

// 1000 / 20 = 50. Wait 50 counts

while (Overflows < 50);

angle();

desired\_heading = GetDesiredHeading();

heading\_p\_gain = GetHeadingPGain();

heading\_d\_gain = GetHeadingDGain();

thrust\_p\_gain = GetPowerPGain();

thrust\_d\_gain = GetPowerDGain();

lcd\_clear();

Overflows = 0;

while (1) {

if (Overflows > 20) {

printf("%u,%u,%u,%u\r\n", desired\_heading, current\_heading,

DESIRED\_HEIGHT, range);

LCD\_Display(current\_heading, range);

Overflows = 0;

}

if (new\_heading) {

current\_heading = ReadCompass(); // Read the electronic compass

if (!SSS) {

steer\_prev\_error = Steer(current\_heading, heading\_p\_gain,

heading\_d\_gain, steer\_prev\_error);

} else {

// Make the wheels straight

STEER\_PW = STEER\_PW\_NEUT;

PCA0CPL0 = STEER\_PW;

PCA0CPH0 = STEER\_PW >> 8;

}

new\_heading = 0;

}

if (new\_range) {

range = Read\_Ranger(); // Read the ultrasonic ranger

if (!DSS) {

// Change the thrust based on the range read.

thrust\_prev\_error = Thrust\_Fans(range, thrust\_p\_gain, thrust\_d\_gain,

thrust\_prev\_error);

} else {

// Set the motor to neutral.

Thrust\_Fans(DESIRED\_HEIGHT, thrust\_p\_gain, thrust\_d\_gain, 0);

}

i2c\_write\_data(0xE0, 0, info, 1); // Write the ping signal to register 0

// of the ranger

new\_range = 0; // Reset the flag and wait for 80ms

}

}

}

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

// Port\_Init

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

//

// Set up ports for input and output

//

void Port\_Init() {

P1MDIN &= 0xDF; // set P1.5 as an anlog input

P1MDOUT |= 0x0F; // set output pin for CEX0 through 4 in push-pull mode

P1MDOUT &= 0xDF; // set input pin P1.7 in open-drain mode.

P3MDOUT &= 0x3F; // set input pins P3.6 and P3.7 (Slide switches) in

// open-drain mode

P1 |= 0x20; // Set input pin P1.7 (A/D) to high impedance.

P3 |= 0xC0; // Set input pins P3.6 and P3.7 (Slide switches) to high

// impedance

}

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

// Interrupt\_Init

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

//

// Enable proper interrupts.

//

void Interrupt\_Init() {

// IE and EIE1

EA = 1; // Enable interrupts globally

EIE1 |= 0x08; // Enable PCA0 interrupts

}

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

// XBR0\_Init

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

//

// Set up the crossbar

//

void XBR0\_Init() {

// 0001 1111

XBR0 = 0x27; // configure crossbar with UART0, SPI, SMBus, and CEX channels

}

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

// SMB\_Init

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

//

// Set up the system bus

//

void SMB\_Init(void) {

SMB0CR = 0x93; // set SCL to use 100 kHz

ENSMB = 1; // Bit 6 of SMB0CN. Enables the system bus.

}

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

// PCA\_Init

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

//

// Set up Programmable Counter Array

//

void PCA\_Init(void) {

PCA0CN = 0x40;

PCA0CPM0 = 0xC2;

PCA0CPM1 = 0xC2;

PCA0CPM2 = 0xC2;

PCA0CPM3 = 0xC2;

PCA0MD = 0x81;

}

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

// ADC\_Init

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

//

// Initialize the analog to digital conversion.

//

void ADC\_Init(void) {

REF0CN &= 0xF7; // 1111 0111 Configure ADC1 to use VREF

REF0CN |= 0x03; // 0000 0011

ADC1CF = 0x01; // 0000 0001 Set a gain of 1

ADC1CN |= 0x80; // 1000 0000 Enable ADC1

}

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

// Read\_Ranger

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

//

// Reads the latest ping value from the Ultrasonic Ranger, and returns the

// result.

//

int Read\_Ranger(void) {

unsigned char info[2] = {'\0'}; // Space for us to read information from

// ranger

int range = 0; // Inititalize the range value to 0.

unsigned char addr = 0xE0; // Address of the ranger

i2c\_read\_data(addr, 2, info, 2); // Read 2 bytes (size of an unsigned int)

// starting at register 2

range = (((int)info[0] << 8) | info[1]); // Convert the two bytes of data to

// one short int

return range;

}

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

// Thrust\_Fans

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

//

// Vary the pulsewidth based on the user input to change the speed

// of the drive motor.

//

int Thrust\_Fans(int range, unsigned int kp, unsigned int kd, int prev\_error) {

MOTOR\_PW = (long)THRUST\_PW\_NEUT;

MOTOR\_PW += (((long)(DESIRED\_HEIGHT - range) \* 184 \* (long)kp) / 100);

MOTOR\_PW += (((long)(DESIRED\_HEIGHT - range) - (long)prev\_error) \* (long)kd);

if (MOTOR\_PW > THRUST\_PW\_MAX) {

MOTOR\_PW = THRUST\_PW\_MAX;

} else if (MOTOR\_PW < THRUST\_PW\_MIN) {

MOTOR\_PW = THRUST\_PW\_MIN;

}

PCA0CPL2 = 0xFFFF - MOTOR\_PW;

PCA0CPH2 = (0xFFFF - MOTOR\_PW) >> 8;

PCA0CPL3 = 0xFFFF - MOTOR\_PW;

PCA0CPH3 = (0xFFFF - MOTOR\_PW) >> 8;

return (DESIRED\_HEIGHT - range);

}

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

// ReadCompass

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

//

// Fuction to read the electronic compass.

//

signed int ReadCompass() {

unsigned char Data[2]; // array with length of 2

signed int heading; // the heading returned in degrees between 0 and 3599

i2c\_read\_data(0xC0, 2, Data, 2); // reads 2 bytes into Data[]

// combines the two numbers into degrees accurate to 1/10 of a degree

heading = (((signed int)Data[0] << 8) | Data[1]);

return heading; //return heading (in degrees)

}

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

// Steer

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

//

// Function to turn the wheels towards desired heading.

//

signed int Steer(int current\_heading, unsigned int kp, unsigned int kd,

signed int prev\_error) {

int error = (int)((int)desired\_heading -

(int)current\_heading);

// This keeps the error within the -1800 to 1800 range.

if (error < -1800) {

error += 3600;

} else if (error > 1800) {

error -= 3600;

}

STEER\_PW = (long)STEER\_PW\_NEUT + ((long)kp \* (long)error) / 10 +

((long)kd \* (long)(error - prev\_error));

if ((int)STEER\_PW < (int)STEER\_PW\_MIN) {

STEER\_PW = STEER\_PW\_MIN;

} else if ((int)STEER\_PW > (int)STEER\_PW\_MAX) {

STEER\_PW = STEER\_PW\_MAX;

}

PCA0CPL0 = 0xFFFF - STEER\_PW;

PCA0CPH0 = (0xFFFF - STEER\_PW) >> 8;

return error;

}

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

// PCA\_ISR

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

//

// Interrupt Service Routine for Programmable Counter Array Overflow Interrupt

//

void PCA\_ISR ( void ) interrupt 9 {

if (CF) {

// Reset PCA to the correct start value (65,535 - PCA\_COUNTS)

PCA0L = 0xFFFF - PCA\_COUNTS;

PCA0H = (0xFFFF - PCA\_COUNTS) >> 8;

// Increment D\_Counts variable (used for waiting 80 ms)

D\_Counts++;

// Increment S\_Counts variable (used for waiting 40 ms)

S\_Counts++;

// Increment Overflows variable (used for waiting 1 s)

Overflows++;

if (D\_Counts >= 4) {

new\_range = 1; // signal start of read operation

D\_Counts = 0; //Reset Counts

}

if (S\_Counts >= 2) {

new\_heading = 1;

S\_Counts = 0;

}

CF = 0;

} else {

PCA0CN &= 0xC0; // all other type 9 interrupts

}

}

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

// Read\_Port\_1

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

//

// Reads the value on Port 1 Pin 7, and performs A/D conversion to return a

// value between 0 and 255.

//

unsigned char Read\_Port\_1(void) {

AMX1SL = 7; // Set the ADC conversion to read P1.7

ADC1CN &= 0xDF; // 1101 1111 Clear the flag from the previous ADC1 conversion

ADC1CN |= 0x10; // 0001 0000 Start A/D Conversion

while ((ADC1CN & 0x20) == 0x00); // Wait for conversion to be complete

return ADC1; // Return the A/D conversion result

}

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

// ConvertToVoltage

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

//

// Converts the battery value fron digital units to voltage (float).

//

float ConvertToVoltage(unsigned char battery) {

return ((12.0 \* battery) / 255.0);

}

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

// LCD\_Display

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

//

// Displays statistics about the car.

//

void LCD\_Display(unsigned int current\_heading, int range) {

lcd\_clear();

lcd\_print("Heading: %d\nRange: %d cm\nBattery: %d",

current\_heading, range, (Read\_Port\_1() \* 15) / 255);

}

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

// GetHeadingPGain

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

//

// Using the LCD display and keypad, queries the user to determine the

// proportional steering gain constant.

//

unsigned int GetHeadingPGain(void) {

char keypad, i = 0, buf[MAX\_LEN];

buf[MAX\_LEN - 1] = '\0';

lcd\_clear();

lcd\_print("Heading P. Gain? ");

while (1) {

do {

keypad = read\_keypad();

Overflows = 0;

while (Overflows < 1);

} while (keypad == -1);

// Wait until user releases the keypad

do {

Overflows = 0;

while (Overflows < 1);

} while (read\_keypad() != -1);

// keypad now holds 1 character.

if (keypad == '\*' || keypad == '#') {

buf[i] = '\0';

break;

}

buf[i] = keypad;

i++;

if (i == MAX\_LEN - 1) {

break;

}

}

return atoi(buf); // Subtract the value of '0' to get the numeric value

// between 0 and 9.

}

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

// GetHeadingDGain

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

//

// Using the LCD display and keypad, queries the user to determine the

// derivative steering gain constant.

//

unsigned int GetHeadingDGain(void) {

char keypad, i = 0, buf[MAX\_LEN];

buf[MAX\_LEN - 1] = '\0';

lcd\_clear();

lcd\_print("Heading D. Gain? ");

while (1) {

do {

keypad = read\_keypad();

Overflows = 0;

while (Overflows < 1);

} while (keypad == -1);

// Wait until user releases the keypad

do {

Overflows = 0;

while (Overflows < 1);

} while (read\_keypad() != -1);

// keypad now holds 1 character.

if (keypad == '\*' || keypad == '#') {

buf[i] = '\0';

break;

}

buf[i] = keypad;

i++;

if (i == MAX\_LEN - 1) {

break;

}

}

return atoi(buf); // Subtract the value of '0' to get the numeric value

// between 0 and 9.

}

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

// GetPowerPGain

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

//

// Using the LCD display and keypad, queries the user to determine the

// proportional power gain constant.

//

unsigned int GetPowerPGain(void) {

char keypad, i = 0, buf[MAX\_LEN];

buf[MAX\_LEN - 1] = '\0';

lcd\_clear();

lcd\_print("Power P. Gain? ");

while (1) {

do {

keypad = read\_keypad();

Overflows = 0;

while (Overflows < 1);

} while (keypad == -1);

// Wait until user releases the keypad

do {

Overflows = 0;

while (Overflows < 1);

} while (read\_keypad() != -1);

// keypad now holds 1 character.

if (keypad == '\*' || keypad == '#') {

buf[i] = '\0';

break;

}

buf[i] = keypad;

i++;

if (i == MAX\_LEN - 1) {

break;

}

}

return atoi(buf); // Subtract the value of '0' to get the numeric value

// between 0 and 9.

}

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

// GetPowerDGain

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

//

// Using the LCD display and keypad, queries the user to determine the

// derivative power gain constant.

//

unsigned int GetPowerDGain(void) {

char keypad, i = 0, buf[MAX\_LEN];

buf[MAX\_LEN - 1] = '\0';

lcd\_clear();

lcd\_print("Power D. Gain? ");

while (1) {

do {

keypad = read\_keypad();

Overflows = 0;

while (Overflows < 1);

} while (keypad == -1);

// Wait until user releases the keypad

do {

Overflows = 0;

while (Overflows < 1);

} while (read\_keypad() != -1);

// keypad now holds 1 character.

if (keypad == '\*' || keypad == '#') {

buf[i] = '\0';

break;

}

buf[i] = keypad;

i++;

if (i == MAX\_LEN - 1) {

break;

}

}

return atoi(buf); // Subtract the value of '0' to get the numeric value

// between 0 and 9.

}

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

// GetDesiredHeading

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

//

// Using the LCD display and keypad, queries the user to determine the desired

// heading constant.

//

unsigned int GetDesiredHeading(void) {

char keypad, temp;

lcd\_clear();

lcd\_print("Desired heading?\n1) 0 deg 2) 90 deg\n3) 180 deg\n4) 270 deg");

do {

do {

keypad = read\_keypad();

Overflows = 0;

while (Overflows < 1);

} while (keypad == -1);

Overflows = 0;

while (Overflows < 1);

// Wait until user releases the keypad

do {

temp = read\_keypad();

Overflows = 0;

while (Overflows < 1);

} while (temp != -1);

} while (keypad < '1' || keypad > '4'); // Restrict input to 1, 2, 3, or 4

switch (keypad) {

case '1':

return 0;

case '2':

return 900;

case '3':

return 1800;

case '4':

return 2700;

default: // This should never happen

return 0;

}

}

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

// atoi

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

//

// Converts an array of number characters to the equivalent number.

//

int atoi(char \*buf) {

int sum = 0;

char i = 0;

if (buf == NULL) return 0;

while (buf[i]) {

sum \*= 10;

if (buf[i] < '0' || buf[i] > '9') return 0;

sum += buf[i] - '0';

i++;

}

return sum;

}

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

// angle

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

//

// Using the LCD display and keypad, queries the user to vary the thrust fan

// angle.

//

void angle(void)

{

signed char input\_angle;

unsigned int Angle\_PW = THRUST\_ANGLE\_NEUTRAL;

lcd\_clear();

lcd\_print("Print 1 to turn the\nfans left, 3 for\nright, and 2 for\ndone");

while (1) {

do {

input\_angle = read\_keypad();

Overflows = 0;

while (Overflows < 1);

} while (input\_angle == -1);

// Wait until user releases the keypad

do {

Overflows = 0;

while (Overflows < 1);

} while (read\_keypad() != -1);

if (input\_angle == '1') {

Angle\_PW -= 100;

}

else if (input\_angle == '3') {

Angle\_PW += 100;

}

else if (input\_angle == '2') {

break;

}

printf("Angle\_PW: %d\r\n", Angle\_PW);

PCA0CPL1 = 0xFFFF - Angle\_PW;

PCA0CPH1 = (0xFFFF - Angle\_PW) >> 8;

}

PCA0CPL1 = 0xFFFF - THRUST\_ANGLE\_NEUTRAL;

PCA0CPH1 = (0xFFFF - THRUST\_ANGLE\_NEUTRAL) >> 8;

}

# Appendix E: Division of Labor

Michael Stark - Software Description  
 - Results Plots  
 - Circuit Schematic

Michael Wilkins - Hardware Description  
 - Flowchart

Alan Shimmel - Results & Conclusions

David Melecio-Vázquez -Introduction  
 -Formatting & Final Editing