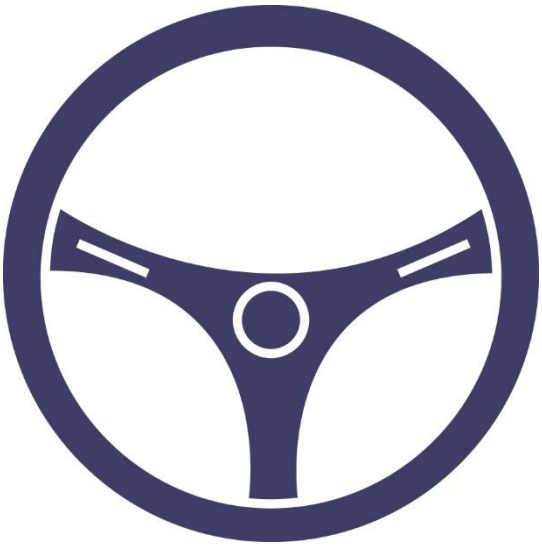


4/4/2015

Steering Control

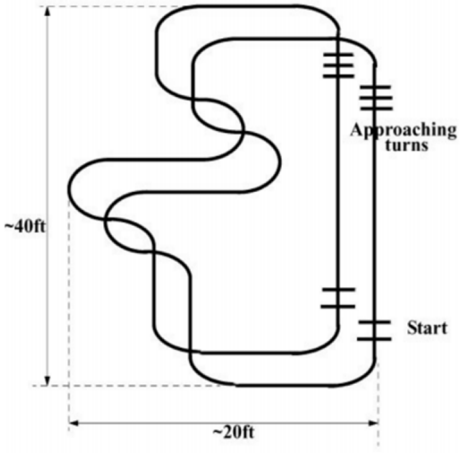
ELE 302 Project #2

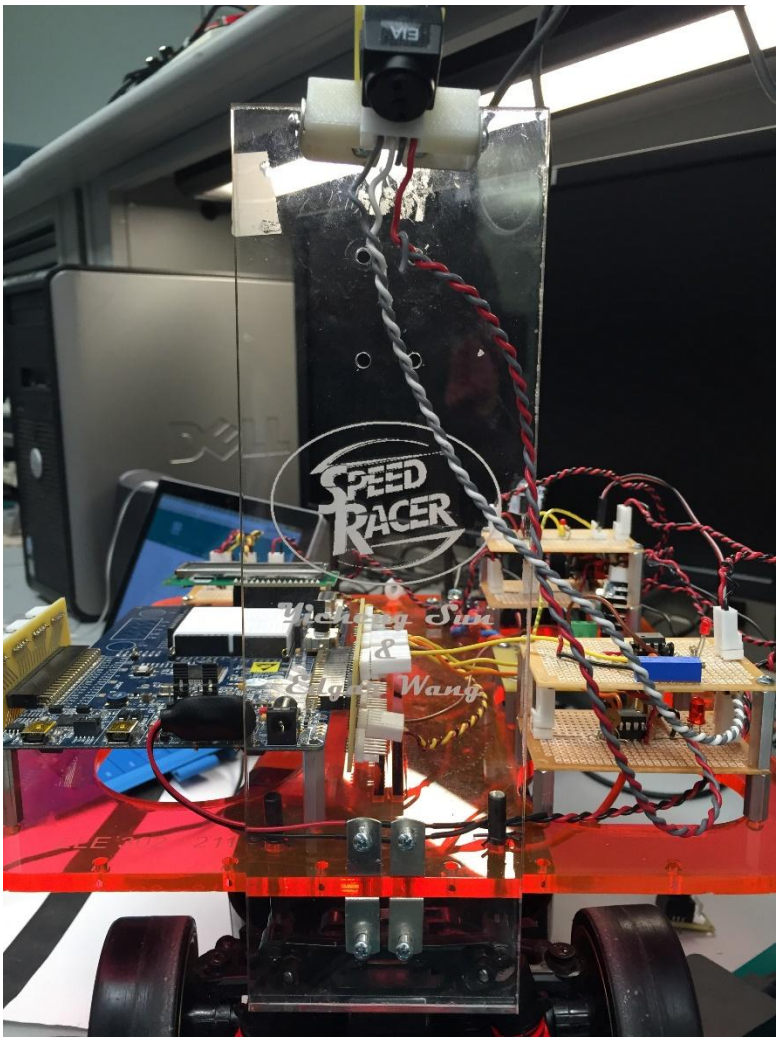
Yicheng Sun and Edgar Wang

Steering Control

ELE 302 Project #2

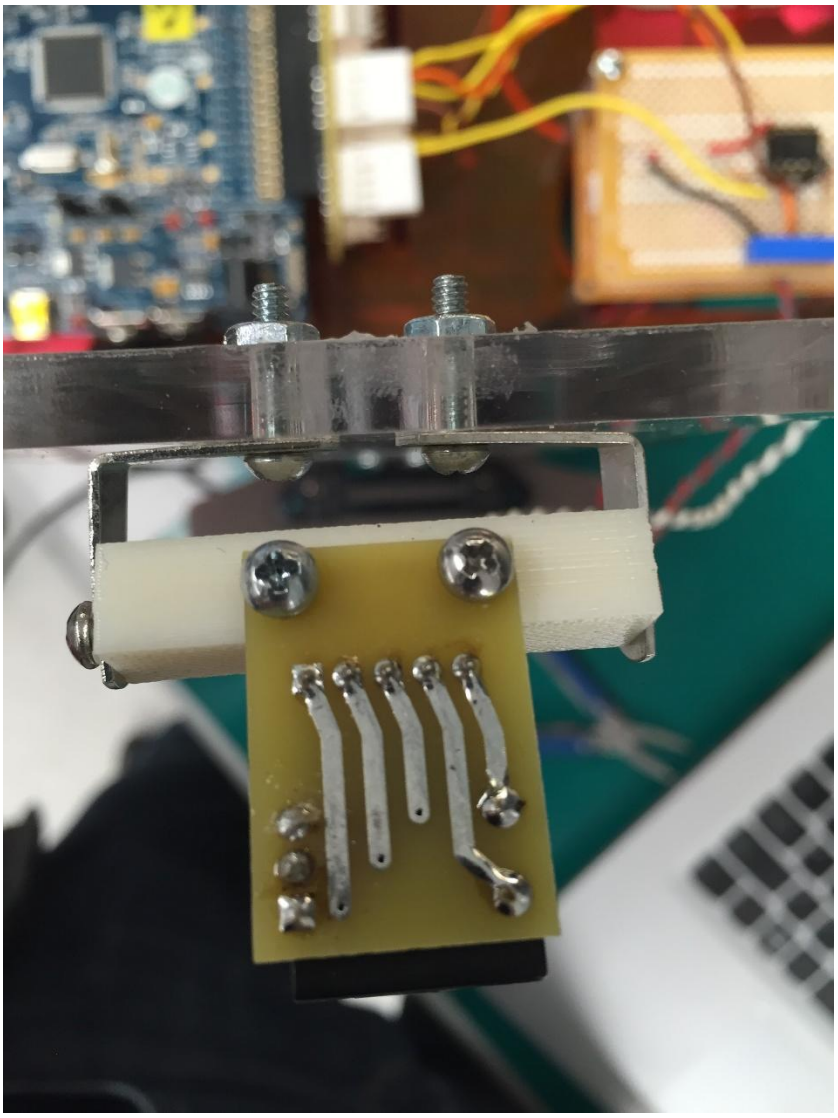
The purpose of this project was to extend our previous car to follow a track without any external control. The car must travel at a 3 ft/s speed or faster, make turns of 3 foot radius, and complete the track twice. In order to do this we installed a camera that read the line and implemented PD control center the car on the line as it moves forward. This is broken into three parts, the camera and sync separator, hardware comparator and reading data with PSOC, and PD control code. The complete finished car can be seen on the right.





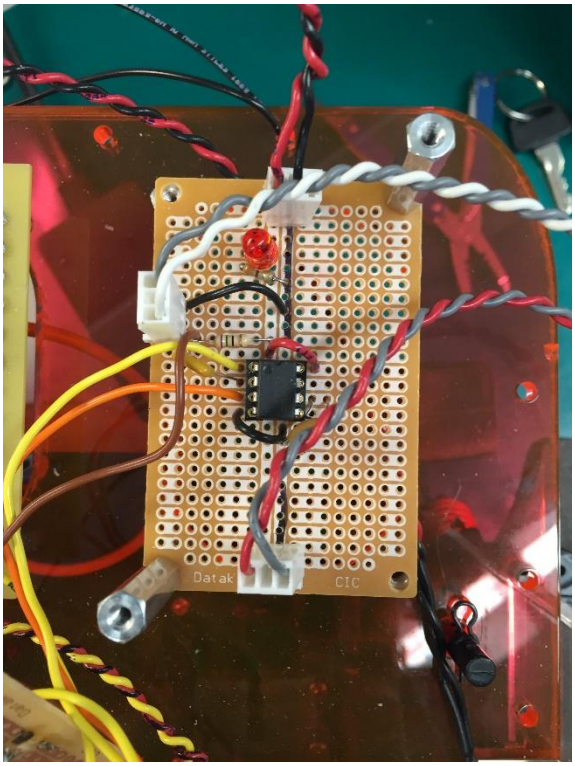
Camera and Camera Mast

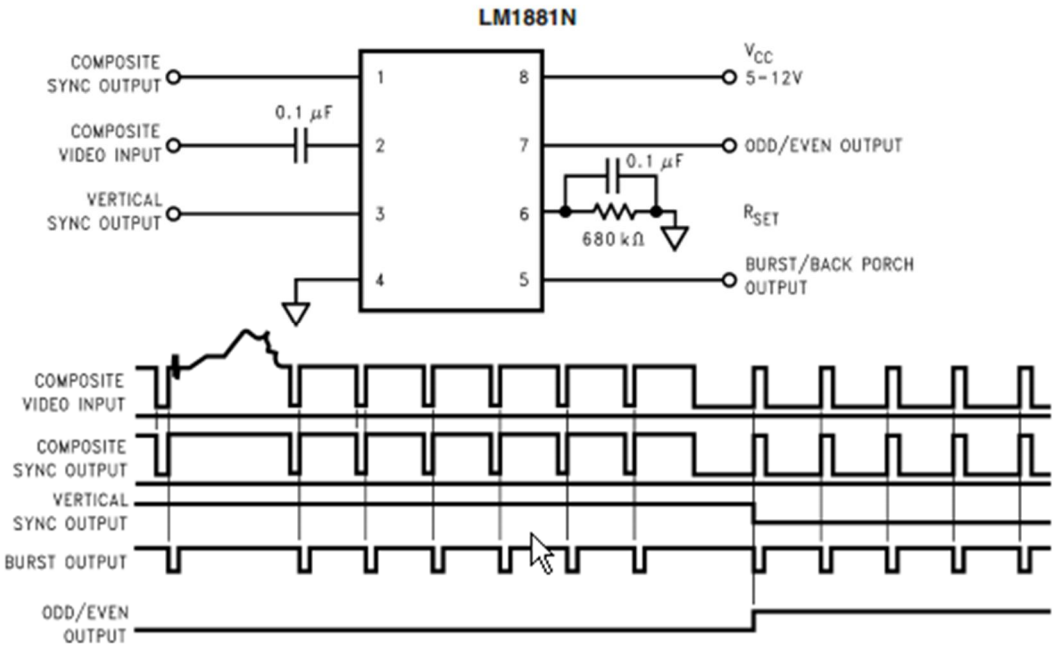
We were supplied the standard C-Cam2 video camera. The camera output was connected to the sync separator board (with a 750hm resistor pulling output to ground). 5V power was supplied via a voltage regulator, and only one of the camera grounds was connected to ground as to avoid a ground loop. To secure the camera at a certain angle and height, we laser cut a mast out of acrylic.



Sync Separator

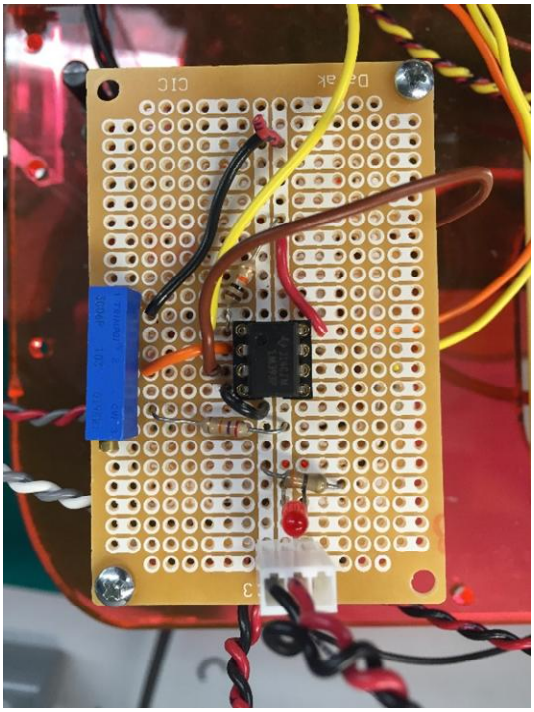
The LM1881 sync separator board allowed us to determine where the separate lines and frames where in the composite video output. The sync separator breaks the composite video output into a composite sync output (with each high low marking a line), and the vertical sync (with each high low marking a frame). The odd/even frame was unnecessary, and the capacitor from the video input and the RC circuit coming out of pin 6 had values chosen from the datasheet. We added a 75ohm resistor from the camera output to ground for impedance matching. In addition a red LED in series with a 330 ohm resistor was added in parallel with the voltage source for easy debugging.





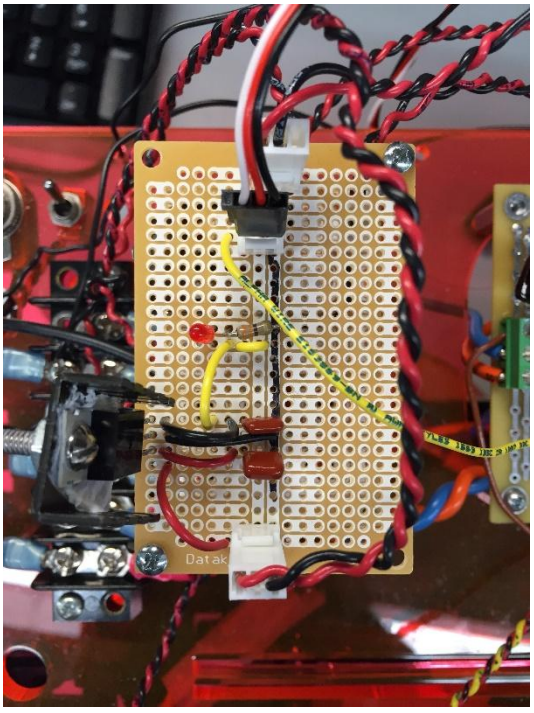
Comparator Board and Steering Voltage Regulator

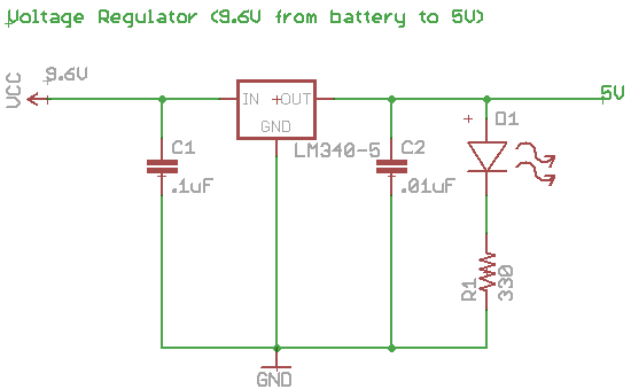
We decided to use a hardware comparator (top picture) over a software comparator for multiple reasons. One reason was simplification of the programmed hardware on the PSOC. We would've required a comparator and an internal voltage source connected to the PSOC's potentiometer. This left a lot of room for small bugs, and we had some difficulty with our PSOC's potentiometer and voltage source. The hardware removed the need for these programmed hardware pieces and made it simple for us to change the voltage threshold to compare white to black pixels and re-calibrate the car on the track. We chose to use the LM393 as it had low current drain (.4mA), a low input offset voltage (5mV) helpful because the threshold needed to be very accurate for timely output changes, and a response time in a typical 1.5us, which was fast enough to react to the pixel color changes. We used a pull-up resistor of 1k (small enough to pull the line high while not too large to draw too much current) to give a lower time delay constant in comparator response times to voltage changes.



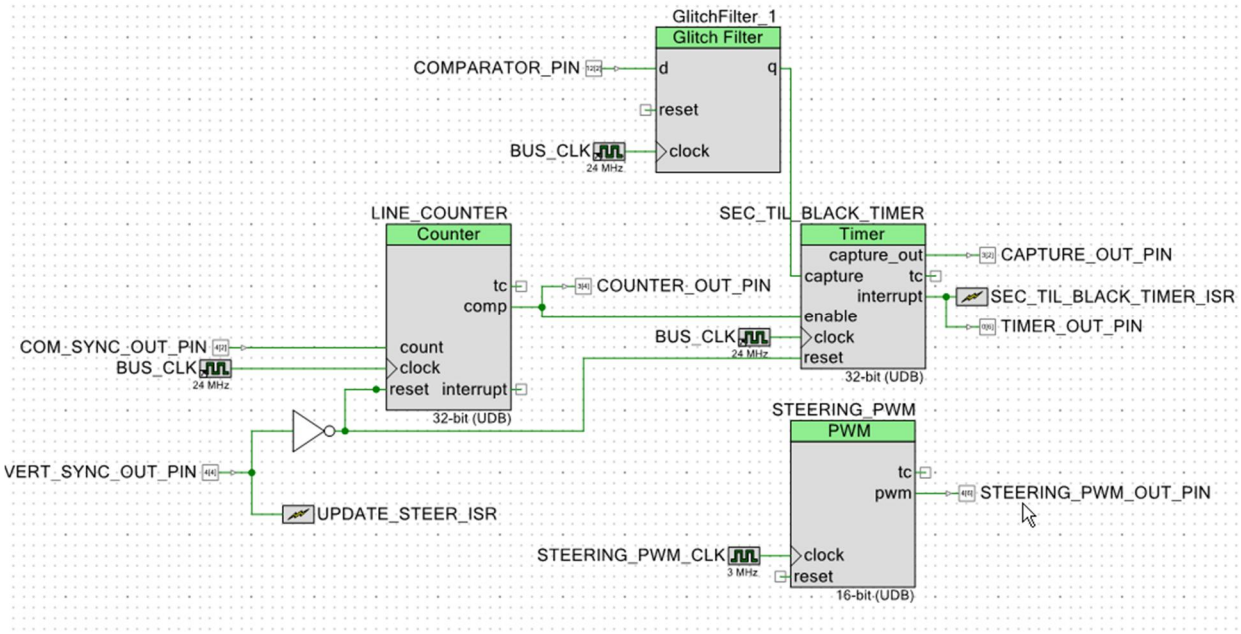
Ultimately the potentiometer was set to 3.38 KΩ to properly differentiate between the black tape and dirty ground of the lab.

In addition, we built another 5V voltage regulator (bottom picture) to deal with strong current draw from the servos. The servo generally pumps 10mA but turning can pump up to 500mA. The new board would supply this current and prevent the servo from interfering with the current provided to other more sensitive boards, like the Hall Effect sensor.





PSOC Hardware, Camera Reading Code



CAMERA READING

Our programmed hardware made use of a counter to count the line number, a glitch filter to smooth out input from the comparator, and a timer to determine where the black line position was in terms of timer units. We used the position of the line by capturing only one position per frame and comparing that position to the intended position to the keep the black line in the middle of the camera frame. We relied more on hardware than software by using interrupts and resets in hardware, which allowed our system to run much faster (software interrupts were too slow to keep up).

Counter

The counter reset on vertical sync low, which allowed it to reset at the end of every frame and prepare it to count anew on the new frame. The compare value was set to 25 (which meant the 50th line due to the camera scanning evens then odds). When the compare value was high, it was connected to the enable of the timer.

Timer

The timer was also reset on vertical sync low, which allowed it to reset at the end of every frame like the counter and avoid overflow. The timer was enabled on the compare value of the counter high. The counter is high for an entire line, which allows the timer to function for an entire line. The timer triggers an interrupt on two captures, and the timer captures low edges from the comparator output. So after the timer has captured the first falling edge for the black line and second falling edge for the end of the frame, an interrupt is triggered. The two captures provide enough information to determine the position of the black line in the frame.

Glitch Filter

The comparator output is passed through a glitch filter then into the timer. The glitch filter prevents the timer from reading oscillatory output from the comparator since there's a nebulous region in the comparator in which the compare can be read as low or high, and the glitch filter prevents these small perturbations from leading to oscillatory readings. In addition, it can remove some noise that could've caused the comparator to have small irregular changes when it first encounters readings from the black line and lead to faulty captures of the timer.

Code

```
CY_ISR(UPDATE_STEER_inter) {
    updateSteering();
}

CY_ISR(SEC_TIL_BLACK_TIMER_inter) {
    char buffer[15];
    gfirstpos = SEC_TIL_BLACK_TIMER_ReadCapture();
    gsecondpos = SEC_TIL_BLACK_TIMER_ReadCapture();
    SEC_TIL_BLACK_TIMER_ClearFIFO();

    gblack_totalpos_diff = (double)(gsecondpos - gfirstpos);
    if (gblack_totalpos_diff < 280){
        gblack_totalpos_diff = 280;
    }
    else if (gblack_totalpos_diff > 1200){
        gblack_totalpos_diff = 1200;
    }

    SEC_TIL_BLACK_TIMER_ReadStatusRegister();
    }
```

PD Control Code

The steering was updated on every frame. Our approach was to read around the 50th line (Early enough in the frame to detect turns quickly), check where the left side of the black line was in relation to the right edge of the camera, and capture that as a position.

We wanted to keep the position of the black line centered so we calibrated for a position->PWM mapping that would keep the car centered. We also included an algorithm that would make the car move faster along straight lines and slower along curves (to give it more time to react to the turns). The code also forced the car to turn hard right and hard left once the calculated PWM hit certain thresholds to allow for successful turns even if the line is lost.

We only used proportional control as integral control wasn't helpful due to fast movements of the line left and right of center. Derivative control could have been helpful in navigating turns, our speed modification algorithm that slowed the car down around turns was sufficient.

```
void updateSteering() {
    double error;
    double duty_cycle_buffer;
    uint16 duty_cycle;
    char buffer[10];


    //Calculate the error for feedback
    error = gblack_totalpos_diff - CENTER_LINE;

    // left max 3600; center 4560; right max 5800
    //gki_steererror = gki_steererror+error*.000011;
    gsteer_dutycycle = CENTER_DUTY - Kp_steer*error -
Kd_steer*(gsteer_error_prev-error)/395372.0;
    gsteer_error_prev=error;


//    LCD_ClearDisplay();
//    LCD_Position(0,0);
//    sprintf(buffer, "%f", error);
//    LCD_PrintString(buffer);
//    LCD_Position(1, 0);
//    sprintf(buffer, "%f", duty_cycle_buffer);
//    LCD_PrintString(buffer);

    if (gblack_totalpos_diff > CENTER_LINE){
        gExpectedSpeed = 5.5 - (gblack_totalpos_diff-CENTER_LINE)/240.0*2;
    }
    else {
        gExpectedSpeed = 5.5 - (CENTER_LINE-gblack_totalpos_diff)/310.0*2;
    }
```

```
//Have in place error checking to prevent sporadic  behavior
if (gsteer_dutycycle > 5700){
    gsteer_dutycycle = 5800;
}
if (gsteer_dutycycle <= 3700) {
    gsteer_dutycycle = 3550;
}

STEERING_PWM_WriteCompare(gsteer_dutycycle);
```

Complete Code

```
#include <project.h>
#include <stdio.h>

#define INCH_PER_MAGNET 1.58
#define SEC_PER_PERIOD 357.914

#define THREE_FT_DUTY 900
#define CENTER_DUTY 4560
#define CENTER_LINE 780

#define Kp 1250
#define Ki 120
#define Kd 0

#define Kp_steer 2.25
#define Ki_steer 0
#define Kd_steer 30000

double gExpectedSpeed = 3.5;
double gTotalTraveled = 0;

double gprev_HE_count = 0;
int gfirst_HE_read = 1;
int gspeedMeasurements = 0;
double gcurSpeed = 0;
double speedCounts[5];
double gki_speederror = 0;
double gki_steererror = 0;
int glinepos = 0;
double gsteer_dutycycle = 0;

int gnum_line_reads = 0;
double gblack_pos_first_diff = 0;
double gblack_pos_second_diff = 0;
double gblack_totalpos_diff = 0;
int gCounterNReads = 0;
int gblackcount = 0;
uint32 gfirstpos = 0;
uint32 gsecondpos = 0;
uint32 gthirdpos = 0;
uint32 gfourthpos = 0;
uint32 gcaptures = 0;
int count = 0;
int gsteer_error_prev = 0;

//Averages out speed for the last wheel rotation to even out magnet spacing
double getSpeedAvg(double speeds[]){
    double counter = 0;
```

```
    uint32 i = 0;
    uint32 size = 5;
    if (gspeedMeasurements < 5) {
        size = gspeedMeasurements;
    }
    for (i = 0; i < size; i++){
        counter = counter + speeds[i];
    }
    return counter/(double)size;
}

//Grab current speed via unit conversions
double getCurSpeed(){
    double current_Speed = 0;
    //average clock tix b/w two magnets in one rotation
    current_Speed = getSpeedAvg(speedCounts);
    //average sec elapsed b/w two magnets
    current_Speed = (double)current_Speed/HE_TIMER_ReadPeriod() *
SEC_PER_PERIOD;
    //average speed b/w two magnets
    current_Speed = (double)INCH_PER_MAGNET/current_Speed/12;
    // return (double)current_Speed;
    return current_Speed;
}

void updateSteering() {
    double error;
    double duty_cycle_buffer;
    uint16 duty_cycle;
    char buffer[10];


    //Calculate the error for feedback
    error = gblack_totalpos_diff - CENTER_LINE;

    // left max 3600; center 4560; right max 5800
    //gki_steererror = gki_steererror+error*.000011;
    gsteer_dutycycle = CENTER_DUTY - Kp_steer*error -
Kd_steer*(gsteer_error_prev-error)/395372.0;
    gsteer_error_prev=error;


//    LCD_ClearDisplay();
//    LCD_Position(0,0);
//    sprintf(buffer, "%f", error);
//    LCD_PrintString(buffer);
//    LCD_Position(1, 0);
//    sprintf(buffer, "%f", duty_cycle_buffer);
//    LCD_PrintString(buffer);

    if (gblack_totalpos_diff > CENTER_LINE){
        gExpectedSpeed = 5.5 - (gblack_totalpos_diff-CENTER_LINE)/240.0*2;
    }
```

```
    else {
        gExpectedSpeed = 5.5 - (CENTER_LINE-gblack_totalpos_diff)/310.0*2;
    }
    //Have in place error checking to prevent sporadic  behavior
    if (gsteer_dutycycle > 5700){
        gsteer_dutycycle = 5800;
    }
    if (gsteer_dutycycle <= 3700) {
        gsteer_dutycycle = 3550;
    }

    STEERING_PWM_WriteCompare(gsteer_dutycycle);
}

CY_ISR(UPDATE_STEER_inter) {
    updateSteering();
}

CY_ISR(SEC_TIL_BLACK_TIMER_inter) {
    char buffer[15];
    gfirstpos = SEC_TIL_BLACK_TIMER_ReadCapture();
    gsecondpos = SEC_TIL_BLACK_TIMER_ReadCapture();
    SEC_TIL_BLACK_TIMER_ClearFIFO();

    gblack_totalpos_diff = (double)(gsecondpos - gfirstpos);
    if (gblack_totalpos_diff < 280){
        gblack_totalpos_diff = 280;
    }
    else if (gblack_totalpos_diff > 1200){
        gblack_totalpos_diff = 1200;
    }

    SEC_TIL_BLACK_TIMER_ReadStatusRegister();
    }

//Interrupt on each hall effect sensor passing by to update speed and PWM
duty cycle
CY_ISR(HE_inter) {
    double curr_HE_count = 0;
    double time_diff = 0;
    double time_diff_s = 0;
    double error = 0;
    double PID_speed = 0;
    char buffer[15];
    double duty_cycle_buffer = 0;
    uint16 duty_cycle = 0;
    gTotalTraveled+=INCH_PER_MAGNET;

    //Special first time read
    if (gfirst_HE_read == 1) {
        gprev_HE_count = HE_TIMER_ReadCounter();
        gfirst_HE_read = 0;
    }
```

```
    else {
        curr_HE_count = HE_TIMER_ReadCounter();
        if (gprev_HE_count < curr_HE_count) {
            gprev_HE_count = gprev_HE_count + HE_TIMER_ReadPeriod();
        }

        //Calculate the time difference between each magnet passing by
        time_diff = gprev_HE_count - curr_HE_count;
        time_diff_s = time_diff/HE_TIMER_ReadPeriod() * SEC_PER_PERIOD;

        speedCounts[gspeedMeasurements%5] = time_diff;
        gspeedMeasurements++;
        gcurSpeed = getCurSpeed();
        gprev_HE_count = curr_HE_count;
        //Calculate the error for feedback
        error = gExpectedSpeed - gcurSpeed;
        //Accumulate past errors for Ki
        gki_speederror = gki_speederror+error*time_diff_s;
        // Discard saved error from acceleration as it becomes less relevant
after starting
        if (gspeedMeasurements == 28) gki_speederror = 0;
        //Calculate the duty cycle based upon Kp, Ki, and Kd
        duty_cycle_buffer = THREE_FT_DUTY + Kp*error + Ki*gki_speederror +
Kd*error/time_diff_s;

       //LCD output for debugging
       // LCD_ClearDisplay();
       // LCD_Position(0,0);
       // sprintf(buffer, "%f", gsteer_dutycycle);
       //LCD_PrintString(buffer);
    //LCD_Position(1, 1);
       //LCD_PrintString("//");
       //sprintf(buffer, "%f", gblack_totalpos_diff);
       //LCD_PrintString(buffer);

        //Have in place error checking to ensure duty cycle goes to 1 if
negative and caps at a
        //certain duty cycle to prevent sporadic  behavior
        if (duty_cycle_buffer > 4000){
            duty_cycle_buffer = 1350;
        }
        if (duty_cycle_buffer <= 0) duty_cycle_buffer = 1;
        if (gTotalTraveled > 2450){
            duty_cycle_buffer = 1;
        }
        duty_cycle = duty_cycle_buffer;


//        //more LCD debugging code
//        LCD_Position(1, 0);
//        //sprintf(buffer, "%f", duty_cycle);
//        LCD_PrintNumber(duty_cycle);
```

```
        MOTOR_PWM_WriteCompare(duty_cycle);
    }
}

int main()
{
    //initialize all modules
    CYGlobalIntEnable;
    HE_TIMER_Start();
    HE_ISR_Start();
    HE_ISR_SetVector(HE_inter);
    MOTOR_PWM_Start();
    MOTOR_PWM_CLK_Start();

    LINE_COUNTER_Start();
    SEC_TIL_BLACK_TIMER_ISR_Start();
    SEC_TIL_BLACK_TIMER_ISR_SetVector(SEC_TIL_BLACK_TIMER_inter);
    SEC_TIL_BLACK_TIMER_Start();

    UPDATE_STEER_ISR_Start();
    UPDATE_STEER_ISR_SetVector(UPDATE_STEER_inter);

    STEERING_PWM_Start();
    STEERING_PWM_CLK_Start();

    LCD_Start();
    LCD_Position(0,0);
    LCD_PrintString("ELE302 Carlab");
    for(;;)
    {
    }
}
```