ELE 302 Task 2: Line Follower

Ethan Gordon and Luke Pfleger

April 1, 2016

Contents

1 Abstract

The second task of this project was to implement steering control. On the floor of the ELE lab, two tracks were created using black tape. The tracks had straightaways, curves, and parts where they crossed over each other. The goal was to automate the car such that it could complete two laps of either track without deviating from the line, maintaining an average speed of at least 3 ft/s.

To accomplish this, we used a C-Cam2 video camera to follow the track and determine the position of the tape in relation to the car. The car then adusted the angle of its wheels based on the location of the tape, and thus was able to self-correct on straightaways, and to follow the curvature of the track on turns.

We tried a couple methods first: measuring the position of the black line using a single horizontal line of the video, or taking the position at two separate lines and finding the difference, which we converted to the angle between the car and the tape. In the end, we decided to find the average position of the two lines, to find roughly the center position of the tape relative to the car.

Using this method, we were able to complete two laps of the track in 57s, maintaining an average speed of approximately 3.5 ft/s across the run. After attempting to increase our speed, our fastest time dropped to 19.05s for a single lap, achieving an average speed of approximately 5.2 ft/s.

2 Hardware Design

2.1 Physical Modifications

To accomplish this task, we utilized a camera to view the track and convert the black and white signals to voltages to ascertain the position of the tape in each frame. To attach this camera to our car we created a mast out of acrylic. The mast was designed on Adobe Illustrator, and cut using a laser printer. It can be seen in figure 1. It measures 2.5in wide by 11in tall. It has holes at the top by which the camera can be mounted on the mast, and holes on the bottom by which the mast can be mounted on the car, as can be seen in figure 2.

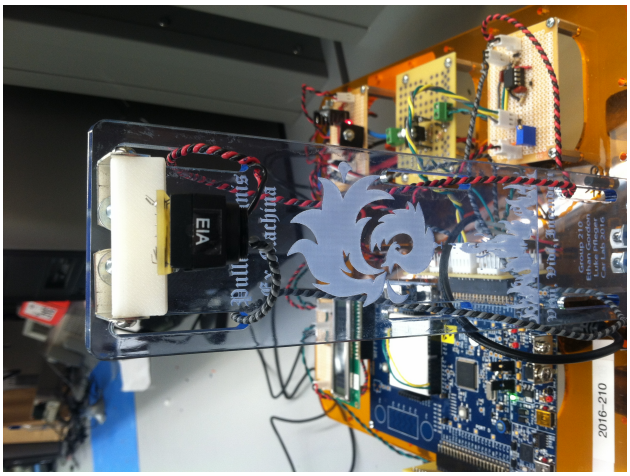

Figure 1: The acrylic mast with the camera mounted from the front.

It also has long holes down the sides through which to thread the wires being sent to the camera. This way, they cannot be pulled out easily, helping to protect the camera and the wiring.

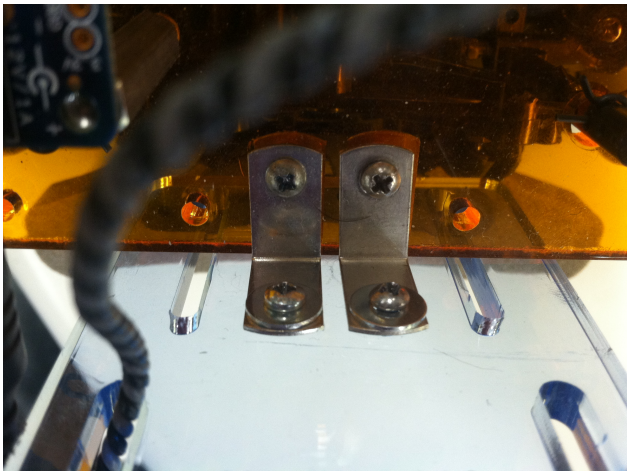

Figure 2: The L-Brackets used to mount the acrylic mast (blue) on the acrylic base of the car (orange).

In addition, to modify the car from the previous task for use in the current task, we removed the RC receiver completely, and attached the motor controlling the steering directly to the PSoC. This way, our car was

completely in control of its own steering as well as its speed, and there was no remote control with which to steer it as there had been previously.

The PSoC controlled the steering motor through a pulse width modulation (PWM) signal. The duty cycle of the signal corresponds to the angle in a linear fashion: a longer duty cycle turns the wheels further to the right, while a shorter signal turns the wheels further to the left. The motor operates on a range of roughly 1-2ms, where a duty cycle of 1.47ms (determined empirically) sets the wheels straight.

2.2 Sensing

The camera that we used was a C-Cam2. It read in the input as a series of lines in an interlaced format, meaning that first it read in every odd-numbered horizontal line in order, and then every even-numbered horizontal line in order. The camera as mounted on the mast can be seen in Figure 1. We read in the input from the camera and put it through a sync separater circuit. The sync separator circuit used the LM1881 chip, which takes the composite video feed as input as well as 5V for power, and outputs a variety of signals. The circuit as modelled in LTSpice can be seen in Figure 3, and the physical implementation on our machine can be seen in Figure 4.

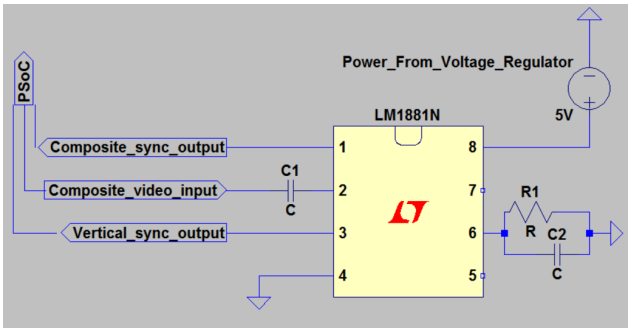

Figure 3: The simulation of the Sync Seperator circuit on LTSpice. This circuit takes the input of the video feed from the camera, and then outputs three signals–the video feed, the composite sync, and the vertical sync– to the PSoC.

The Sync Separator circuit was implemented as described in the data sheet, with capacitors and resistors applied as defined by the producers of the chip. Additionally, the circuit has a potentiometer, which was originally used as part of a comparator. The purpose of the comparator is to create a threshold to separate the white floor from the black tape, such that each pixel can be separated into low (white) and high (black), so that the position of the tape can be ascertained. However, we discovered that creating the comparator entirely on the PSoC is more effective than having it as a separate circuit, so we discontinued use of the potentiometer on this circuit.

2.3 Actuation

The last hardwae changes to the car for this task involved powering the last segments of the car. Providing power to the motors controlling the steering proved a challenge, because the motors produced so much noise that they had to be separate from the rest of the circuits. To this end, we added a second voltage regulator to provide power to the motors. The LTSpice simulation of the updated circuit can be seen in Figure 5 and the implementation of the circuit can be seen in Figure 6.

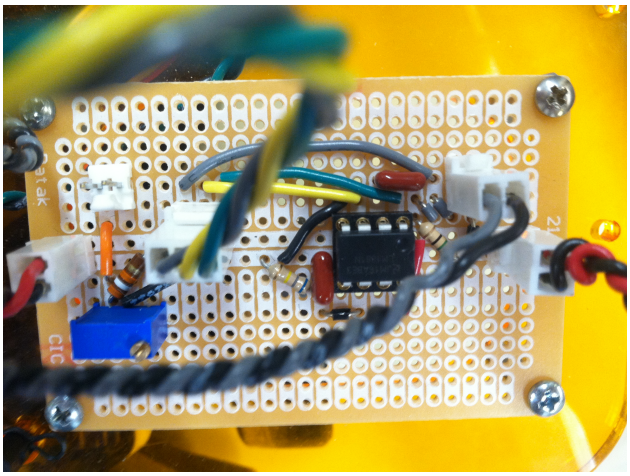

Figure 4: The sync seperator circuit as it is implemented on our car. The three-way kk connector contains the outputs to the PSoC. The potentiometer in the corner is a vestigial part that was originally used as part of a comparator, but is no longer in use. This circuit also has a line of 5V power going to the camera.

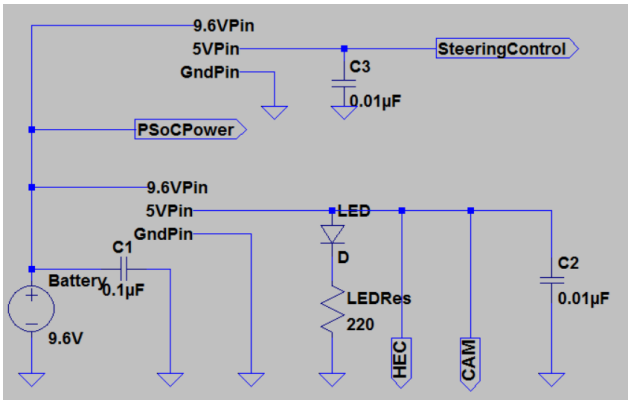

Figure 5: The simulation of the Voltage Regulator circuit on LTSpice. This circuit takes the input of the 9.6V from the battery, and then outputs 5V to be used as power for the different circuits. The first voltage regulator sends power to the Hall Effect Sensor Circuit (HEC) and the Camera (CAM), while the second voltage regulator sends power to the motor controlling the steering (SteeringControl).

The voltage regulator sends power to the Hall Effect Circuit to manage the speed control, as well as the camera. The second voltage regulator sends power to the steering motors.

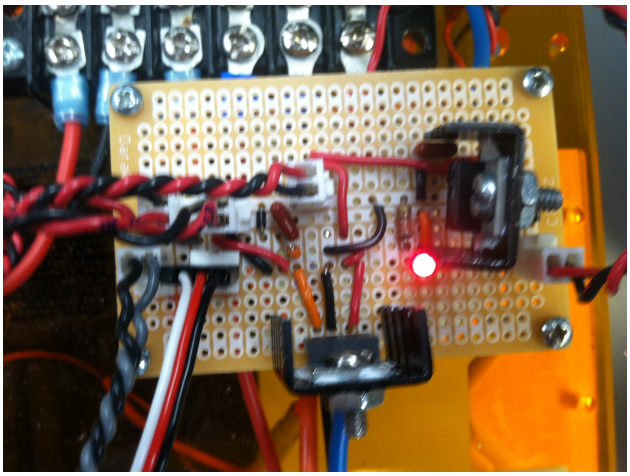

Figure 6: A picture of the implementation of the voltage regulator circuit. The original voltage regulator can be seen on the right side of the screen with its outputs in the left-center, while the additional voltage regulator can be seen on bottom, used to power the steering motor.

3 Software Design (PSoC)

3.1 Firmware

As shown in Figure 7, our Firmware additions were fairly extensive for this task. Our main inputs were the Composite Video (CV), Composite Sync (CSync), and the Vertical Sync (VSync).

For line selection, we fed the VSync and CSync into one-shot counters. The VSync, as the reset signal, would reset the counter continuously until the negative edge, which ocurred at right at the beginning of every frame. Each horizontal line, the CSync would decrement the counter. Therefore, the period of the counter would select the line number, and it would release a pulse at the beginning of that line.

For the actual line position measurement, we first converted the analog CV signal into a digital signal using the PSoC's internal comparator and reference voltage (using a VDAC). The Comparator would directly compare the reference voltage from the VDAC and the CV signal, outputting a digital 1 or 0 on each clock edge. The waveform itself can be seen in Figure 8. At the beginning of each line, the wave would go low with the sync signal, but otherwise, it would only go low when the black line was in view. We set our reference voltage just above the reference black threshold for the brightness level of the lab (468 mV), so only the line would be captured. This is the cause of the messier signal, but because only the first falling edge is measured, the noise did not cause any issues.

The actual measurement was carried out by a timer with a resolution of 0.1 μs. The signal from the counter would start the timer, and it would stop at the first falling edge of the Thresholded CV, capturing the value and interrupting. As the timer counted down, the output values ranged from about 0 (all the way to the right) to about 623 (all the way to the left). We copied the counter and timer to get measurement for two different lines.

For actuation, we simply used a PWM module, set to a 10ms period, or 100Hz frequency. We could adjust the length of the pulse (and thus the angle of the servo) by setting the CMP value, with each increment corresponding to 1 μs. Empirically, a CMP value of 1470 (1.47 ms) corresponded with a turn angle of 0.

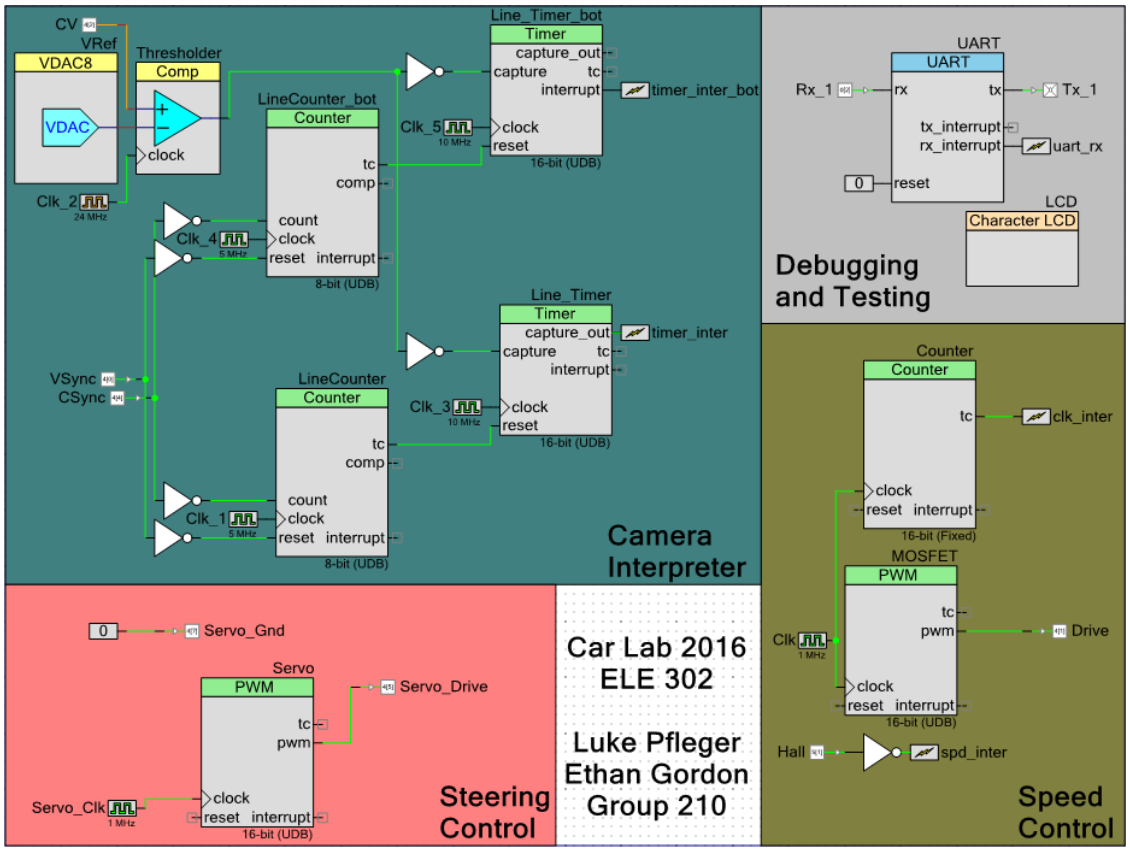

Figure 7: Full firmware diagram for the PSoC. Additions for Task 2 include the Servo Controller and Camera Interpreter.

3.2 Closed-Loop Controller

Our main control loop can be found in the Appendix, Section 5.1. The layout is similar to the speed controller, but even simpler, as only a proportional gain was required. The *timer* and *timer_bot* interrupts were called from the Timer firmware objects. These interrupts read the capture from the timer and, if the value was sane, recorded it. A value less than 67 only occurred if the camera lost the black line, and so those values were rejected. In our original code, the main loop itself simply took the measurements form both lines, added the values (to average without the computationally costly division), found the error relative to the setpoint (empirically set to 720, or $72 = 2 * 36\mu s$), multiplied by the proportional gain, and sent the whole thing to the angle controller (Section 5.2), which would center the value around 1470 (the proportionality constant was factored out into the proportional gain) and write to the PWM module.

We could make a educated guess for our gain magnitude early on. Our maximum error was going to be on the order of 600 (300 from each line measurement), and our maximum output was 30° degrees, calling for a Kp of about $\frac{30}{600} = 0.05$. However, when we factored the proportionality constant (about 12) out of the angle controller, we had to increase our Kp to $0.05 * 12 = 0.6$. This proved to be the basis for our testing and results.

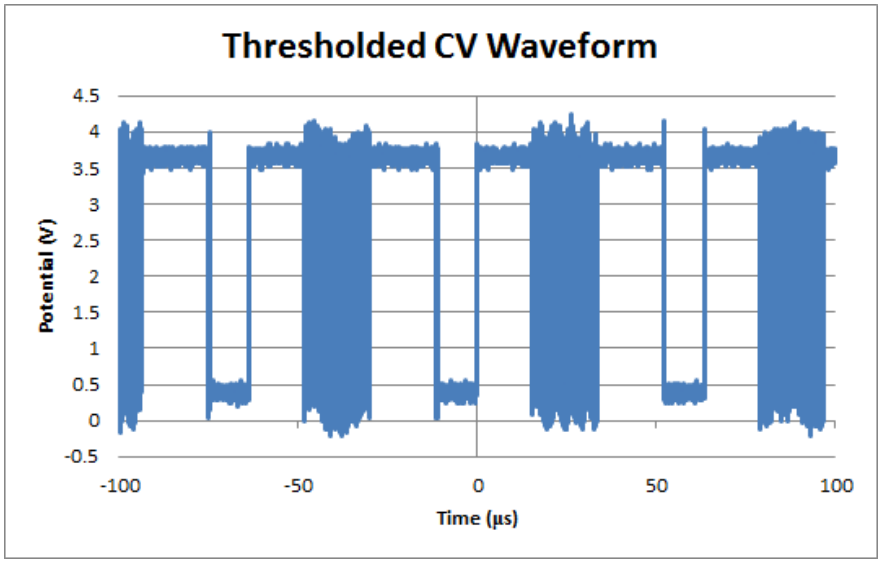

Figure 8: Quantized composite video waveform, demonstrating the full 62.3 μs frame. The clear dip is the horizontal sync, and the opaque section is the black line. The capture in the timer would trigger on the first falling edge, so the messiness is inconsequential.

4 Testing and Results

4.1 Issues

Before our first successful run, we actually ran into a few subtle issues that needed fixing. The first was a large amount of noise in our time measurements. This ended up being caused by a slow clock in our line counter. Increasing the clock frequency from 200 kHz to 5 MHz improved our measurement accuracy drastically.

Secondly, and most importantly, we encountered a serious amount of delay in our initial design, up to 500ms between the movement of the line and the movement of the wheels. Such a delay made it impossible to set our Kp to a reasonable value without excessive oscillations on the straightaway portion of the track. This delay could be attributed to two factors. First of all, in order to reduce the amount of rewritten code, we created PID Objects (Section 5.3) to abstract away the specifics of PID control. While powerful, this object contained a lot of code that was unnecessary for the simple loop that we needed for the direction control, and so we restricted the use of this object to just the speed controller. Secondly, we incorporated a millisecond delay at the end of our main while loop to make the integral component of the speed controller more predictable. Removing this delay and adjusting our I gain accordingly also greatly improved upon the delay.

4.2 Parameter Tuning

We experimented with Kp in the ballpark of 0.6. Three example laps are shown in Figure 9. Low values, like 0.36, made for smoother rides, but we would often lose the line. At values of 0.35 or below, we would lose the line on the first turn. High values, like 0.9, would keep better track of the line, but they would also introduce sever oscillations. At values above 0.9, the oscillations became unstable after hte first lap, making the car undriveable. Figure 10 shows our cumulative results, the average error for each value of Kp. It fits well to a quadratic curve, allowing us to find the optimal Kp:

$$Err_{sum} = 6 * 10^6 * (Kp)^2 - 8 * 10^6 * (Kp) + 3 * 10^6 \quad (1)$$

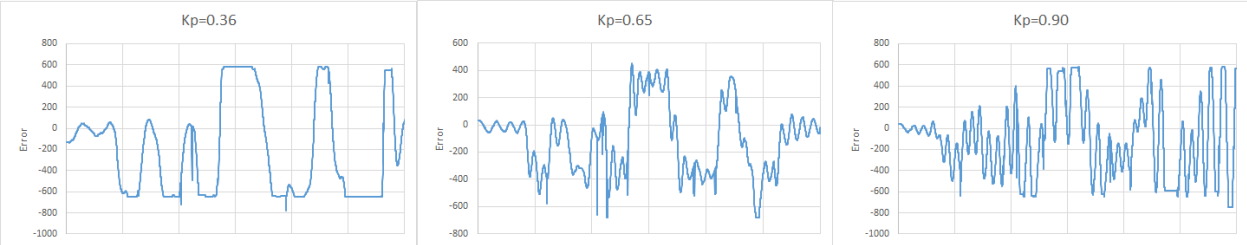

Figure 9: Measured error over the course of a single lap for three different values of Kp. When the error reaches above 600 or below -800, the line has left the vision of the camera. Note: the dip to -800 towards the end of each lap corresponded with a kink in the otherwise smooth track, leaving a sharper turn than usual.

$$Kp_{opt} = \frac{-(-8 * 10^6)}{2 * 6 * 10^6} = 0.66 \tag{2}$$

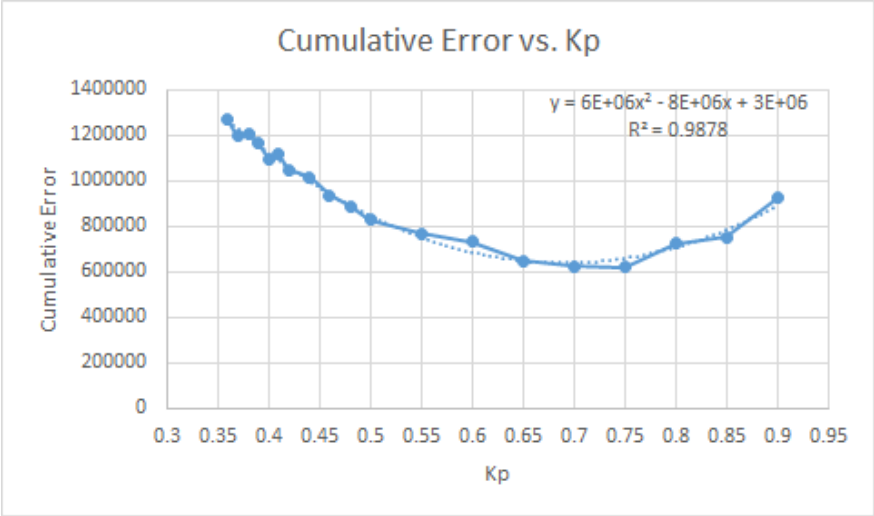

Figure 10: Sum of the absolute value of the error over the course of a single lap for various values of Kp. We were able to fairly accurately fit the data to a quadratic curve.

With our Kp set to 0.66, we were able to complete our 2 laps in **57s**.

4.3 Speed Improvements

We added some modifications to the code to allow the car to go faster. This mostly involved angle detection, or more accurately, detection of the difference between the two lines. From this "angle," we were able to determine the direction that the car was moving: Straight, Left, or Right. When the car was turning left, we selected the left-most point to use for PID, and vice versa. When the car moved straight, we continued to take the average between the lines and increased our speed. Wtih this detection, we were able to more readily reject intersection points where even a tiny confusion on the part of our faster-moving car would lead to a deviation from the track. In the end, our best single-lap time was **19.05s** and our best 2-lap time was **41.72s**.

5 Code Appendix

5.1 Main Loop

```
/* =======================================
 *
 * Copyright Ethan Gordon, Luke Pfleger, 2016
 * Car Lab, Task 2
 * Line Follower
 *
 * Goal: Follow a line for 2 laps.
 * =======================================
*/
#include <device.h>
#include <stdio.h>

/* State Estimation */
#include "state.h"

/* Angle Actuation */
#include "angle.h"

/* PID Loop */
#include "pid.h"

/* Constants */
#define ALPHA 0.2f
/* Total Circumference = 20cm = 0.656168 ft */
/* 5 Magnets per Revolution: 0.656168 / 5 = 0.1312336 ft */
#define MAG_DIST 0.1312336f
#define SECOND 1000
#define TICKS_PER_LAB 750
#define LAPS 1

/* Direction Detection */
#define DIR_ANG 20
#define DIR_CAP 60
#define DIR_THRESH 5


/** Changeable Constants **/
#define SPD_KP  600.0f
#define SPD_KI  0.8f
#define SPD_SETPOINT 4.0f

#define AVG_KP_DEF 0.7f
static float AVG_KP = AVG_KP_DEF;
#define AVG_SETPOINT 720

#define ANG_KP  0.0f
#define ANG_SETPOINT 0
```

```
/** END Changeable Constants **/

/* Macros */
#define ABS(x)   ( ( (x) < 0) ? -(x) : (x) )


/* State Estimators */
State_T spdState = NULL;

/* PID Loops */
PID_T spdPID = NULL;

/* UART Debug Interrupt */
static int16 dataArr[3500] = {0};
int restart = 0;
CY_ISR(uart)
{
    int i, j;
    char c = UART_GetChar();
    if(c == 'd') {
        /* Output Error Data from Last Run */
        for(i = 0; i < 3500; i++) {
            char tstr[9] = {0};
            if(i < 3499) sprintf(tstr, "%+4d, ", dataArr[i]);
            else sprintf(tstr, "%+4d", dataArr[i]);
            for(j = 0; j < 8; j++)
                UART_PutChar(tstr[j]);
        }
    } else if (c == '+') {
        /* Increase Kp */
        char tstr[11] = {0};
        AVG_KP += 0.05f;
        sprintf(tstr, "\nKp=%1.2f, ", AVG_KP);
        for(j = 0; j < 10; j++)
            UART_PutChar(tstr[j]);
    } else if (c == '-') {
        /* Decrease Kp */
        char tstr[11] = {0};
        AVG_KP -= 0.01f;
        sprintf(tstr, "\nKp=%1.2f, ", AVG_KP);
        for(j = 0; j < 10; j++)
            UART_PutChar(tstr[j]);
    } else if (c == 'k') {
        /* Reset Kp to Default */
        char tstr[11] = {0};
        AVG_KP = AVG_KP_DEF;
        sprintf(tstr, "\nKp=%1.2f, ", AVG_KP);
        for(j = 0; j < 10; j++)
            UART_PutChar(tstr[j]);
    } else if (c == 'r') {
        /* Start New Run */
```

```
        restart = 1;
    }

}

/* Clock and Clock Interrupt */
int millis = 0;
CY_ISR(clk)
{
    millis++;

    /* Watchdog Functions */
    State_Watchdog(spdState);
}

/* Line Interrupts */
static int topLineX = AVG_SETPOINT / 2;
static int linePos = AVG_SETPOINT / 2;
static int isLineLost = 0;

CY_ISR(timer)
{
    isLineLost = 0;
    topLineX = Line_Timer_ReadCapture();
    if (topLineX > 67)
        linePos = topLineX;
    else isLineLost = 1;
}

static int botLineX = AVG_SETPOINT;
static int linePosBot = AVG_SETPOINT;
CY_ISR(timer_bot)
{
    botLineX = Line_Timer_bot_ReadCapture();
    if (botLineX > 67)
        linePosBot = botLineX;
    else isLineLost = 1;
}

/* Hall Effect Interrupt */
static int ticks = 0;
CY_ISR(spd)
{
    static int lastTime = 0;
    int currentTime = millis;

    /* Update State Estimate */
    if (lastTime != 0) {
        State_Update(spdState, (MAG_DIST * 1000.0f)/((float)(currentTime - lastTime)))
    }
    lastTime = currentTime;
```

```
    ticks++;
}

void main()
{

    /* LCD Display */
    char tstr[17] = {0};

    CyGlobalIntEnable;

    /***** Firmware Declarations *****/

    /* Start Reference */
    VRef_Start();

    /* Start Comparator */
    Thresholder_Start();

    /* Start Counters */
    Counter_Start();
    LineCounter_Start();
    Line_Timer_Start();
    LineCounter_bot_Start();
    Line_Timer_bot_Start();

    /* Start LCD */
    LCD_Start();

    /* Start PWM */
    MOSFET_Start();
    Angle_Start();

    /* Start UART */
    UART_Start();

    /***** End Firmware Declarations *****/

    /***** Object Declarations *****/

    /* Start Speed State Estimator */
    spdState = State_Alloc();
    State_Start(spdState, ALPHA, 0.0f, SECOND);

    /* Start Speed PID Loop */
    spdPID = PID_Alloc();
    PID_Start(spdPID, SPD_KP, SPD_KI, 0.0f, (float)(MOSFET_ReadPeriod() + 1));

    /***** End Object Declarations *****/

    /***** Interrupt Declarations *****/
```

```
/* Clock Interrupt */
clk_inter_Start();
clk_inter_SetVector(clk);

/* Line Timer Interrupt */
timer_inter_Start();
timer_inter_SetVector(timer);
timer_inter_bot_Start();
timer_inter_bot_SetVector(timer_bot);

/* UART Interrupt */
uart_rx_Start();
uart_rx_SetVector(uart);


/* Speed Interrupt */
spd_inter_Start();
spd_inter_SetVector(spd);
/***** End Interrupt Declarations *****/

/*** Set PID Setpoints ***/
for(;;) {
    int i = 0;
    int dir = 0;
    /* Wait for Serial Restart */
    while(!restart) {
        LCD_Position(1, 0);
        LCD_PrintString("Wait on Restart");
    }
    /* 4 Second Delay */
    {
        int targetmillis = millis + 4000;
        LCD_Position(1, 0);
        LCD_PrintString("Starting........");
        while(millis < targetmillis);
    }
    ticks = 0;
    restart = 0;
    PID_Setpoint(spdPID, SPD_SETPOINT);

    while(ticks < (TICKS_PER_LAB * LAPS))
    {
        int16 err, avg, ang;

        /* Update PID Loop */
        PID_Update(spdPID, State_Avg(spdState), 1);
        avg = linePos + linePosBot;
        ang = linePos - linePosBot;

        /**** Direction Detection ****/
```

```
        if (!isLineLost) {
            if(ang > DIR_ANG) {
                dir += 2;
            } else if (ang < -DIR_ANG) {
                dir -= 2;
            } else {
                if(dir > 0) dir--;
                else dir++;
            }
        }

        /* Clamp Direction Variable */
        if (dir < -DIR_CAP) dir = -DIR_CAP;
        if (dir > DIR_CAP) dir = DIR_CAP;

        if (dir > DIR_THRESH) { /* LEFT: Use left-most point. */
            if (linePos > linePosBot)
                avg = linePos + linePos;
            else avg = linePosBot + linePosBot;
        } else if (dir < -DIR_THRESH) { /* RIGHT: Use right-most point. */
            if (linePos < linePosBot)
                avg = linePos + linePos;
            else avg = linePosBot + linePosBot;
        } /* Else STRAIGHT: Use average of points. */

        /*** END Direction Detection ***/

        /* Debug Output */
        LCD_Position(0, 0);
        sprintf(tstr, "Avg:␣%5d", avg);
        LCD_PrintString(tstr);
        LCD_Position(1, 0);/*
        if(dir > DIR_THRESH) {
            LCD_PrintString("Left     ");
        } else if (dir < -DIR_THRESH) {
            LCD_PrintString("Right    ");
        } else {
            LCD_PrintString("Straight ");
        }*/
        sprintf(tstr, "Spd:␣%1.1f␣␣␣␣␣␣␣", State_Avg(spdState));
        LCD_PrintString(tstr);

        /* Calculate Error */
        err = AVG_SETPOINT - avg;
        if(i < 7000) dataArr[i/2] = err;

        /* Write */
        MOSFET_WriteCompare(PID_Output(spdPID));
        Angle_Set((int)(AVG_KP * (float)err));
        i++;
    }
```

```
        /* Stop */
        MOSFET_WriteCompare(0);
        Angle_Set(0.0f);
    }
}

/* [] END OF FILE */
```

5.2 Angle Conversion

5.2.1 Header

```
/* ========================================
 *
 * Copyright Ethan Gordon, Luke Pfleger, 2016
 * Car Lab, Task 2
 * Angle Interface
 *
 * ========================================
*/
#ifndef ANGLE_H
#define ANGLE_H

/**
  * Initialize Angle Servo
  **/
void Angle_Start();

/* Set Angle */
void Angle_Set(int angle);

#endif
/* [] END OF FILE */
```

5.2.2 Module

```
/* ========================================
 *
 * Copyright Ethan Gordon, Luke Pfleger, 2016
 * Car Lab, Task 2
 * Angle Module
 *
 * ========================================
*/
#include "angle.h"
#include <device.h>

/**
  * Initialize Angle Servo
  **/
void Angle_Start()
{
```

```
    Servo_Start();
    Angle_Set(0.0f);
}

/* Set Angle (in degrees), Clamps from -30 to 30 */

/* Servo_WriteCompare(1470); Center */
/* Servo_WriteCompare(1847); Right */
/* Servo_WriteCompare(1093); Left */
/* angle = m*x + b; m = 0.079576; b=-116.98 */
/* inverted slope: 1/m = 12.5667 */
/* inverted intercept: -b/m = 1470 */
void Angle_Set(int angle)
{
    int output;

    output = angle + 1470;

    Servo_WriteCompare(output);
}

/* [] END OF FILE */
```

5.3 PID Object

5.3.1 Header

```
/* ========================================
 *
 * Copyright Ethan Gordon, Luke Pfleger, 2016
 * Car Lab, Task 2
 * PID Loop Interface
 *
 * ========================================
*/
#ifndef PID_H
#define PID_H

/* PID Object */
typedef struct PID* PID_T;

/* Pull a pointer for a PID Object. */
PID_T PID_Alloc();

/**
  * Initialize PID Object
  **/
void PID_Start(PID_T pObj, float kp, float ki, float lowClamp, float highClamp);

/* Change PID Setpoint, Also Resets Transient Measurement Information */
void PID_Setpoint(PID_T pObj, float setpoint);
```

```
/* Update PID Loop with value. If customDt == 0, dt is calculated from millisecond clo
void PID_Update(PID_T pObj, float val, int customDt);

/* Get output of PID loop at this time step. */
float PID_Output(PID_T pObj);

/*
 * Populates Array with Transient Information:
 * Rise Time, Overshoot, Undershoot,
 * and Steady State Estimation
 */
void PID_Transients(PID_T pObj, float* ret);

#endif
/* [] END OF FILE */
```

5.3.2 Module

```
/* ========================================
 *
 * Copyright Ethan Gordon, Luke Pfleger, 2016
 * Car Lab, Task 2
 * PID Loop Module
 *
 * ========================================
*/
#include "pid.h"
#include <stdlib.h>

extern int millis;

/* Constants */
#define MAX_PID 1
#define EMA_ALPHA 0.1f
#define DEADBAND 0.001f
#define THRESHOLD 0.1f

/* Macros */
#define ABS(x)  ( ( (x) < 0) ? -(x) : (x) )

/* PID Structure */
struct PID {
    float kp, ki; /* Gains */
    float rt, ov, un, av; /* Transients */
    float set, out; /* Setpoint, Output */
    float sum; /* Past Error Information */
    float cLow, cHigh; /* Clamps */
    int lastTime, firstTime;
};

/* Static State Array */
struct PID mPIDArr[MAX_PID];
```

```
/* Pull a pointer for a PID Object. */
PID_T PID_Alloc()
{
    static int sAllocated = 0;
    if (sAllocated >= MAX_PID)
        return NULL;
    return &mPIDArr[sAllocated++];
}

/**
  * Initialize PID Object
  **/
void PID_Start(PID_T pObj, float kp, float ki, float lowClamp, float highClamp)
{
    pObj->kp    = kp;
    pObj->ki    = ki;
    pObj->set   = 0.0f;
    pObj->sum   = 0.0f;
    pObj->out   = 0.0f;
    pObj->cLow  = lowClamp;
    pObj->cHigh = highClamp;
    pObj->rt = pObj->ov = pObj->un = pObj->av = 0.0f;

    pObj->lastTime = 0;
    pObj->firstTime = 0;
}

/* Change PID Setpoint, Also Resets Transient Measurement Information */
void PID_Setpoint(PID_T pObj, float setpoint)
{
    pObj->set = setpoint;
    pObj->rt = pObj->av = 0.0f;
    pObj->un = pObj->ov = setpoint;
    pObj->lastTime = 0;
    pObj->firstTime = 0;
    pObj->sum   = 0.0f;
}

/* Update PID Loop with value. If customDt == 0, dt is calculated from millisecond clo
void PID_Update(PID_T pObj, float val, int customDt)
{
    float p, i, error;

    /* Custom Timestep in Millis */
    int dt = customDt ? customDt : millis - pObj->lastTime;
    pObj->lastTime = millis;

    /* Calculate Steady State Average */
    if (pObj->rt > DEADBAND) {
        pObj->av = EMA_ALPHA * val + (1.0f-EMA_ALPHA)*pObj->av;
```

```
    }

    /* Calculate Rise Time when value is within 10% of Setpoint */
    if (pObj->rt < DEADBAND && ABS(pObj->set - val) < (pObj->set * THRESHOLD))
        pObj->rt = (float)(millis - pObj->firstTime) / 1000.0f;

    /* Calculate Overshoot: Maximum value reached. */
    if (pObj->rt > DEADBAND && val > pObj->ov)
        pObj->ov = val;

    /* Calculate undershoot: Lowest value reached correcting overshoot. */
    if (pObj->ov > pObj->set && val < pObj->un)
        pObj->un = val;

    /*** PID Loop ***/
    /* Get Error */
    error = pObj->set - val;
    pObj->sum += (error * (float)(dt));

    /* Set PID */
    p = (pObj->kp * error);
    i = (pObj->ki * pObj->sum);

    pObj->out = (int)(p + i);

    /* Clamp */
    if (pObj->out > pObj->cHigh) pObj->out = pObj->cHigh;
    if (pObj->out < pObj->cLow)  pObj->out = pObj->cLow;
}

/* Get output of PID loop at this time step. */
float PID_Output(PID_T pObj)
{
    return pObj->out;
}

/*
 * Populates Array with Transient Information:
 * Rise Time, Overshoot, Undershoot,
 * and Steady State Estimation
 */
void PID_Transients(PID_T pObj, float* ret)
{
    ret[0] = pObj->rt;
    ret[1] = pObj->ov;
    ret[2] = pObj->un;
    ret[3] = pObj->av;
}


/* [] END OF FILE */
```