LITEC Car/Gondola Lab Report

Motor Control, Accelerometers, and Control Algorithms

Kathryn DiPippo, Rebecca Halzack, Seth Rutman

TA: Shen Wang
Lab Section: 4 - B Side
Lab Time: TF 11:00 AM - 2:00 PM
Due Date: 5/12/2015

Table of Contents

Introduction

Purpose and Objectives

After mastering basic embedded control with switches and LEDs, we were able to move on to much more intricate projects involving more complicated controls. We learned how to use servo control, understand various other elements with new hardware, and communicate with the microcontroller.

The goal of lab 3 was to create a program that could interact with a given device using the SMBus, and use the information from that device to adjust the servos of the car. One half of lab 3 used an electric compass to adjust the steering servo motor of the car, while the other half used an ultrasonic ranger to adjust the drive motor of the car. Both halves of the lab relied on using the PCA to adjust the pulsewidth used to control the motors. Lab 3 used only proportional control in the algorithm to adjust the motors.

The goal of lab 4 was to combine the steering and speed control elements of of both halves of lab 3 to drive the car along a track. This required integration of both SMBus device and the PCA to configure both steering and speed simultaneously. Lab 4 also combined these with A/D conversion to measure the car battery's voltage and gathers data to plot and analyze. Lab 4, similarly to lab 3, used only a proportional control algorithm for both servos.

After learning how to adapt the embedded control code for the ultrasonic ranger and the compass from Lab 4, Lab 5 focused on instead replacing both of these devices with a different sensor system. The accelerometer was used to determine the pulsewidth for the steering servo and drive motor based on the force of gravity being placed on the car, measured to its three cardinal axes. Using a closed-loop feedback system, we were able to successfully control the steering and speed of the car using the sensor information from the accelerometer while also monitoring the vehicle's battery voltage. Lab 5 also used only a proportional control algorithm for both servos.

With the Smart Car now fully functional, Lab 6 required us to adapt the similar components of the car to a Gondola. The gondola can be controlled using similar code to that of the prior labs, but this time we use derivative control along with proportional control to allow a smoother handling of the blimp. The main reason for this difference is a lack of natural damping within the blimp relative to the amount of damping within the car. The electronic compass in this case will first read the actual heading, compare it to the desired heading, and readjust the pulsewidth for all three steering fans based on the error calculation. The ultrasonic ranger will be used to adjust the desired heading that the gondola will steer towards.

Overview of Accelerometer and Gondola Feedback Control

Both the accelerometer and the gondola feedback control required an understanding of Proportional, Integral, and Derivative control (PID) and how each affects the movement of the vehicle. The control algorithm calculates the difference between a measured variable and a desired value as an error value. Each separate control term can be interpreted in terms of time. The proportional control depends on the present error; the integral control depends on the accumulation of past errors; and derivative control is a prediction of future errors based on the current rate of change of the error values. The PID values are then summed together to form a final error value used to adjust the pulsewidth. For our Lab 5 code we only used proportional control, while our Lab 6 code only used proportional and derivative control.

The accelerometer was used in lab 5 to fulfill the same role as both the ultrasonic ranger and electronic compass from labs 3 and 4. The acceleration from the accelerometer is measured in terms of hundredths of a meter per second squared based on the orientation of the accelerometer. Although the accelerometer is able to read the acceleration along all three of its axes, for this lab we only needed to read the value on the y-axis, oriented such that positive points towards the front of the car, and the x-axis, oriented such that positive points towards the left of the car.

The value of the acceleration was then multiplied by a specified gain constant, and then used in a proportional control algorithm. In this case, unlike the compass and ranger, the “desired” value for both axes used was 0, such that if the accelerometer was perfectly perpendicular to the direction of gravity, both servos would remain in their neutral position, keeping the car still. The yaw of the car was used to adjust the steering servo, such that the car would attempt to correct itself to steering uphill, and both the yaw and pitch were used to adjust the drive motor. This is to kickstart the car before it can start moving, so it can have a pitch necessary to drive uphill by the time it corrects the yaw. High gains for either constant would cause the car to have very jerky movement up the ramp, with large immediate adjustments. Low gains for either constant would leave the car stationary, as the motors would not have the necessary pulsewidths to provide the power to move up the ramp.

The gondola used a PD control algorithm based on the instantaneous and previous error to adjust the pulsewidth while keeping the pulsewidth within predefined boundaries. This algorithm was only used for steering control, and it used independent proportional and derivative gain constants. Because the gondola needs to rotate and uses two side fans, they needed to be adjusted to opposite ends of their neutral positions. This means that that one fan would have the adjustment added to neutral, and the other would have the adjustment subtracted from neutral. A high proportional gain will cause the gondola to rapidly spin out of control, constantly trying to adjust but in the end not finding a resting point. Low proportional gain will cause the vehicle to not move at all. Derivative gains create artificial damping, so the higher derivative gains will cause the gondola to spin quickly one time and then readjust itself. Low derivative gains will allow the gondola to gradually adjust itself rather than move in one jerky swing.

Hardware Component Description

Throughout the labs we utilized an assortment of hardware equipment to aid with our code in creating a more efficient vehicle, whether it was the Smart Car or the Gondola. Most devices that were used needed to work with the SMBus interface. The hardware for the motors to move the car were already available, and only needed to be connected to the C8051 microprocessor.

Servo Motor Controller

The drive motor control supplies a control signal that rotates the front wheels of the Smart Car to the left or right, depending on the pulsewidth signal going to the servo. It has modest power and voltage requirements, allowing it to run using a single buffer. The internal circuitry of the servo motor contains a potentiometer, an internal clock, and a comparator circuit that allow the motor to vary the pulsewidth of the signal.

Drive Motor Controller

Drive motors, most commonly used in applications requiring controlled rotary movement, require very little driving circuitry. They can be manipulated by modulating the pulsewidth of the driving signal. The drive servo motor allows the back wheels to move forward or backward, depending on the pulsewidth signal going to the motor.

Compass

The electronic compass uses the Earth's magnetic field to determine the direction of the vehicle being controlled. It then returns the current direction with respect to magnetic north. The wiring for the compass was simple, requiring only a voltage source, a ground, and the SDA/SCL lines to connect to the SMBus Interface.

Ranger

The ultrasonic ranger detects the distance between itself and an object or surface by creating short bursts of high frequency sound waves and reads when they bounce back to itself off of anything in its path. The wiring for the ranger was very similar to that of the compass in that it only required a voltage source, a ground, and the SDA/SCL lines which connect to the SMBus Interface.

LCD and Keypad

The keypad and LCD both provided an easy way for users to enter characters and display outputs rather than using the SecureCRT screen. The LCD is 4 lines long and 20 characters wide, while the keypad can read at most 12 button inputs at a time. The LCD keypad required only a ground, voltage source, and the SDA/SCL lines to connect to the SMBus interface. After wiring and testing the LCD keypad, our group decided to use the secureCRT terminal emulator to obtain user inputs, as it was easier to test as a user.

Accelerometer

The accelerometer allowed its vehicle to detect accelerations in all directions. When used with the Smart Car, it allowed the car to change its speed to overcome changes in height, such as hills. The accelerometer required only a ground, a voltage source, and the SDA/SCL lines to connect to the SMBus interface.

Serial Bus

The System Management Bus used for the labs was configured in a Master-Slave system, with the C8051 microprocessor as the master and various sensors used as slaves. This system was a two-wire system. The Serial Clock, or SCL, was used by the master to synchronize every device's input signals to the master. The Serial Data, or SDA, was used to transmit data between the master and slave devices. Every device on the bus had a unique identifying address, so that data going on the line would have an identifiable source.

The SMBus wiring required the SDA and SCL to be wired to each slave device before going to the EVB, with two pull-up resistors to pull the lines to a logic high state. Any device was capable of pulling the lines low to begin a data transfer in either direction. This is useful because it means that two-way communication can happen on the same lines without requiring any additional hardware beyond the initial wiring. Every device was wired in similar ways with respect to the SMBus.

RF Serial Link

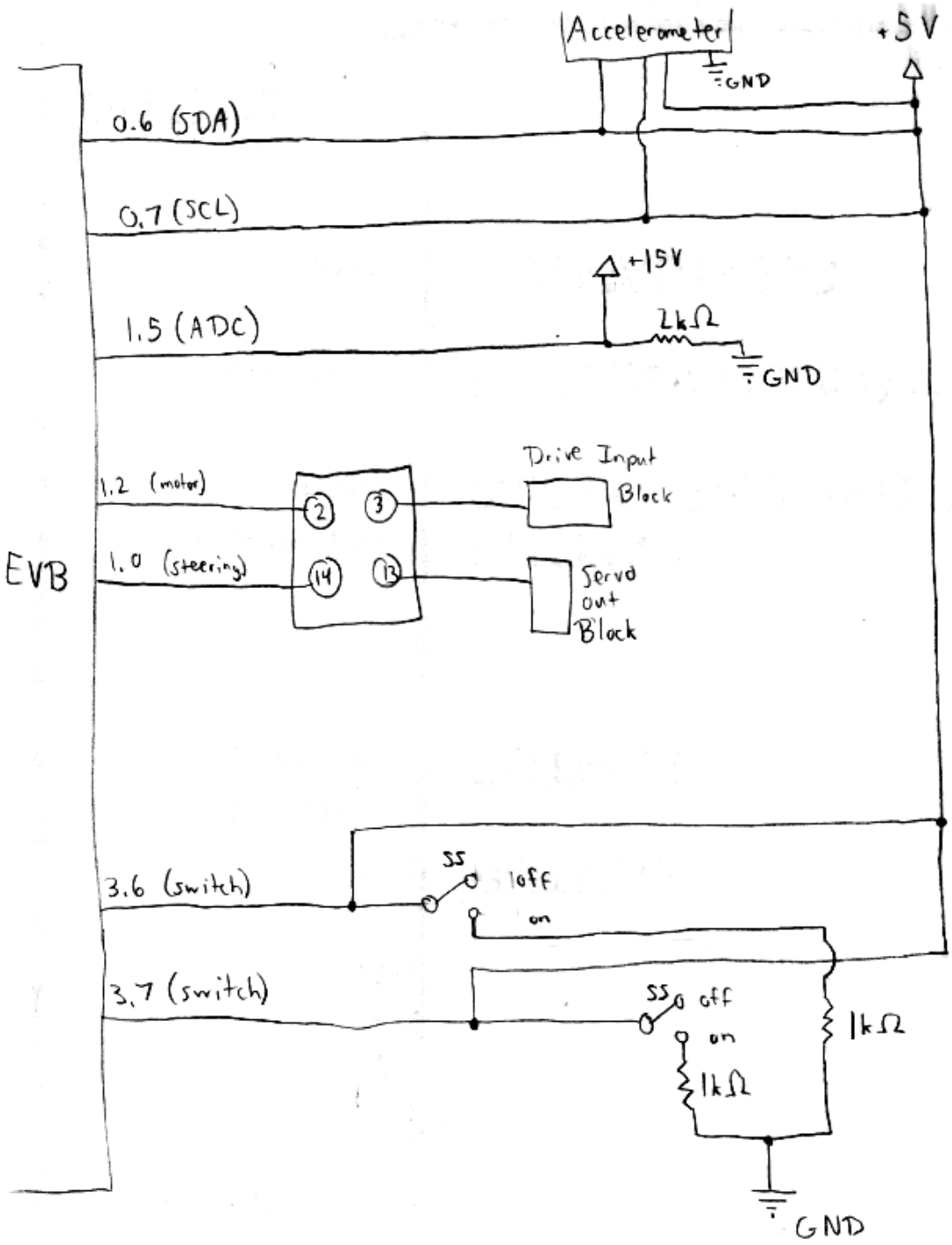
The RF Serial Link allowed for wireless communication between the gondola or Smart Car and the laptop without requiring a wired USB/RS-232 adapter. This made it easier to control the car or gondola while in motion without needing to follow its movement around with the cable. It also allowed for values to be received from the vehicle to record data plots over time much more easily.

Pin Connections from Priority Table

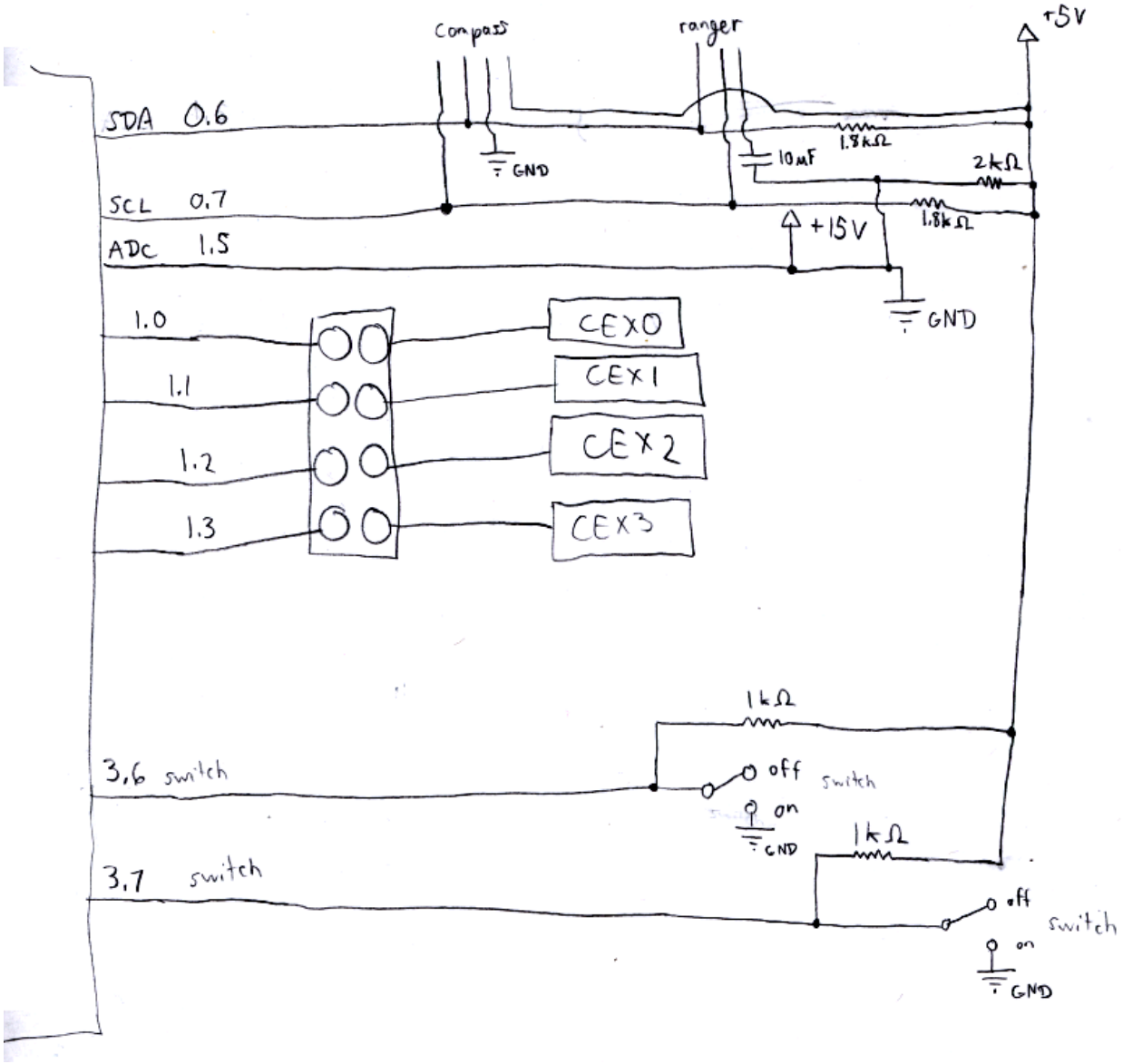
For each of these labs, the crossbar of the C8051 microprocessor was configured such that XBR0 = 0x27. This configuration enabled, in order of pin priority, UART0, SPI0, SMB0, CEX0, CEX1, CEX2, and CEX3. With this configuration, all of the CEX systems were configured to be on their respective pins on port 1 of the EVB, such that CEX0 is on 1.0, CEX1 is on 1.1, and so on.

Schematics

Wiring Diagram of the Lab 5 Car



Wiring Diagram of the Gondola



Software Description

Overview

The code for lab 4 was adapted from that of lab 3. We needed to combine the code from both halves of lab 3, speed and steering, so that we could use both the ultrasonic ranger and the electronic compass simultaneously. In this lab, the compass decided the direction based on the magnetic field of the Earth. The ranger determined the distance to an object in front of the car, and was used for collision detection. In addition, this was the first lab to require use of the wireless RF serial link and had to be wired as such to allow user input of a gain constant and a desired heading, to be used for the error calculation to aid with steering.

For Lab 5, we edited pieces of the code for Lab 4 to fit our needs; we revised functions used to alter the ranger and the compass so that they worked with the accelerometer alone. Code utilizing secureCRT was kept to allow ease of user entry for the gain constants. A proportional control equation was also used, where an error calculation was required to alter the PWM.

For Lab 6, the code from Lab 4 was adapted to be used with the gondola. The primary difference in code was adjusting the values for the speed and directions of the fans as opposed to the pulsewidths for the two sets of wheels on the Smart Car. A PD control equation was also used to determine the value for the error to be used to change the values for the fans.

Initialization - Ports, PCA, ADC, and XBR0

The port initializations only needed to configure the analog input pin and pins associated with the run/stop switches to open-drain, and the pins associated with the CEX modules to push-pull. The crossbar needed to be configured to enable UART0, SPI0, SMB0, and CEX0 - CEX3. The PCA needed to be configured such that the system would use SYSCLK/12, enable CF interrupts, and operate in 16-bit comparator mode. The A/D conversion was configured to use the 2.4 V internal reference voltage, enable ADC1, and to use a gain constant of 1 for conversion.

Uses of the PCA

The Programmable Counter Array (PCA) allowed for use of a timer function while requiring less CPU intervention. It contained a 16-bit counter/timer stored in two 8-bit registers, a mode register, a control register, and five capture/compare modules that contain an I/O line known as CEXn that can be routed through the crossbar. The PCA was used to control various motors on both the car and the gondola. The comparison modules compared their values to the present value of the PCA, and they switched to an 'on' state to send a pulse to the motors. This set a duty cycle that would cause them to function according to their individual specifications.

The PCA was adjusted largely using interrupts. Every time the PCA overflowed, it triggered an interrupt that set the PCA to a 20 ms period, and incremented variables used for the timing of other systems. These timing variables were used to call functions that adjusted the pulsewidth of each comparison module using control algorithms based on the nature of that specific module. Each comparison module was adjusted relative to a defined center-value; that would set the motor to its neutral position.

Pulse Streams

A pulse stream is a series of pulses sent to a motor, which would cause it to behave proportional to its duty cycle (Simsic, 2003). For our purposes, all elements that required pulsewidth modulation required a period of 20 ms, and a duty cycle between 5.5 and 9.5 percent. To stay within the mechanical limits of the motors, the duty cycle had to be maintained between these limits. The pulsewidths, and by extension the duty cycles, were modulated according to the motors' function.

Setting the Crossbar

The crossbar allocated and assigned pins on Ports 0 through 3 to the digital peripherals on the device, including UARTs, SMBus, PCA, and timers only if the corresponding enable bits of the peripheral are set to logic 1. Port pins, starting with P0.0 through P3.7, were assigned to digital peripherals in a priority order with UART0 having the highest priority and CNVSTR having the lowest. Crossbar configuration registers were stored as XBR0, XBR1, and XBR2.

I2C Read and Write

The I2C read and write functions for the labs were provided to us for each device used. The base functions were provided in the i2c.h header file. For every device used on the SMBus, the system needed to perform read operations to get data out of the devices. In the case of the ranger, a write operation was also necessary to convert this data into centimeters. Each device had a given address and registers that contained the necessary information, so the only necessary steps were to perform the read operations, and then combine the values of the registers into useful numbers for the control algorithms.

Use of I2C (SMB)

The SMBus interface was used to obtain data from various devices for use in the control algorithms throughout the labs. The configuration set the C8051 microprocessor as the master and the individual devices as the slaves. This allowed us to obtain the data from each device going only to the microprocessor. Each device communicated values specific to its function. For instance, the electronic compass had two registers which, when combined, would provide the direction the car was facing relative to magnetic north. By reading these values and passing them to the control algorithm, we were able to steer the car towards a given direction. Each lab consisted of using different slave devices for different functions, although the basic structure of using the SMBus remained the same between them.

Reading Analog Input, Scaling, and Results

The A/D converter examined the voltage on an analog input pin and determined about how far this voltage lay between a predefined voltage range, set for these labs as falling in between 0V and 2.4V. A voltage was read and then examined to calculate the percentage of how far along between the scale it is, i.e. 1.2V is 50% of the scale. The A/D result was then calculated by using the equation shown in Figure 1 where the gain constant was decided by the user or the code at the start of the program.

$$\text{A/D result} = floor\left[\frac{(V_{P1.x}) \cdot Gain}{V_{Ref}} \cdot 256\right] \quad \text{for} \quad 0V \le V_{P1.x} \le V_{Ref}$$

Figure 1: A/D Conversion Equation

Keypad Input and LCD Output

Code for reading the keypad and writing to the lcd were included in the i2c.h header file. The function lcd_clear() wiped the current output on the LCD while lcd_print() took in a string of character inputs and wrote them to various bits on the screen. The function read_keypad() took in a single character input from the keypad and included exceptions if the input character was a ‘0’, ‘9’. ‘#’, or ‘*’. When reading in data via the keypad, only one character could be read in at a time. When calculating two digit numbers, a simple equation was used to sum the second input with the first input multiplied by 10.

Feedback Control Loops

There are a number of different control algorithms that could have been used for each of the lab assignment. Examples of these control algorithms include Closed-Loop Control, Proportional Control, Proportional plus Integral Control (PI Control), and Proportional plus Derivative Control (PD Control).

For Lab 4, we decided to control the Smart Car using a closed-loop control algorithm. A closed-loop control system uses the concept of an open loop system but instead reads in output as current input to calculate based on feedback, as shown in Figure 2. They are designed to maintain the desired output condition by comparing it with the actual condition and generates an error signal by subtracting the two.

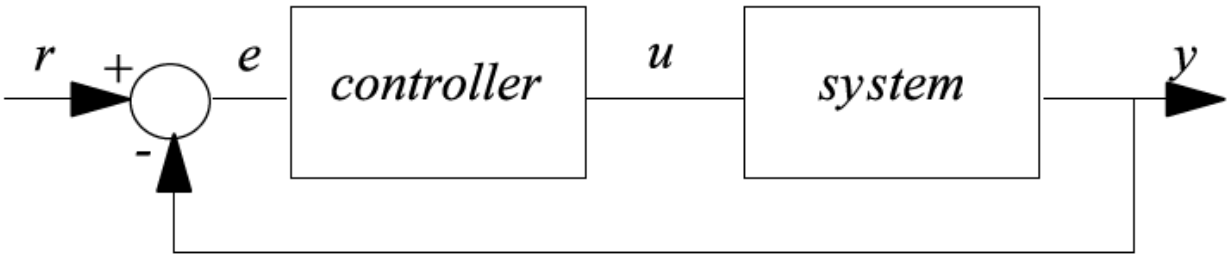


Figure 2: Simple closed-loop control system (Schoch)

For Lab 5, we decided to control the Smart Car using a proportional control algorithm. Proportional control is a type of closed-loop control algorithm where the difference between the desired output and the actual output is multiplied by a constant designated as K_p, and summed with the latest output setting, as shown in Figure 3 (Storr).

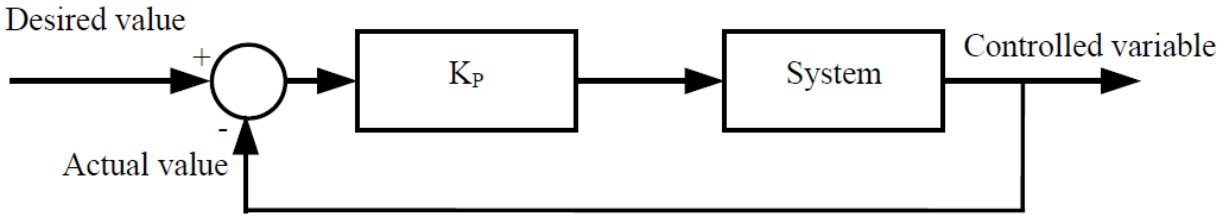


Figure 3: Proportional control block diagram (Schoch)

For Lab 6, we decided to control the gondola using a PD control algorithm. A PD control algorithm is similar to a proportional control algorithm but instead sums a third term comprised of the change between two previous errors multiplied by a different constant designated as K_d, as shown in Figure 4. PD control is especially useful for systems that require damping effects, such as the blimp.

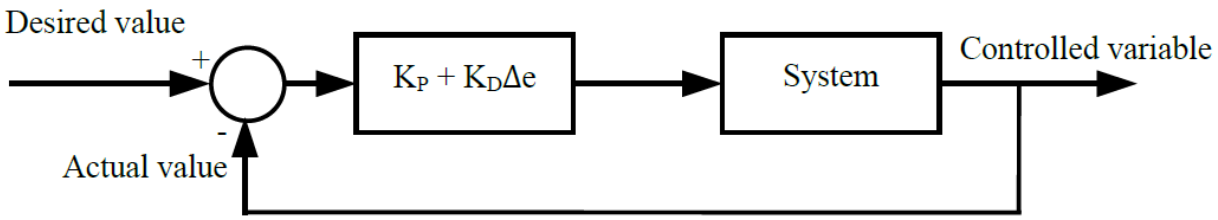


Figure 4: Proportional and derivative control block diagram (Schoch)

Timing

The timing for each lab was handled using the PCA interrupt. The configuration gave the interrupt a 20 ms period, with each interrupt incrementing variables used for timing other functions. For example, the compass required a minimum of 33 ms to reliably obtain a new reading. It was necessary to increment the timing variable associated with the compass a minimum of two times before that function could be called. The timing of when each function could be called needed to be adjusted above these predefined minimum values to prevent the SMBus from overloading itself and crashing. Additionally, extra variables were added to adjust the timing of when print statements would appear, so that data would be provided only after a meaningful amount of time had passed.

We determined that the necessary timings for each function through trial and error, until the SMBus was able to run without crashing. These timings are shown below in Table 1.

Function	Time Delay (ms)
Compass Reading	160
Ranger Reading	240
Accelerometer Reading	20
A/D Conversion	200

Table 1: Timing Delay for Functions

Results and Conclusions

Description of Accelerometer Performance

The accelerometer successfully adjusted the velocity of the car based on the angle of incline of the hill. To test this, we placed the Smart Car at the bottom of the hill and ran the program. It was able to climb up the hill and then stop at the top once it had reached a flat surface, or a zero degree incline. When the car started perpendicular to the incline, it was successfully able to turn itself either right or left so that it would face the top of the hill and repeat the process of climbing up and stopping.

However, the performance of the car was imperfect. Instead of the intended smooth driving, the car had jerky motion up the ramp in both cases. This is mostly because large gain constants were needed in order for the car to provide enough power to drive up the ramp in the first place. In addition, the accelerometer readings were erratic and exhibited a large amount of noise. This could have been solved by creating an initial offset at the beginning of the program.

Description of Gondola Performance

The performance of the gondola was reliant on the gain constants selected by the user. A high proportional gain would cause the gondola to spin out of control, never able to find the resting point. A low proportional gain would prevent the gondola from ever moving in the first place. A high derivative gain caused the gondola to have very jerky motion, making a few large swings. A low derivative gain caused the gondola to gradually adjust itself.

With moderate values for both gain constants, the gondola would quickly adjust itself to the resting point, and oscillate around it for several seconds. The derivative term in the control algorithm caused damped oscillation, until eventually the gondola rested in position. Due to natural variation in the compass and speed of the fans, there would always be a small amount of oscillation, but it was small enough that the control algorithm would not supply enough power to the fans to move the gondola far from the resting point.

Verification of Performance to Specifications

The goal for the Smart Car was to smoothly drive up the ramp from a variety of starting positions. Although the car was capable of making it up the ramp no matter the starting position, the path taken was jerky and the data was filled with noise. The Smart Car only technically accomplished the goals of the lab to specification, though it was not entirely as intended.

The goal for the gondola was to stabilize around the resting point within 15 seconds of running the code. When optimal gain constants were selected through trial and error, this objective was met. However, selecting extreme gain constants for either term in either direction would cause the gondola to never reach the desired resting point. It would either spin out of control or oscillate around a different point.

In general, most gain constants tested did not result in the intended behavior for the gondola. If either term of the control algorithm could overpower the other term, the gondola would be unable to reach the desired heading. A narrow band of suitable gain constants were used for the purpose of demonstration to meet the specifications of the lab.

Analysis of Results

Accelerometer Analysis

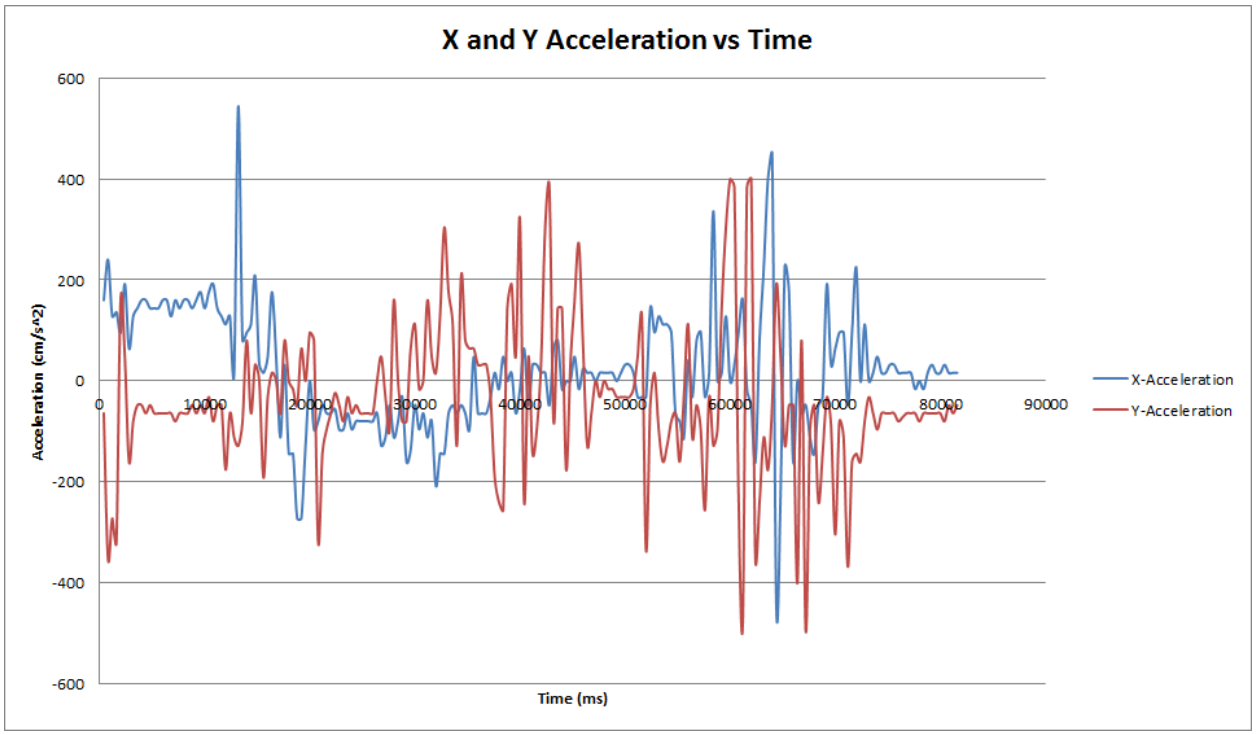


Figure 5: Accelerations along two axes over time

As shown in Figure 5, the X and Y accelerations of the car as it drove up the ramp were incredibly sensitive to variation and noise. Although some noise was controlled by averaging acceleration values over several readings, the end result was still very erratic for both acceleration axes.

The general trend in the data does show a general increase of Y acceleration as the car moves up the ramp, which is to be expected. The Y axis points towards the front of the car, and the car tilted upwards moving up the ramp, increasing the pitch and by extension the Y acceleration. Both accelerations decreased to near-zero by the end, which is also to be expected as the car is leveling out perpendicular to the direction of gravity. Much of the erratic behavior exhibited in this graph can also be explained by the jerky motion of the car, which is explained below.

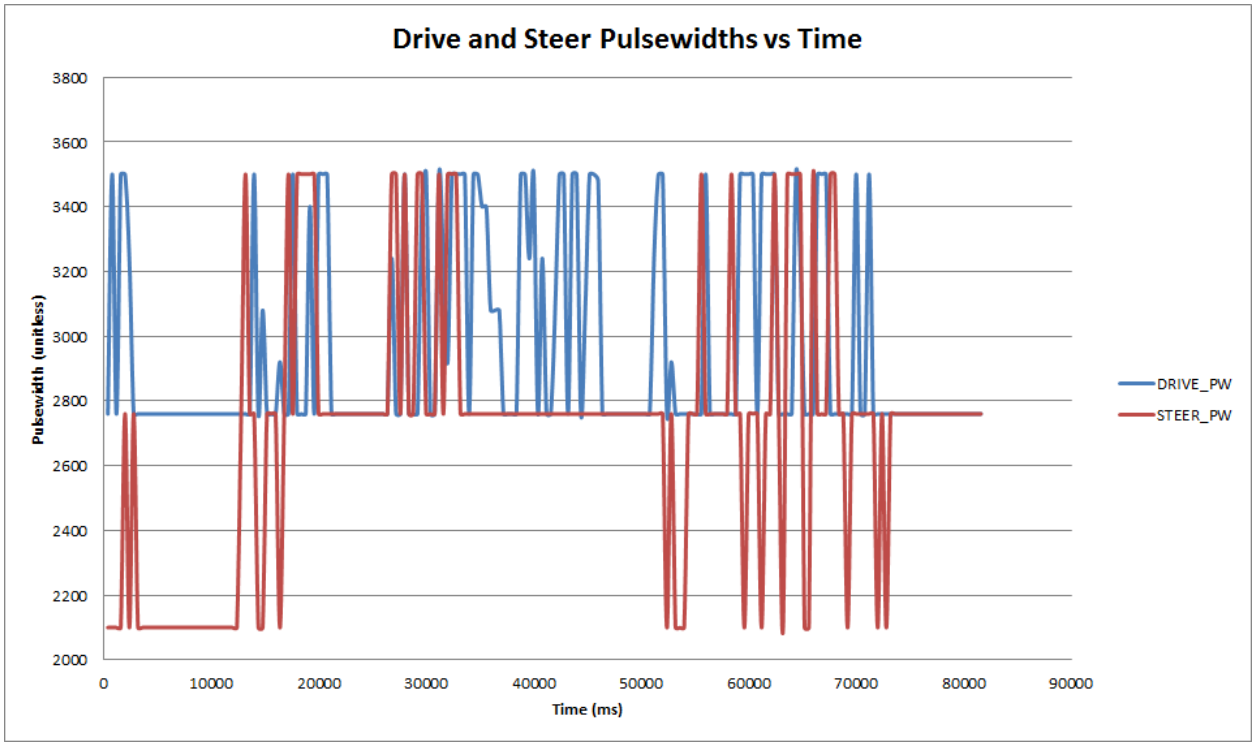


Figure 6: Motor pulsewidths over time

The drive and steer pulsewidths displayed in Figure 6 showed a very jerky behavior of the car, opposed to the intended smooth path. The two pulsewidths used mostly their maximum possible values. This is because we needed to use very large gain constants for both steering and driving. A value of 10 was selected for each system. This was because the Smart Car lacked the necessary power to move up the ramp at all unless the motors were pushed to their maximum values. However, this also resulted in incredibly jerky movement, and the car was only capable of inching up the ramp. This jerky motion is clearly shown in the graph oscillating between maximum values and the neutral position at a pulsewidth of 2760.

Gondola Analysis

Each figure in this section shows the behavior of the gondola on the turntable with various gain constants selected. These graphs are followed by an analysis of why this behavior was observed, and an explanation of any notable features of the graphs. For each graph, the blue line indicates the measured heading error of the gondola, and the red line indicates zero, which represents the desired heading of the gondola.

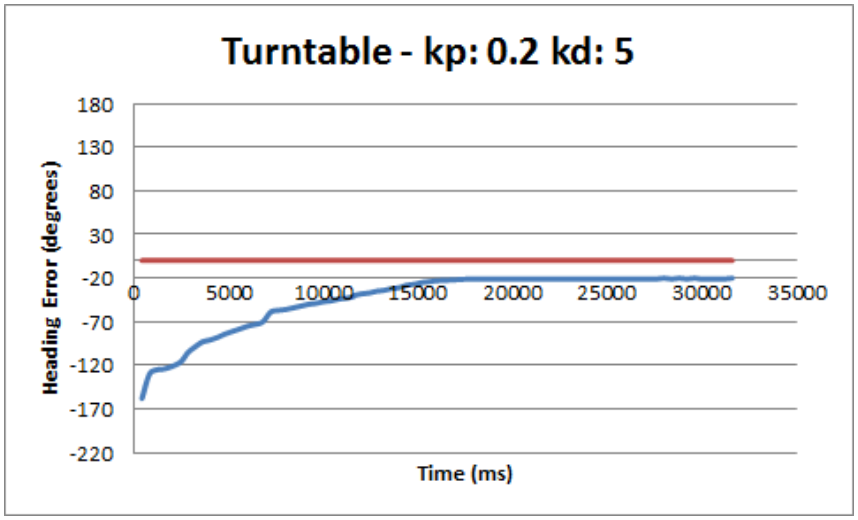


Figure 7: Low proportional gain constant; moderate derivative gain constant

The first trial, the graph for which is illustrated in Figure 7, used a low proportional gain constant and a moderate derivative gain constant. The gondola very slowly turned towards the neutral position but was unable to reach the designated resting point. This was largely because not enough power was provided to the fans to continue turning as the error decreased, and the damping from the derivative gain overpowered the power from the proportional gain constant.

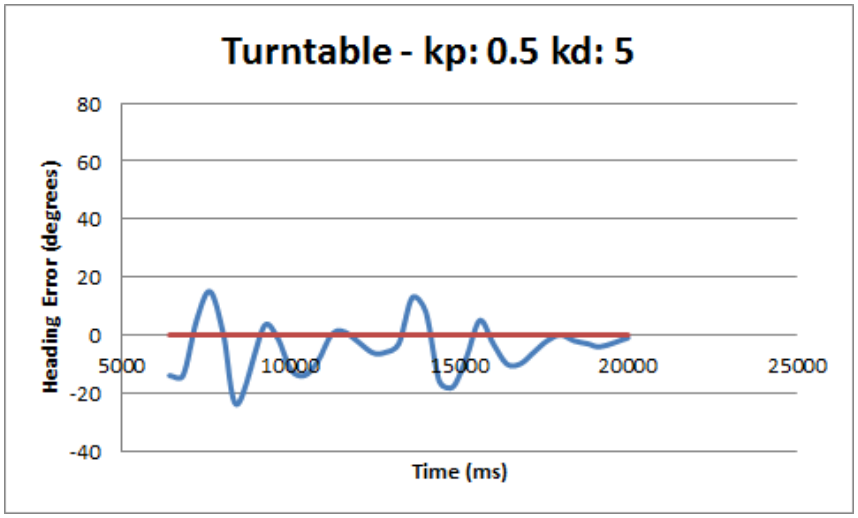


Figure 8: Low proportional gain constant; moderate derivative gain constant

The second trial, shown in Figure 8, used another low proportional gain constant and a moderate derivative gain constant. In this case, the gondola oscillated around the desired heading. The oscillations are irregular in shape, though they do exhibit damping. This irregularity was mainly due to the derivative term of the control algorithm providing large error corrections that temporarily overpower the proportional term.

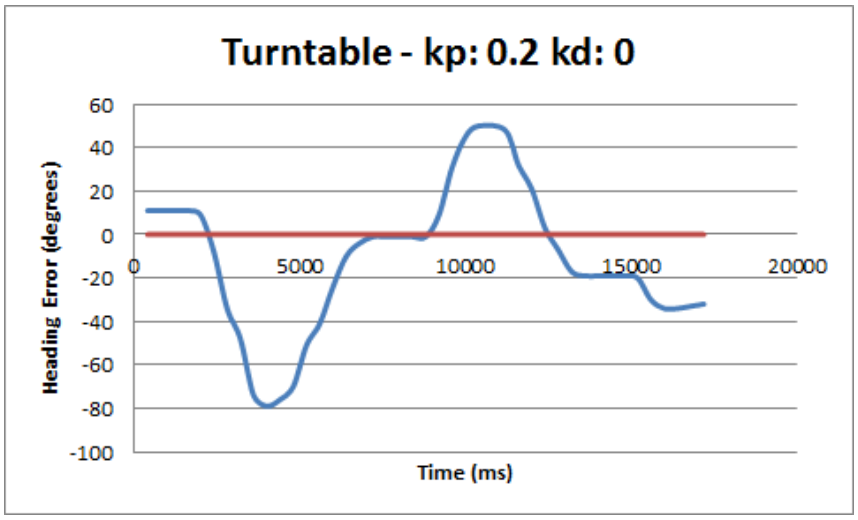


Figure 9: Low proportional gain constant; derivative gain constant of zero

The third trial, illustrated in Figure 9, used a low proportional gain constant and a derivative gain constant of zero. This provided almost no damping to the system, but also very little power to the fans to rotate the gondola. In this trial run, the gondola was pushed from the neutral position after stabilizing the first time, hence why it hovers at 0 error temporarily.

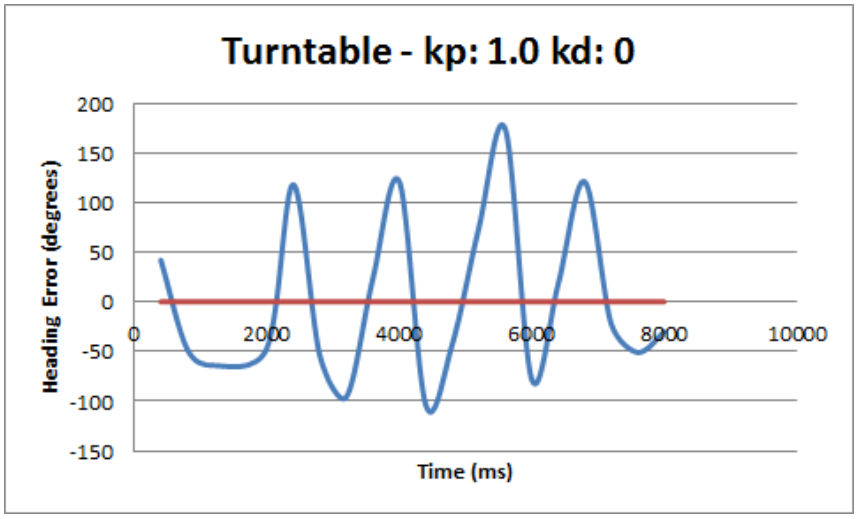


Figure 10: High proportional gain constant; derivative gain constant of zero

The fourth trial, shown in Figure 10, used a high proportional gain constant and a derivative gain constant of zero. This caused the gondola to spin rapidly out of control, oscillating only because the gondola spun in circles very quickly, never centering in on the resting point. The trial was prematurely ended due to the fact that the gondola was moving too fast and spinning out of control.

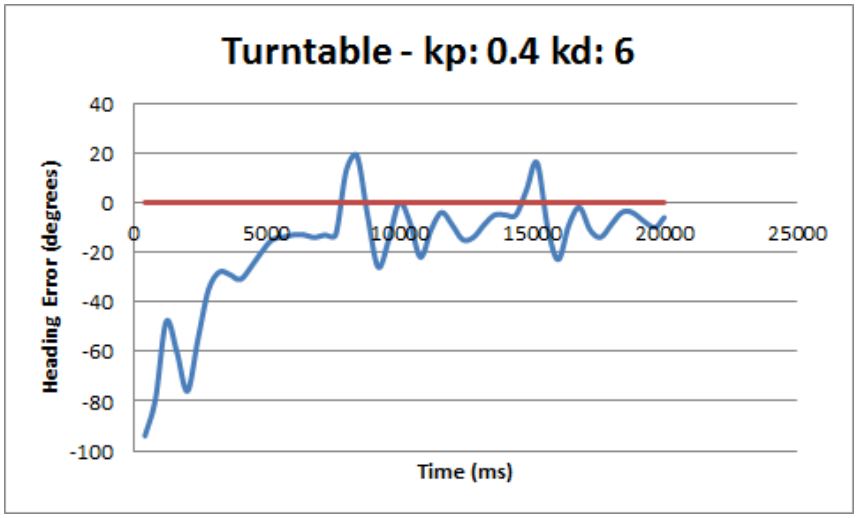


Figure 11: Low proportional gain constant; moderate derivative gain constant

The fifth trial, as seen in Figure 11, used a low proportional gain constant and a moderate derivative gain constant. This caused the gondola to rapidly adjust towards the resting point, and then oscillate randomly near it. The oscillations appear erratic and did not center on the resting point, primarily due to the proportional term of the control algorithm again being unable to adjust at small immediate error values.

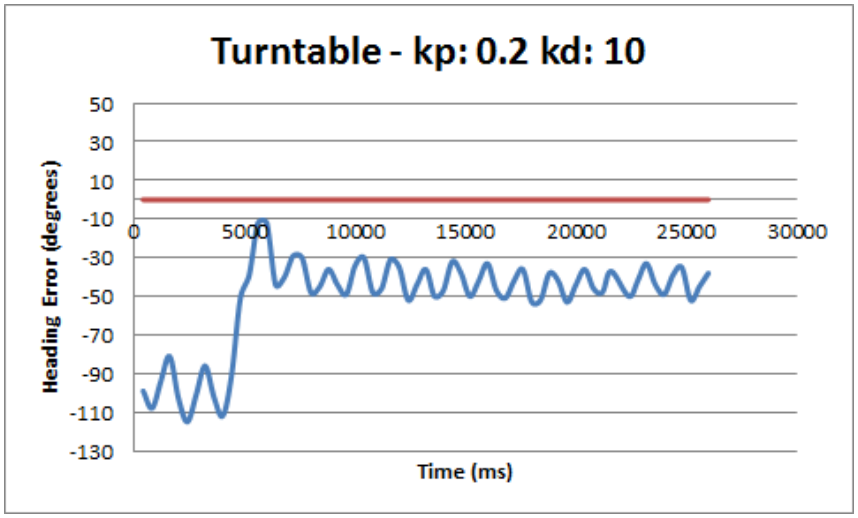


Figure 12: Low proportional gain constant; high derivative gain constant

The sixth trial, shown in Figure 12, used a low proportional gain constant and a high derivative gain constant. This caused a very small adjustment initially towards the resting point. After 4 seconds, the gondola was pushed closer to the resting position, resulting in the large change in heading error. From that point, the gondola was unable to reach the resting point, instead oscillating around a different heading. This is due to the large derivative term of the control algorithm completely overpowering the small proportional term, and therefore only adjusting based on the error differences, which were very small.

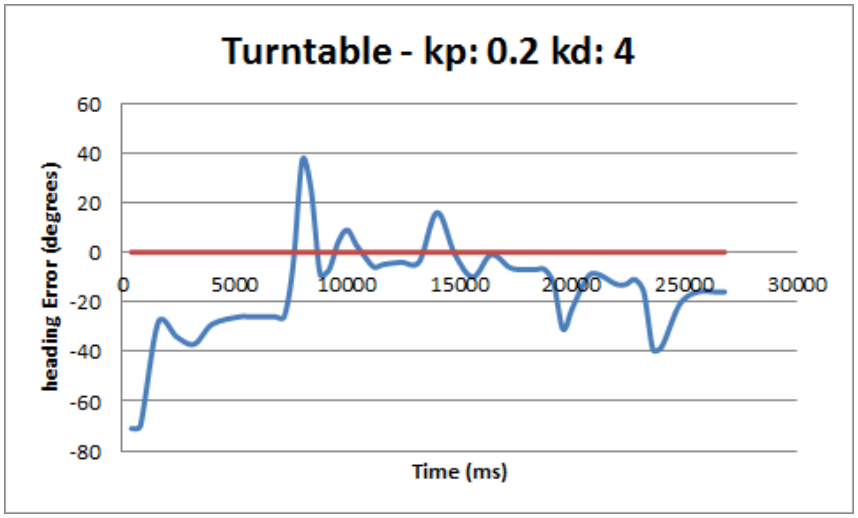


Figure 13: Low proportional gain constant; low derivative gain constant

The seventh trial, shown in Figure 13, used a low proportional gain constant and a low derivative gain constant. This caused the gondola to have erratic motion, with very sudden jerky motions based on rapid error differences. This was due to the two terms of the control algorithm fighting with each other and trading off which one overpowers the other. The physical effect of this was referred to as the gondola "looking over its own shoulder" where the otherwise smooth motion towards the resting point would be interrupted by a sudden twist.

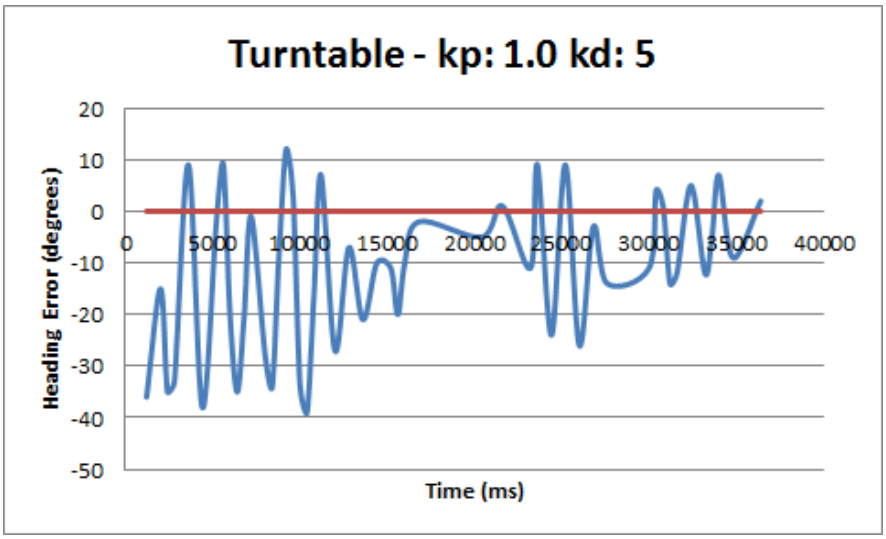


Figure 14: High proportional gain constant; moderate derivative gain constant

The eighth trial, illustrated in Figure 14, used a high proportional gain constant and a moderate derivative gain constant. This initially caused rapid adjustments about the resting point. The gondola was then held and moved to the resting point manually, then given a push. This is why the data exhibits a sudden departure from the pattern between 15 and 20 seconds. This was repeated between 25 and 30 seconds. When it was adjusted near the resting point manually, the gondola oscillated near the resting point, but moved too fast to establish it as a clear center.

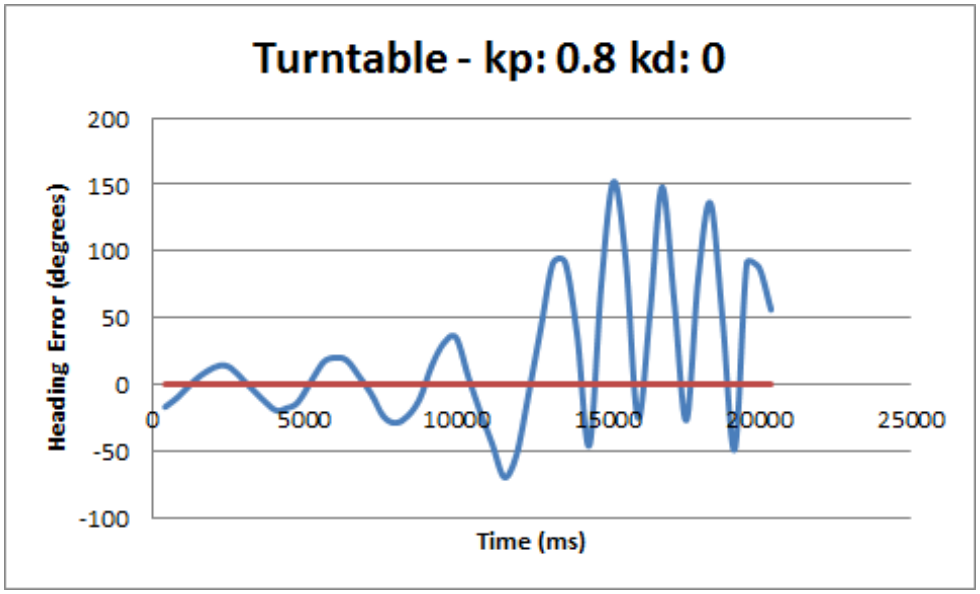


Figure 15: Moderate proportional gain constant; derivative gain constant of zero

The ninth trial, shown in Figure 15, used a moderate proportional gain constant and a derivative gain constant of zero. The gondola was also started very near to the resting point. The gondola exhibited very slow motion to begin, but quickly ramped up in speed of oscillation as it moved about the resting point. After 12 seconds, the resting point was no longer the center of the oscillations. This is due to the error becoming negative, and causing a large shift from the derivative gain constant. This behavior is responsible for the heading error quickly spiking up as soon as it goes negative, preventing the resting point from being the center of oscillations.

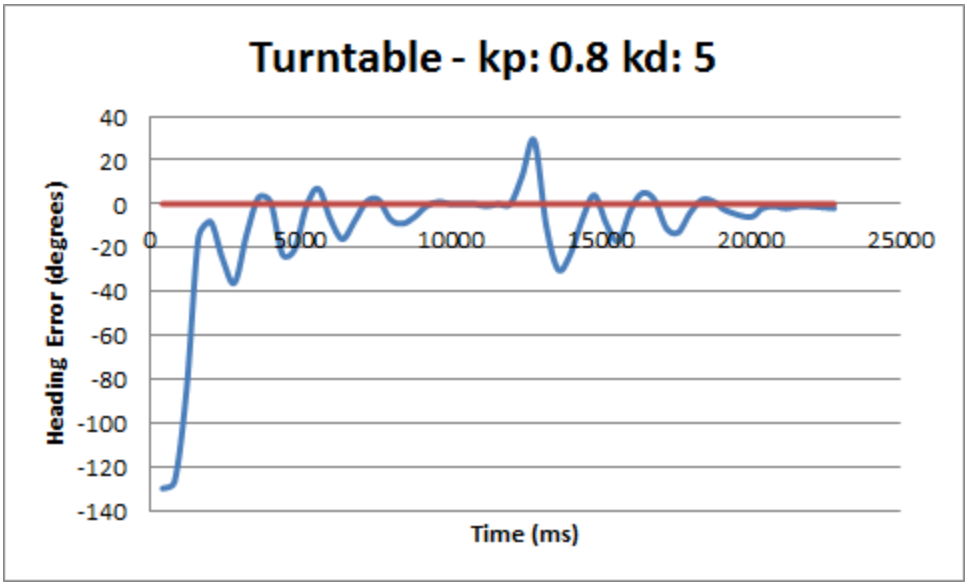


Figure 16: Moderate proportional gain constant; moderate derivative gain constant

The tenth trial, shown in Figure 16, used a moderate proportional gain constant and a moderate derivative gain constant. This trial also included the function of the ultrasonic ranger to adjust the desired heading from the initial value. For a constant desired heading, the gondola quickly adjusted itself to the resting point and successfully stopped there. The desired heading was then adjusted at 12 seconds, causing a sudden spike in heading error. The gondola again quickly adjusted itself to the new desired heading and stabilized. The gain constants used for this trial were determined to be the optimal gain constants for the intended behavior of the gondola.

Lessons Learned

During the first part of the semester, the team fell into a routine where one person was in charge of hardware and wiring, and two people were in charge or software, coding, and debugging. It was simple and efficient for the first three labs, but when upon reaching lab four, this strategy was no longer effective. Having each person focused on his or her own subsystem became a weakness rather than a strength when people no longer knew what was happening in the other subsystems, which was detrimental to learning the material. We learned that it is important in any group for each member to be well-informed on each subsystem, even if they are not actively working on it themselves.

Problems Encountered and Solution

The problems that this group encountered during this lab were relatively simple to fix. We felt that we were short on time during class, so we often found ourselves attending lab hours for extra time. Our other main problem was frustration with code that seemed as though it should work but, for some reason or another, it did not. Our solution to this was a lot of "rubber duck" debugging and discussing the code out loud in order to find our mistakes. If we could not find our errors, we took advantage of our resources and asked a TA or the professor for help. The largest problem encountered was in the timing of each system. We found that we were overloading the SMBus, resulting in junk values being returned. This held us up for two weeks before the issue was able to be resolved and we could move forward.

References

Simsic, Lara. "The C Programmers Guide to Verilog." *Embedded*. N.p., 9 July 2003. Web. 09 May 2015.

P. Schoch, A. Gutin, S. Lee, C. Sankar, LITEC Lab Manual - version 14.5, 2015

Storr, Wayne. "Closed-loop System and Closed-loop Control Systems." *Basic Electronics Tutorials.* 30 Aug. 2013. Web. 10 May 2015.

Division of Labor

All three members participated in the documentation of the lab. Seth and Kathryn focused on writing the C code to work with the Smart Car and the Gondola, Rebecca assisted in hardware and diagramming the system. This report was drafted by Rebecca and Kathryn, and it was revised by all three partners.

Kathryn DiPippo Rebecca Halzack Seth Rutman

Pseudocode, and Code

Lab 5 Pseudocode

```
#include <stdio.h>
#include <stdlib.h>
#include <c8051_SDCC.h> // Include files. This file is available online in LMS
#include <i2c.h>        // Get from LMS, THIS MUST BE INCLUDED AFTER stdio.h
#define PCA_START 28672 // 28672 for exactly 20ms
#define MAX_RANGE 55
#define PW_CENTER 2760
#define PW_MAX 3500
#define PW_MIN 2030

//-----------------------------------------------------------------------------
// Function Prototypes
// ....

// Global variables
// ....

//=============================================================================
main function
      Initialize the sys, port, interrupt, PCA, ADC, SMBus, and accelerometer
    Set the PCA to start in a center position

    Wait a long time (1s) for motors to initialize

      Read compass gain as input from the user in the SecureCRT command window
      Read drive gain as input from the user in the SecureCRT command window

      while the car is still moving
            printf("\n\r------------DATA COLLECTION------------\n");
            printf("\n\rX-Accel            |     Y-Accel           |
STEER_PW    |     DRIVE_PW\n\r");
      end while

      while true
            while both of the switches are off
                  if the accelerometer is ready to be read
                        reset the accelerometer flag
                        read the accelerometer
                        set the pulsewidth accordingly (refer to set_PW()
below)
                  end if statement

                  if the analog to digial converter needs to be read
                        reset the A/D flag
```

```
                        Read analog input on pin 1.5
                        Convert result back to input voltage
                  end if statement

                  Output the results for transfer into excel every 20 cycles
          end while loop

            if either of the switches is turned on
                  Stop the car
                  Reread steering gain as inputed by the user
                  Reread drive gain as inputed by the user
                  Pick_Steering_Gain();
                  Pick_Drive_Gain();
            end if statement
      end while loop
end of main function
```

Lab 6 Pseudocode

```
compiler directives
      ...
declare global variables
signed int left_pw;
signed int right_pw;
signed int Error = 0;
signed int prev_error = 0;
unsigned char Counts, nCounts;
unsigned char new_range = 0;
unsigned char new_AD = 0;
unsigned char new_heading;
unsigned char h_count;
unsigned char r_count = 0;
unsigned char adc_count = 0;
unsigned int DRIVE_PW = 2760;
unsigned int STEER_PW = 2760;
unsigned char Data[2]; // Data is an array with a length of 2
unsigned char print_delay = 0;
unsigned char AD_Result = 0;
unsigned char voltage = 0;
unsigned int heading = 0;
unsigned int range = 0;
unsigned int desired_heading = 0;
float heading_kp = 0;
unsigned char heading_kd = 0;
int print_error = 0;
signed int init_heading = 0;
unsigned int time = 0;
```

```
function prototypes
void Port_Init(void);
void PCA_Init(void);
void SMB0_Init(void);
void ADC_Init(void);
void Interrupt_Init(void);
void accelerometer_adjustment(void);
void PCA_ISR(void) __interrupt 9;
void set_PW(void);
unsigned char read_AD_input(unsigned char n);
void read_accel(void);
void Pick_Heading(void);
int read_ranger(void);
void Set_Desired_Heading(void);
int read_compass(void);
void Pick_Heading_kp(void);
void Pick_Heading_kd(void);
void Set_Fan_Angle(void);

//-------------------------------------------------------------------------------
-
main function
initialize system
      initialize ports
      initialize PCA board
      initialize the SMBus
      initialize interrupts
      initialize ADC
      Set initial pulsewidths to neutral positions
      Set fan angle
      Obtain desired heading
      Obtain proportional gain
      Obtain derivative gain
      while(1)
            if(new_range)
                  read the ranger
                  adjust the desired heading
            if(new_heading)
                  read the compass
                  adjust the pulsewidths
            if(new_AD)
                  perform ADC
                  calculate battery voltage
            if(print_delay == 20) //so 400 ms has passed
                  print relevant information
      end while
end main
```

```
//-------------------------------------------------------------------------
-
//functions

Port_Init
      set output pin for CEX0, CEX1, CEX2, CEX3 in push-pull mode
      set input pin for 3.6,7 to open-drain
      set input pin for 3.6,7 to high impedence
      set XBR0 to enable CEX0, 1, 2, 3, UART0EN, SPI0EN, SMB0EN
      set ADC pins

Interrupt_Init
      Enable global interrupts
      Enable PCA interrupts

PCA_Init
      SYSCLK/12, enable CF interrupts, suspend when idle
      Set CPA0CPM0 and CPA0CPM2 to 16-bit comparator mode
      Enable PCA

SMB0_Init
      Set SCL to 100 kHz
      Enable SMBUS0

PCA_ISR __interrupt 9
      if(CF)
            increment print_delay, r_count, h_count, adc_count
            set new_range, new_heading, new_AD when appropriate
            increment overflows and seconds
ADC_Init
      set to use Vref
      enable ADC1
      set AD gain to 1

AD_Result
      set appropriate pin for ADC
      clear conversion complete flag
      initiate A/D conversion
      wait for conversion complete
      return

Pick_Heading
      increment desired heading using secureCRT
      loop heading to avoid impossible angles

read_compass
      set addr as the address of the sensor, 0xC0 for the compass
      initialize Data as an array with a length of 2
```

```
      initialize read_heading as the heading returned in degrees between 0 and
3599
      read two byte, starting at reg 2
      combine the two values and store in read_heading
      return read_heading

read_ranger
      set the address of the sensor, 0xC0 for the compass
      initialize Data as an array with a length of 2
      set the range to be equal to 0 as default
      read two byte, starting at reg 2, using i2c header file
      set range to be the combination of the two values
      write 0x51 to reg 0 of the ranger:
      write one byte of data to reg 0 at addr
      return range

set_PW
      Error = desired_heading - heading
      adjust error to stay within -1800 to 1800
      calculate PW based on proportional and derivative gains
      calculate left_pw and right_pw to opposite sides of neutral
      keep servos within mechanical constraints
      adjust pulsewidths

Set_Desired_Heading
      read ranger
      calculate new desired heading based on range
      adjust new desired heading to stay within real values

Pick_Heading_kp
      increment desired heading kp using secureCRT
      bound it to useful values

Pick_Heading_kd
      increment desired heading kd using secureCRT
      bound it to useful values

Set_Fan_Angle
      increment angle pulsewidth using secureCRT
      adjust pulsewidth with each increment
      end when desired orientation is reached
```

Lab 5 Code

```
Names: Kathryn DiPippo, Rebecca Halzack, Seth Rutman
Section: 4B
Date: 4/7/2015
File name: Lab 5
```

```
#include <stdio.h>
#include <stdlib.h>
#include <c8051_SDCC.h> // Include files. This file is available online in LMS
#include <i2c.h>         // Get from LMS, THIS MUST BE INCLUDED AFTER stdio.h
#define PCA_START 28672 // 28672 for exactly 20ms
#define MAX_RANGE 55
#define PW_CENTER 2760
#define PW_MAX 3500
#define PW_MIN 2030

//-----------------------------------------------------------------------------
// Function Prototypes
//-----------------------------------------------------------------------------
void Port_Init(void);   // Initialize ports for input and output
void PCA_Init(void);
void SMB0_Init(void);
void ADC_Init(void);
void Interrupt_Init(void);
void accelerometer_adjustment(void);
void PCA_ISR(void) __interrupt 9;
void set_PW(void);
unsigned char read_AD_input(unsigned char n);
void read_accel(void);
void Pick_Steering_Gain(void);
void Pick_Drive_Gain(void);

// Global variables
signed int avg_gx = 0;
signed int avg_gy = 0;
unsigned int Counts, nCounts;
unsigned char a_count = 0;
unsigned char adc_count = 0;
unsigned char delay = 0;
unsigned char new_accel = 0;
unsigned int DRIVE_PW = 2760;
unsigned int STEER_PW = 2760;
__sbit __at 0xB7 COMPASS_SWITCH;
__sbit __at 0xB6 RANGER_SWITCH;
unsigned char AD_Result = 0;
unsigned char voltage = 0;
//unsigned char Data[2]; // Data is an array with a length of 2
unsigned char print_delay = 0;
signed int gx = 0;
signed int gy = 0;
signed int gx_adj = 0;
signed int gy_adj = 0;
signed int gx_motor_adj = 0;
float steer_gain = 0;
float drive_gain = 0;
unsigned char new_AD = 0;

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

```
//-----------------------------------------------------------------------------
// Main Function
void main(void)
{
      Sys_Init(); // System Initialization - MUST BE 1st EXECUTABLE STATEMENT
      Port_Init();
      Interrupt_Init();
      PCA_Init();
      ADC_Init();
      SMB0_Init();
      Accel_Init();
      putchar('\r');  // Dummy write to serial port
      printf("\nStart\r\n");
      PCA0CP0 = 0xFFFF - PW_CENTER;
      PCA0CP2 = 0xFFFF - PW_CENTER; //Car isn't moving to start
      Counts = 0;
      while (Counts < 1);  // Wait a long time (1s) for motors to initialize
      Pick_Steering_Gain();
      Pick_Drive_Gain();
      printf("\rThe car will move quickly at first to move up the ramp\n");
      Counts = 0;
      nCounts = 0;
      while(Counts <=2) PCA0CP2 = 0xFFFF - 3500;
      printf("\n\r------------DATA COLLECTION------------\n");
      printf("\n\rX-Accel            |     Y-Accel            |     STEER_PW
|     DRIVE_PW\n\r");
      while (1)
      {
            while(!RANGER_SWITCH && !COMPASS_SWITCH)  //These two switches act
as run/stop switches
            {
                  if(new_accel)     //If the accelerometer is ready to be read
                  {
                        new_accel = 0;
                        read_accel();
                        set_PW();
                  }
                  if(new_AD)
                  {
                        new_AD = 0;
                        AD_Result = read_AD_input(7); //Read analog input on
pin 1.5
                        voltage = ((12.8/255)*(AD_Result)); //Convert back to
input voltage
                  }
                  if(print_delay == 20)
                  {
                        printf("\r%d             |     %d          |     %d
      |     %d\n", gx, gy, STEER_PW, DRIVE_PW);

                        print_delay = 0;

                  }
```

```
                  // Output the results for transfer into excel
          }
            if(RANGER_SWITCH || COMPASS_SWITCH)
            {
                  PCA0CP0 = 0xFFFF - 2760;
                  PCA0CP2 = 0xFFFF - 2760;
                  Pick_Steering_Gain();
                  Pick_Drive_Gain();
            }
}
}
//***************************************************************************
//---------------------------------------------------------------------------
// Set up ports for input and output
void Port_Init(void)
{
      XBR0 = 0x27;
      P1MDIN      &= 0x7F;     // set pin 1.5 for analog input
      P1MDOUT |= 0x05;  //set output pin for CEX0/2 in push-pull mode
      P1MDOUT &= 0x7F;  // set input pin for 1.5 to open-drain
      P1          |= ~0x7F;   // set input pin for 1.5 to high impedence
      P3MDOUT &= 0x7F;  // set input pin for 3.6/7 to open-drain
      P3          |= ~0x7F;   // set input pin for 3.6/7 to high impedence

}

//---------------------------------------------------------------------------
// Set up interrupts
void Interrupt_Init(void)
{
      IE |= 0x02;
      EIE1 |= 0x08;
      EA = 1;
}

//---------------------------------------------------------------------------
// Set up Programmable Counter Array
void PCA_Init(void)
{
      PCA0MD = 0x81;     // SYSCLK/12, enable CF interrupts, suspend when idle
      PCA0CPM0 = 0xC2;   // 16 bit, enable compare, enable PWM; NOT USED HERE
      PCA0CPM2 = 0xC2;
      PCA0CN = 0x40;     // enable PCA
}

//---------------------------------------------------------------------------
// Set up the SMB
void SMB0_Init(void)     // This was at the top, moved it here to call wait()
{
      SMB0CR = 0x93;     // Set SCL to 100KHz
      ENSMB = 1;         // Enable SMBUS0
}

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

```
// PCA_ISR: Interrupt Service Routine for Programmable Counter Array Overflow
Interrupt
void PCA_ISR(void) __interrupt 9
{
      if (CF)
      {
      CF = 0;                  // clear the interrupt flag
          nCounts++;                              // Counts overflows for
initial delay
      PCA0 = PCA_START;
      if (nCounts > 50)              //Initial one second delay
      {
            //nCounts = 0;
            Counts++;                // seconds counter
      }
            print_delay++;                        // delay for print statements
            a_count++;
            if (a_count>=1)
            {
                  a_count = 0;
                  new_accel = 1;
            }
            adc_count++;
            if(adc_count >=10)
            {
                  adc_count = 0;
                  new_AD = 1;
            }
      }
      else PCA0CN &= 0xC0;           // clear all other 9-type interrupts
}

//-----------------------------------------------------------------------------
--
// Analog/Digital Conversion Initialization

void ADC_Init(void)
{
      REF0CN = 0x03; // Set Vref to use internal reference voltage (2.4 V)
      ADC1CN = 0x80; // Enable A/D converter (ADC1)
      ADC1CF |= 0x01; // Set A/D converter gain to 1
}

//-----------------------------------------------------------------------------
--
// Analog/Digital Conversion Function
unsigned char read_AD_input(unsigned char n)
{
      AMX1SL = n; // Set P1.n as the analog input for ADC1
      ADC1CN = ADC1CN & ~0x20; // Clear the "Conversion Completed" flag
      ADC1CN = ADC1CN | 0x10; // Initiate A/D conversion

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

```
	return ADC1; // Return digital value in ADC1 register
}

//-------------------------------------------------------------------------
//Selecting the steering gain function
void Pick_Steering_Gain(void)
{
	char input;
	printf("\rPlease select a desired steering gain.\n");
	printf("\r'u' will increment by 0.1. 'd' will decrement by 0.1.\n");
	printf("\r'f' when finished\n");
	while(1)
	{
		input = getchar();
		if(input == 'u') steer_gain += 0.1;
		if(input == 'd') steer_gain -= 0.1;
		if(input == 'f') return;
		if(steer_gain >= 10) steer_gain = 10;
		if(steer_gain <= 0) steer_gain = 0;
		printf_fast_f("\rDesired steering gain: %2.1f\n", steer_gain);
	}
}
//-------------------------------------------------------------------------
//Selecting the drive gain function
void Pick_Drive_Gain(void)
{
	char input;
	printf("\rPlease select a desired drive gain.\n");
	printf("\r'u' will increment by 0.1. 'd' will decrement by 0.1.\n");
	printf("\r'f' when finished\n");
	while(1)
	{
		input = getchar();
		if(input == 'u') drive_gain += 0.1;
		if(input == 'd') drive_gain -= 0.1;
		if(input == 'f') return;
		if(drive_gain >= 10) drive_gain = 10;
		if(drive_gain <= 0) drive_gain = 0;
		printf_fast_f("\rDesired drive gain: %2.1f\n", drive_gain);
	}
}

//-------------------------------------------------------------------------
//Adjusting the motor speed
void set_PW(void)
{
	accelerometer_adjustment();
	STEER_PW = 2760 - (gx_adj);
	//Stay within limits of the servo
	//Depending on the car, these numbers may need to be determined using Lab
3-1 - Steering
	if(STEER_PW < 2100)
	{
		STEER_PW = 2100;
```

```
	}
	if(STEER_PW > 3500)
	{
		STEER_PW = 3500;
	}
	DRIVE_PW = 2760 + (gy_adj) + (gx_motor_adj);
	if(DRIVE_PW < 2760) DRIVE_PW = 2760;
	if(DRIVE_PW > 3500) DRIVE_PW = 3500;
	PCA0CP0 = 0xFFFF - STEER_PW; // Change pulse width
	PCA0CP2 = 0xFFFF - DRIVE_PW;
}

//
=============================================================================
// Revise the C code used in Lab 4. Write a  that first calls the
// accelerometer initialization routine and then calls a refunctionad_accel()
function
// and sets the PWM for the steering servo based on the side-to-side tilt of
the
// car so that is turns in the direction of the upward slope. Both the
// side-to-side tilt and the front-to-back tilt will be used to determine a PWM
// for the drive motor. The main code will also need to average 4 to 8 samples
// from the accelerometer each time, since there is noise on the signal.
void accelerometer_adjustment(void)
{
	if((gx > -100) && (gx < 100)) gx_adj = 0;
	else gx_adj = (int)((steer_gain)*(gx));
	gy_adj = (int)((drive_gain)*(gy));
	if((gx > -100) && (gx < 100))
	{
		gx_motor_adj = 0;
	}
	else gx_motor_adj = abs((int)((drive_gain)*(gx)));
}
// returns 1 if the accelerometer is ready to be read
unsigned char status_reg_a(void)
{
	unsigned char Data[2];
	unsigned char addr = 0x30; // the address of the sensor, 0x30 for the
accelerometer
	i2c_read_data(addr, 0x27, Data, 2); // read two byte, starting at reg
0x27
	if (Data[0] && Data[1])
	{
		return 1;
	}
	else
	{
		return 0;
	}
}

void read_accel(void)
{
```

```
      //Note that the accelerometer gives values in hundredths of g. i.e. 980 =
g
      signed char Data[4];
      unsigned char addr = 0x30;
      signed int x_value;
      signed int y_value;
      int i;
      unsigned char j = 0;
      //Wait one 20ms cycle to avoid hitting the SMB too frequently and locking
it up - included in PCA_ISR
      new_accel = 0;
      j = 0;
      //Clear the averages
      avg_gx = 0;
      avg_gy = 0;
      for (i=0; i<8; i++) //For 4 iterations (or maybe 8)
      {
            //Read status_reg_a into Data[0] (register 0x27, status_reg_a,
indicates when data is ready)
            //Make sure the 2 LSbits are high: (Data[0] & 0x03) == 0x03
            if(status_reg_a())
            {
                  //Read 4 registers starting with 0x28. NOTE: this SMB device
follows a modified protocol. To
                  //read multiple registers the MSbit of the first register
value must be high:
                  i2c_read_data(addr, (0x28|0x80), Data, 4); //assert MSB to
read mult. Bytes
                        //Discard the low byte, and extend the high byte sign
to form a 16-bit acceleration
                        //value and then shift value to the low 12 bits of the
16-bit integer.
                        //Accumulate sum for averaging.
                  x_value = ((Data[1] << 8)>>4);
                  y_value = ((Data[3] << 8)>>4);
                  //These lines convert the value from the registers into a
12-bit integer
                  avg_gx += x_value; //a simple >>4 WILL NOT WORK;
                  avg_gy += y_value; //it will not set the sign bit correctly
                  j++;
            }
      }
            //Set global variables and remove constant offset, if known.
            if(j > 0) //This averages based on how many values we actually
measured
            {
                  gx = (avg_gx)/(j); //(or = avg_gx - x0 if nominal gx offset
is known)
                  gy = (avg_gy)/(j); //(or = avg_gy - y0 if nominal gy offset
is known)
            }
}
```

```
//
================================================================================
// Continue to use routines to output parameter and settings on both the LCD
// display and SecureCRT terminal to aid in troubleshooting.

//
================================================================================
// As in Lab 4, there will be parameters to adjust affecting the behavior of
// the system. This time there are 3 different adjustable proportional gain
// feedback constants that must be optimized to give the car the best
performance.
// Write C code to allow user entry of gain constants using the keypad or
terminal.
```

Lab 6 Code

```
Names: Kathryn DiPippo, Rebecca Halzack, Seth Rutman
Section: 4B
Date: 5/1/2015
File name: Lab 6

#include <stdio.h>
#include <stdlib.h>
#include <c8051_SDCC.h>
#include <i2c.h>
#define PCA_START 28672
#define PW_CENTER 2765
#define PW_MAX 3500
#define PW_MIN 2030

//-------------------------------------------------------------------------------
// Function Prototypes
//-------------------------------------------------------------------------------
void Port_Init(void);   // Initialize ports for input and output
void PCA_Init(void);
void SMB0_Init(void);
void ADC_Init(void);
void Interrupt_Init(void);
void accelerometer_adjustment(void);
void PCA_ISR(void) __interrupt 9;
void set_PW(void);
unsigned char read_AD_input(unsigned char n);
void read_accel(void);
void Pick_Heading(void);
int read_ranger(void);
void Set_Desired_Heading(void);
int read_compass(void);
void Pick_Heading_kp(void);
void Pick_Heading_kd(void);
void Set_Fan_Angle(void);

// Global variables
signed int left_pw;
```

```
signed int right_pw;
signed int Error = 0;
signed int prev_error = 0;
unsigned char Counts, nCounts;
unsigned char new_range = 0;
unsigned char new_AD = 0;
unsigned char new_heading;
unsigned char h_count;
unsigned char r_count = 0;
unsigned char adc_count = 0;
unsigned int DRIVE_PW = 2760;
unsigned int STEER_PW = 2760;
unsigned char Data[2]; // Data is an array with a length of 2
unsigned char print_delay = 0;
unsigned char AD_Result = 0;
unsigned char voltage = 0;
unsigned int heading = 0;
unsigned int range = 0;
unsigned int desired_heading = 0;
float heading_kp = 0;
unsigned char heading_kd = 0;
int print_error = 0;
signed int init_heading = 0;
unsigned int time = 0;

//=========================================================================
//-------------------------------------------------------------------------
// Main Function
void main(void)
{
      Sys_Init(); // System Initialization - MUST BE 1st EXECUTABLE STATEMENT
      Port_Init();
      Interrupt_Init();
      PCA_Init();
      ADC_Init();
      SMB0_Init();
      putchar('\r');  // Dummy write to serial port
      printf("\nStart\r\n");
      PCA0CP0 = 0xFFFF - PW_CENTER;
      PCA0CP1 = 0xFFFF - PW_CENTER;
      PCA0CP2 = 0xFFFF - PW_CENTER;
      PCA0CP3 = 0xFFFF - PW_CENTER;
      Counts = 0;
      while (Counts < 1);  // Wait a long time (1s) for motors to initialize

      // Read the number pad or SecureCRT terminal to set the initial desired
heading
      // Be able to alter data by raising or lowering hand over the
      Set_Fan_Angle();
      Pick_Heading();
      Pick_Heading_kp();
      Pick_Heading_kd();
//    Pick_Altitude_kp();
//    Pick_Altitude_kd();
```

```
//    drive_gain = (long)heading_kp;
//    printf("\rDrive gain: %u\n", drive_gain);
      printf_fast_f("\rhkp: %2.1f \n\rhkd: %u", heading_kp, heading_kd);
      Counts = 0;
      nCounts = 0;
       printf("\n\r------------DATA COLLECTION------------\n");
       printf("\n\rDesired Heading  |      Actual Heading     |     Time
(ms)\n\r");
      while (1)
      {
                  if ((new_range)) // enough overflow for a new range
                  {
                        new_range = 0;     //clear and wait for next ping
                        Set_Desired_Heading();
                  }

                  if(new_heading)
                  {#include <stdio.h>
#include <stdlib.h>
#include <c8051_SDCC.h> // Include files. This file is available online in LMS
#include <i2c.h>        // Get from LMS, THIS MUST BE INCLUDED AFTER stdio.h
#define PCA_START 28672 // 28672 for exactly 20ms
#define MAX_RANGE 55
#define PW_CENTER 2760
#define PW_MAX 3500
#define PW_MIN 2030

//-----------------------------------------------------------------------------
// Function Prototypes
// ....

// Global variables
// ....

//=============================================================================
main function
      Initialize the sys, port, interrupt, PCA, ADC, SMBus, and accelerometer
    Set the PCA to start in a center position

    Wait a long time (1s) for motors to initialize

      Read compass gain as input from the user in the SecureCRT command window
      Read drive gain as input from the user in the SecureCRT command window

      while the car is still moving
            printf("\n\r------------DATA COLLECTION------------\n");
            printf("\n\rX-Accel           |     Y-Accel           |
STEER_PW    |     DRIVE_PW\n\r");
      end while

      while true
            while both of the switches are off
                  if the accelerometer is ready to be read
                        reset the accelerometer flag
```

```
                        read the accelerometer
                        set the pulsewidth accordingly (refer to set_PW()
below)
                  end if statement

                  if the analog to digial converter needs to be read
                        reset the A/D flag
                        Read analog input on pin 1.5
                        Convert result back to input voltage
                  end if statement

                  Output the results for transfer into excel every 20 cycles
            end while loop

              if either of the switches is turned on
                        Stop the car
                        Reread steering gain as inputed by the user
                        Reread drive gain as inputed by the user
                        Pick_Steering_Gain();
                        Pick_Drive_Gain();
              end if statement
      end while loop
end of main function

                  }
                  if(new_AD)
                  {
                        new_AD = 0;
                        AD_Result = read_AD_input(5); //Read analog input on
pin 1.5
                        voltage = ((12.8/255)*(AD_Result)); //Convert back to
input voltage
                  }
                  if(print_delay == 20)
                  {
                        time += 400;
                        printf("\r%u            |     %u          |     %u\n",
(desired_heading/10), (heading/10), time);
                        print_delay = 0;
                  }

                  // Output the results for transfer into excel
      }
}
//***************************************************************************
//---------------------------------------------------------------------------
// Set up ports for input and output
void Port_Init(void)
{
      XBR0 = 0x27;
      P1MDIN      &= 0x7F;    // set pin 1.5 for analog input
      P1MDOUT |= 0x0F;  //set output pin for CEX0-3 in push-pull mode
      P1MDOUT &= 0x7F;  // set input pin for 1.5 to open-drain
      P1          |= ~0x7F;   // set input pin for 1.5 to high impedence
```

```
	P3MDOUT &= 0x7F;  // set input pin for 3.6/7 to open-drain
	P3           |= ~0x7F;   // set input pin for 3.6/7 to high impedence

}

//-------------------------------------------------------------------------
// Set up interrupts
void Interrupt_Init(void)
{
	IE |= 0x02;
	EIE1 |= 0x08;
	EA = 1;
}

//-------------------------------------------------------------------------
// Set up Programmable Counter Array
void PCA_Init(void)
{
	PCA0MD = 0x81;     // SYSCLK/12, enable CF interrupts, suspend when idle
	PCA0CPM0 = 0xC2;   // 16 bit, enable compare, enable PWM
	PCA0CPM1 = 0xC2;
	PCA0CPM2 = 0xC2;
	PCA0CPM3 = 0xC2;
	PCA0CN = 0x40;     // enable PCA
}

//-------------------------------------------------------------------------
// Set up the SMB
void SMB0_Init(void)     // This was at the top, moved it here to call wait()
{
	SMB0CR = 0x93;     // Set SCL to 100KHz
	ENSMB = 1;         // Enable SMBUS0
}

//-------------------------------------------------------------------------
// PCA_ISR: Interrupt Service Routine for Programmable Counter Array Overflow
Interrupt
void PCA_ISR(void) __interrupt 9
{
	if (CF)
	{
	CF = 0;                  // clear the interrupt flag
	nCounts++;                           // Counts overflows for initial
delay
	PCA0 = PCA_START;
	if (nCounts > 50)              //Initial one second delay
	{
		//nCounts = 0;
		Counts++;                // seconds counter
	}
		print_delay++;                        // delay for print statements
		r_count++;
		if (r_count>=12)              //delay for ranger reading
		{
```

```
                  new_range = 1;
                  r_count = 0;
            }
            h_count++;
            if (h_count >=8)
            {
                  new_heading = 1;
                  h_count = 0;
            }
            adc_count++;
            if(adc_count >=10)
            {
                  adc_count = 0;
                  new_AD = 1;
            }
      }
      else PCA0CN &= 0xC0;           // clear all other 9-type interrupts
}

//-----------------------------------------------------------------------------
--
// Analog/Digital Conversion Initialization
void ADC_Init(void)
{
      REF0CN = 0x03; // Set Vref to use internal reference voltage (2.4 V)
      ADC1CN = 0x80; // Enable A/D converter (ADC1)
      ADC1CF |= 0x01; // Set A/D converter gain to 1
}

//-----------------------------------------------------------------------------
--
// Analog/Digital Conversion Function
unsigned char read_AD_input(unsigned char n)
{
      AMX1SL = n; // Set P1.n as the analog input for ADC1
      ADC1CN = ADC1CN & ~0x20; // Clear the "Conversion Completed" flag
      ADC1CN = ADC1CN | 0x10; // Initiate A/D conversion

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

      return ADC1; // Return digital value in ADC1 register
}

//-----------------------------------------------------------------------------
//Selecting the drive gain function
void Pick_Heading(void)
{
      char input;
      printf("\rPlease select a desired heading.\n");
      printf("\r'u' will increment by 5 degrees. 'd' will decrement by 5
degrees.\n");
      printf("\r'f' when finished\n");
      while(1)
      {
```

```
            input = getchar();
            if(input == 'u') desired_heading += 50;
            if(input == 'd') desired_heading -= 50;
            if(input == 'f')
            {
                  init_heading = (int)desired_heading;
                  return;
            }
            if(desired_heading >= 3600) desired_heading = 3600;
            if(desired_heading <= 0) desired_heading = 0;
            printf("\rDesired heading: %u\n", (desired_heading)/10);
      }
}

//---------------------------------------------------------------------------
-
// Compass Reading Function
int read_compass(void)
{
      unsigned char addr = 0xC0; // the address of the sensor, 0xC0 for the
compass
      unsigned char Data[2]; // Data is an array with a length of 2
      unsigned int read_heading; // the heading returned in degrees between 0
and 3599
      i2c_read_data(addr, 2, Data, 2); // read two byte, starting at reg 2
      read_heading =((Data[0] << 8) | Data[1]); //combine the two values
      return read_heading; // the heading returned in degrees between 0 and
3599
}

//---------------------------------------------------------------------------
--
// new feature - read value, and then start a new ping
int read_ranger(void)
{
      unsigned char addr = 0xE0; // the address of the sensor, 0xC0 for the
compass
      unsigned int st_range = 0; // the range
      i2c_read_data(addr, 2, Data, 2); // read two byte, starting at reg 2
      st_range =((Data[0] << 8) | Data[1]); //combine the two values
      // The following lines convert the result to centimeters
      Data[0] = 0x51 ; // write 0x51 to reg 0 of the ranger:
      i2c_write_data(addr, 0, Data, 1) ; // write one byte of data to reg 0 at
addr
      return st_range;
}

//---------------------------------------------------------------------------
//Adjusting the steering servo
void set_PW(void)
{
      signed long temp_motorpw;
      // compass error equations (stored as Error)
      Error = (desired_heading) - heading;       //Calculate the error
```

```
      if(Error < -1800)  Error = Error + 3600;  //Adjust error so that we turn
efficiently
      if(Error > 1800)   Error = Error - 3600;
      temp_motorpw = (long)((heading_kp)*(long)(Error)) +
((long)(heading_kd)*(long)(Error - prev_error));
      if(temp_motorpw > 800) temp_motorpw = 800;
      if(temp_motorpw < -800) temp_motorpw = -800;
      print_error = prev_error;
      prev_error = Error;
      left_pw = PW_CENTER - (int)temp_motorpw;
      right_pw = PW_CENTER + (int)temp_motorpw;
      //Stay within limits of the servo
      if(left_pw < 2200) left_pw = 2200;  // min
      if(left_pw > 3300) left_pw = 3300;  // max
      if(right_pw < 2200) right_pw = 2200;
      if(right_pw > 3300) right_pw = 3300;

      PCA0CP0 = 0xFFFF - (PW_CENTER + (int)temp_motorpw); // Change pulse width
      PCA0CP2 = 0xFFFF - right_pw;
      PCA0CP3 = 0xFFFF - left_pw;
}
//-------------------------------------------------------------------------
//Adjusting the desired heading
void Set_Desired_Heading(void)
{
      int temp_heading = 0;
      range = read_ranger();
      if(range > 100) range = 100;
      temp_heading = (init_heading) + ((50 - range)*(36));
      while(temp_heading >= 3600) temp_heading -= 3600;
      while(temp_heading <= 0) temp_heading += 3600;
      desired_heading = temp_heading;
}
//-------------------------------------------------------------------------
//Picking initial control constants
void Pick_Heading_kp(void)
{
      char input;
      printf("\rPlease select a desired heading kp.\n");
      printf("\r'u' will increment by 0.1. 'd' will decrement by 0.1.\n");
      printf("\r'f' when finished\n");
      while(1)
      {
            input = getchar();
            if(input == 'u') heading_kp += 0.1;
            if(input == 'd') heading_kp -= 0.1;
            if(input == 'f') return;
            if(heading_kp >= 15) heading_kp = 15;
            if(heading_kp <= 0) heading_kp = 0;
            printf_fast_f("\rDesired heading kp: %2.1f\n", heading_kp);
      }
}
void Pick_Heading_kd(void)
{
```

```
	char input;
	printf("\rPlease select a desired heading kd.\n");
	printf("\r'u' will increment by 1. 'd' will decrement by 1.\n");
	printf("\r'f' when finished\n");
	while(1)
	{
		input = getchar();
		if(input == 'u') heading_kd += 1;
		if(input == 'd') heading_kd -= 1;
		if(input == 'f') return;
		if(heading_kd >= 200) heading_kd = 200;
		if(heading_kd <= 0) heading_kd = 0;
		printf("\rDesired heading kd: %u\n", heading_kd);
	}
}
//-----------------------------------------------------------------------------
//Setting the fan angle to start
void Set_Fan_Angle(void)
{
	char input;
	PCA0CP1 = 0xFFFF - 2300;
	printf("\rAdjust fan angle. 'u' for up, 'd' for down, 'f' when
finished.\n");
	while(1)
	{
		input = getchar();
		if(input == 'u') PCA0CP1 -= 10;
		if(input == 'd') PCA0CP1 += 10;
		if(input == 'f') return;
		if(PCA0CP1 > (0xFFFF - 2000)) PCA0CP1 = (0xFFFF - 2000);
		if(PCA0CP1 < (0xFFFF - 3500)) PCA0CP1 = (0xFFFF - 3500);
		printf("\rPW: %u\n", (0xFFFF - PCA0CP1));
	}
}
```