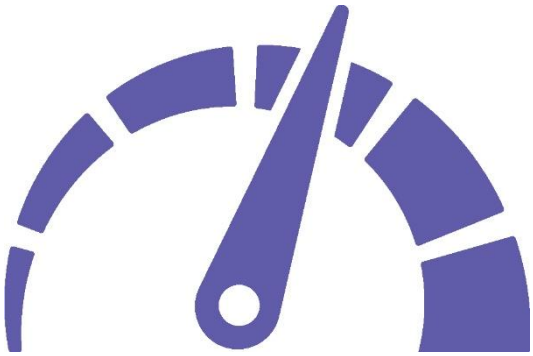


3/6/2015

Speed Control

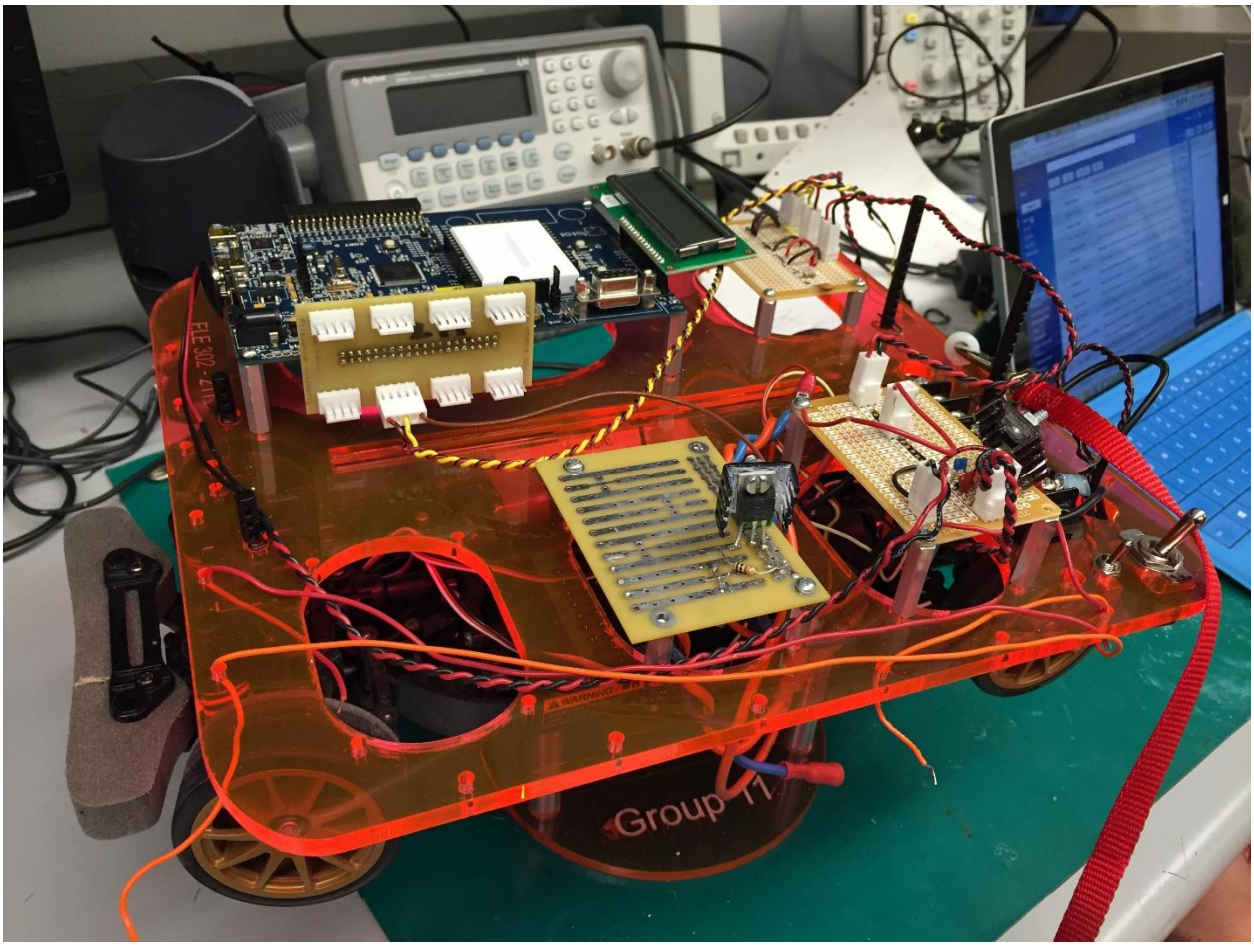
ELE 302 Project #1

Yicheng Sun and Edgar Wang

Speed Control

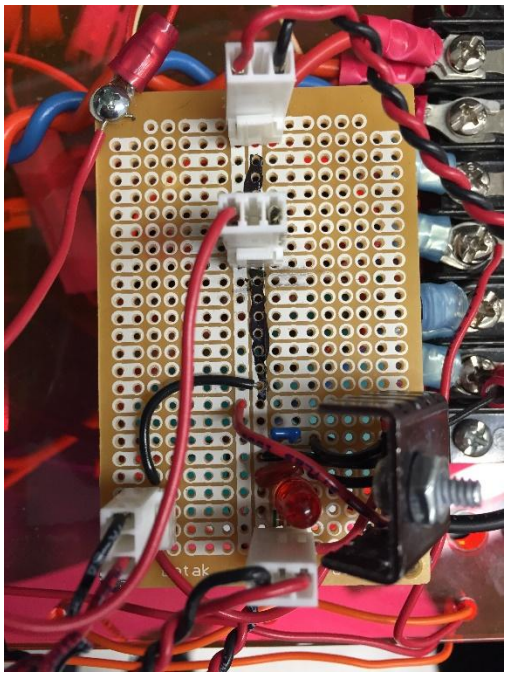
ELE 302 Project #1

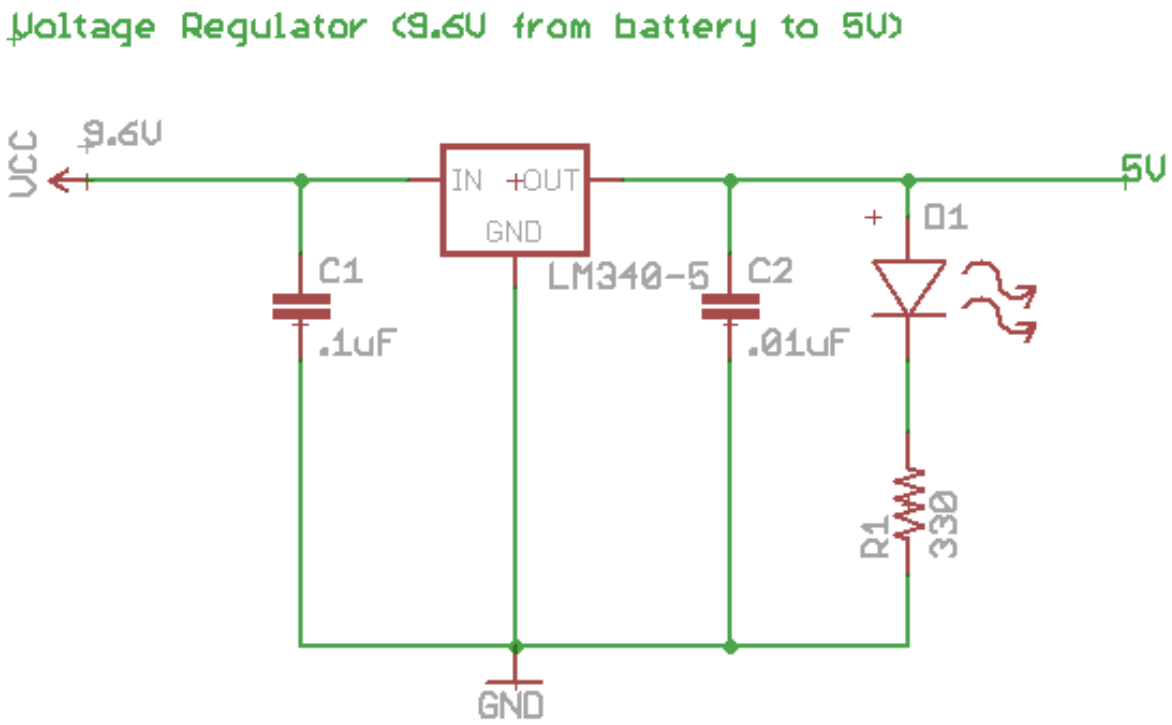
The purpose of this project was to deconstruct an RC car and replace the built-in radio receiver motor control with an autonomous speed control. To this purpose, we implemented PID control using a PSOC microprocessor. The completed car can be found in the image below. The critical components involved in its functionality are the power regulator board, sensor interface board, speed control board, and PSOC PID control.



Voltage Regulator Board

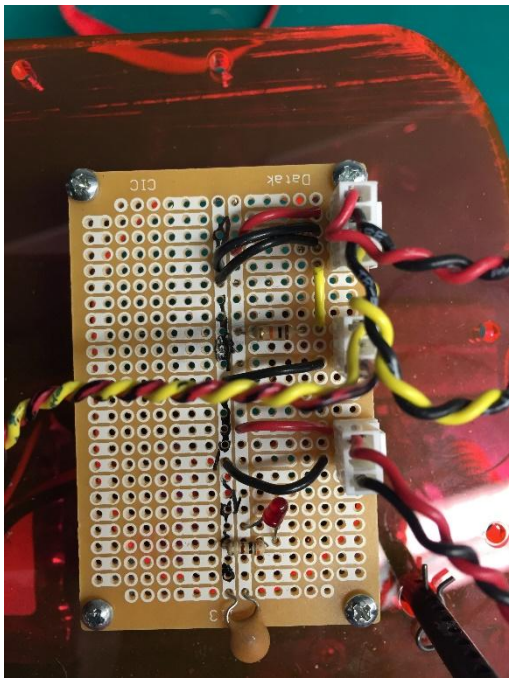
In order to power the PSOC and Hall Effect sensor, we created a 5V power supply source. The electronics battery is 9.6V which we stepped down with a 7805 voltage regulator (in this case an equivalent LM3405). We chose to use a voltage regulator instead of a voltage divider because as our system is battery operated, power consumption is a main concern. In addition, a voltage divider does not regulate a fixed output voltage and output will change based on current draw and input voltage. We added a pair of decoupling capacitors to reduce noise. Since battery noise is more significant than regulated output noise, we used 0.1uF and 0.01uF capacitors for input/output respectively. A red LED was added in parallel to detect when the regulator was correctly outputting. A current limiting resistor was added in series to prevent the LED from melting. We defaulted to a 330Ω resistor. However if needed to be more precise, we could have calculated the proper resistor using the LED forward voltage and the maximum current rating.





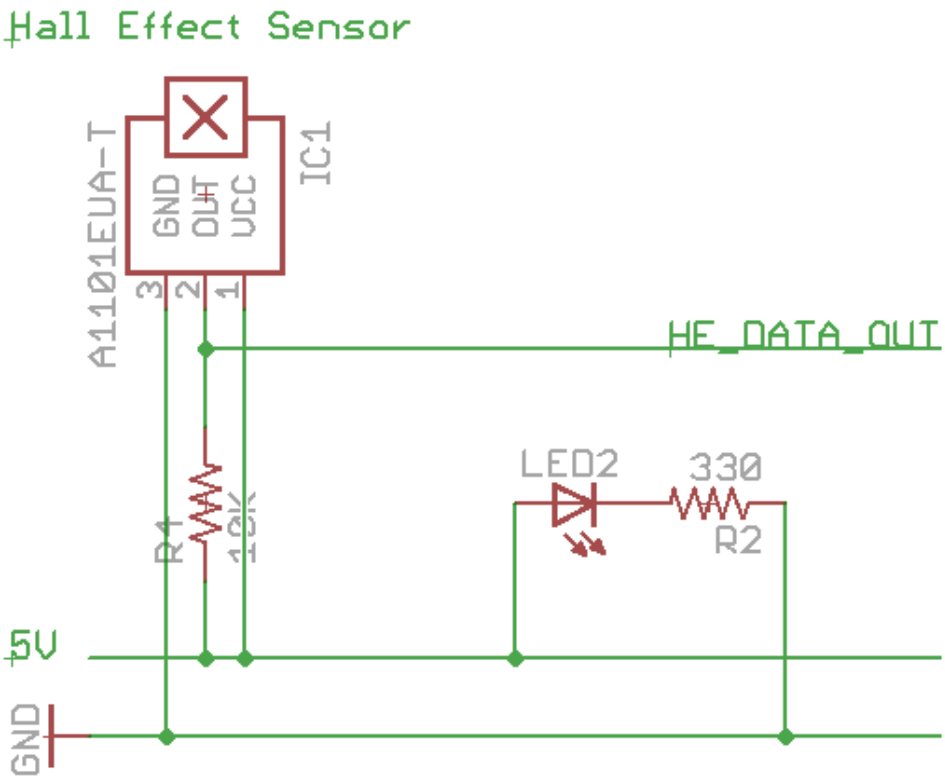
Sensor Interface Board

To figure out the current speed of the car, we measured the speed of rotation of the back right wheel. We did this by using an A1101EUA-T Hall Effect sensor to detect magnets attached to the inner side of the wheel. Five magnets were place equidistantly to give a high resolution of speed. We considered adding more but decided that too many "ticks" per revolution of the wheel may be too fast for the interrupt routines of the PSOC to finish.



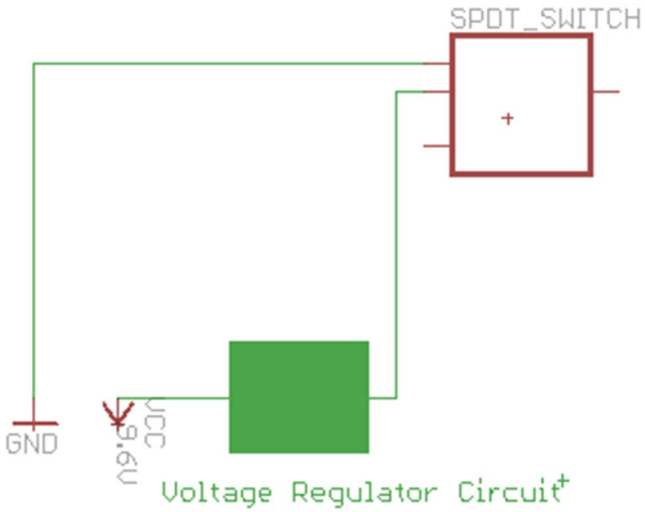
A pull up resistor to 5V was attached to the open collector output of the Hall Effect sensor to prevent a floating signal. This way when the sensor detects the magnet, the data line is drawn low from 5V. We decided on a value of 10KΩ because it needs to be small enough to pull the line high while not being large enough to draw too much current. 10KΩ seems to a good pull-up value for most purposes.

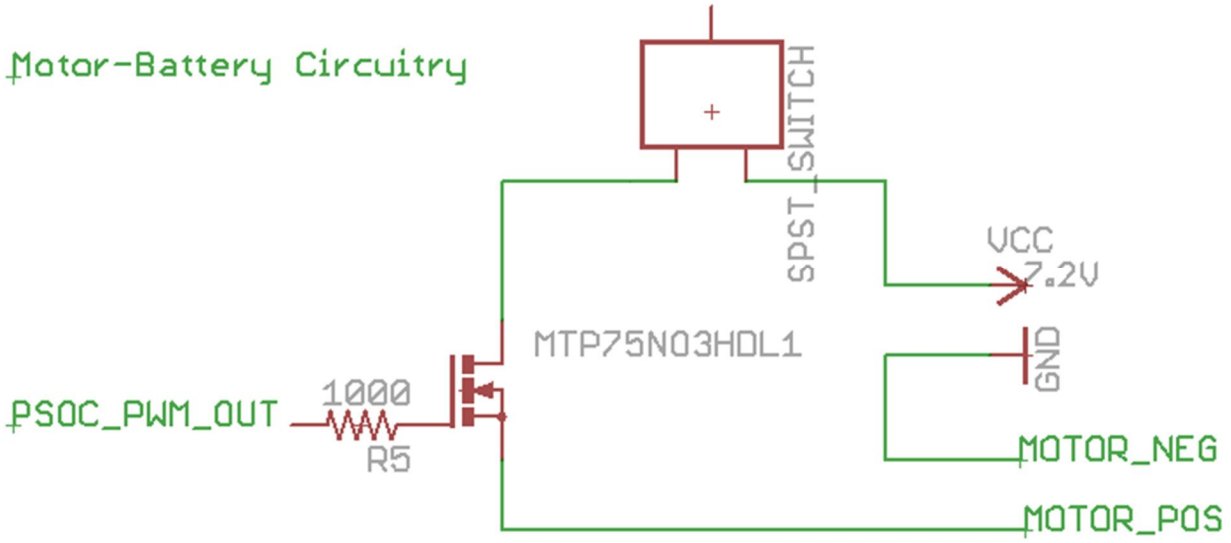
We attached a red LED and 330Ω resistor combo to detect when the 5V was powered on.



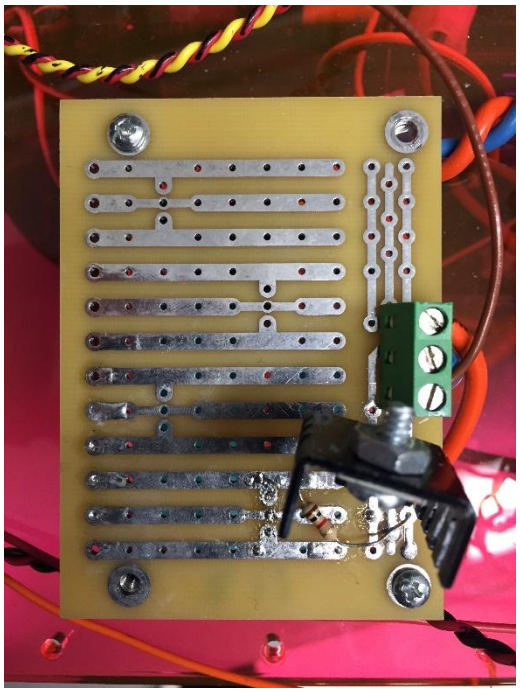
Safety Switches

In order to ensure we can disable our car when it gets out of our control, we installed two safety switches, A SPST switch was used for disabling/enabling the motor and a SPDT switch was used for controlling the voltage regulator, which in turn disables/enables the Hall Effect Sensor, MOSFET, and PSOC.





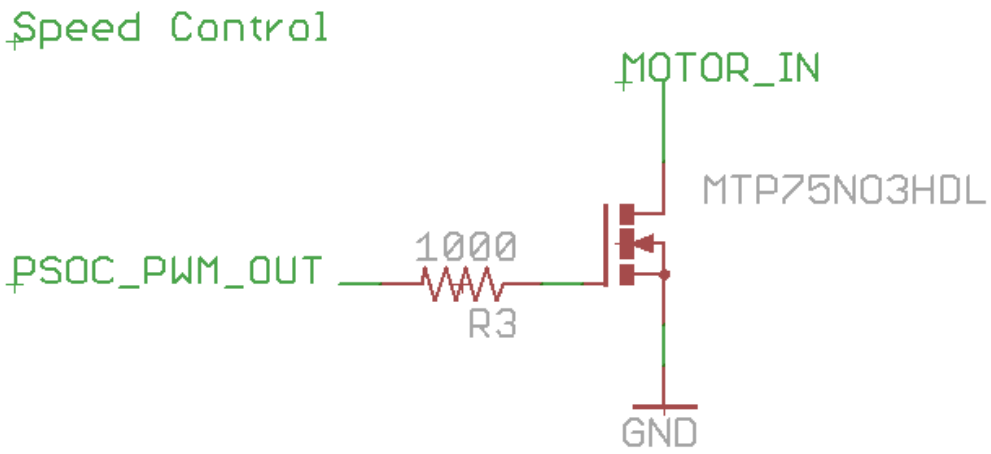
Speed Control Board



Instead of varying the resistance of a motor, which is incredibly inefficient and wastes power, we controlled the motor at variable speeds using pulse width modulation. A higher power MOSFET was used to change the duty cycle of the motor. We chose to use an N-channel MOSFET because when the gate voltage is low in relation to the source, the transistor is off. This is opposed to the P-channel MOSFET where a gate voltage higher than the source causes the transistor to turn off. There might be a situation where we lose control of the gate voltage and it stays low. A PNP MOSFET would stay on and cause max duty cycle, which would lead to potential crashes. We ended up choosing a MTP75NO3HDL power MOSFET because it had a reasonable turn on voltage of around 3.3 V, which is the magnitude of the PSOC output. In addition, using this transistor allowed us to simplify and skip adding a comparator, which any other transistor with higher turn on voltage would have required. Furthermore, this transistor had a power dissipation of 150W, which was among the lowest of the transistors considered and would not heat up as much.

A resistor of 1000Ω was added in series with the gate and PSOC PWM line in order to limit current and prevent the gate voltage from exceeding the supply voltage, which can cause damage to the PSOC driver circuit. We used a low 1000Ω because too high of a resistor will have a large time constant and slow down transistor switching speed.

In addition, 14 gauge wire was used to connect to the motors as they needed to handle larger current load.



PSOC PID Control

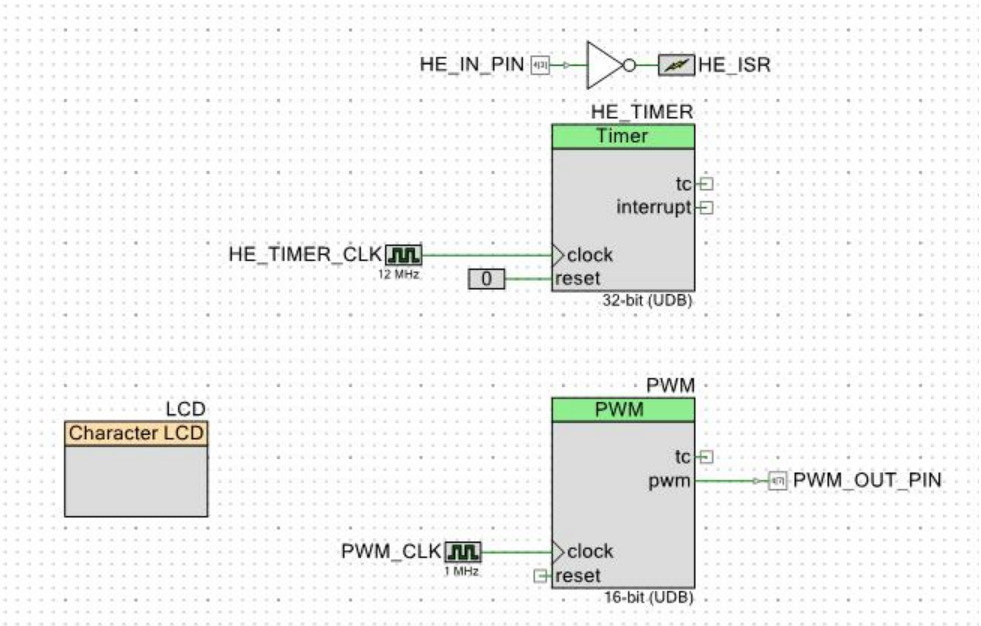
Period: We chose a PWM period of 10ms. We wanted a PWM that would be faster than the car's speed but slow enough for the transistor to switch with the time constant we added with the 1K resistor. We assumed the car shouldn't go above 5ft/sec in any situation, so we converted that time value to around ~25 milliseconds, so 10ms was well under that threshold. The PWM had a high resolution of 16 bits to ensure fine tuning with the duty cycle.

Measuring Speed: We averaged out the speed of an entire wheel rotation since we assumed the magnets were not perfectly spaced apart. Averaging would help iron out small variations in speed.

Duty Cycle Edge Cases: The duty cycle, if negative, was limited to 1 (otherwise, negative numbers produce max duty cycle). And if the duty cycle went over 4000, we cut off the number to 1350 because when the car initially accelerates, it has an unnecessarily high error and speeds up uncontrollably.

Choosing Ki, Kp: We tuned our constants by first using the suggested Ziegler Nichols method. We upped Kp until the car was visibly oscillating around the intended speed. We then cut that by around .6 to get Kp. Since it was difficult for us to determine Tosc, we used a trial & error method with a modified binary search between 0 and Kp. We decided upon a Ki value that corrected for the steady state error and didn't lead to too much oscillation on the flat ground. On the inclined ramp, we had to up Kp and Ki in the same ratios they were on flat ground in order to 1) Provide a higher Ki to help the car weight past errors enough to drive up an incline and 2) maintain a good ratio to avoid tinkering with the flat ground drive. We decided not to go through a Kd as it wasn't necessary to get within the margins. Furthermore, our duty cycle cut off over a certain number, along with discarding initial values, prevented the car from accelerating way over the intended speed in the beginning, effectively substituting as a Kd-like factor.

The following pages are the commented PSOC code and top design.

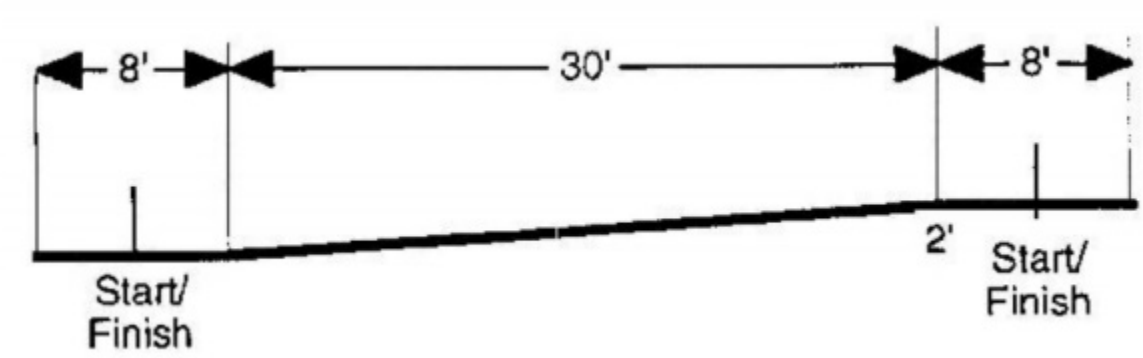


```
#include <project.h>
#include <stdio.h>

#define INCH_PER_MAGNET 1.58
#define SEC_PER_PERIOD 357.914
#define EXPECTED_SPEED 3.05
#define THREE_FT_DUTY 900

#define Kp 5250
#define Ki 0
#define Kd 500

double gprev_HE_count = 0;
int gfirst_HE_read = 1;
int gspeedMeasurements = 0;
double gcurSpeed = 0;
double speedCounts[5];
double gki_error = 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;
}

//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;

    //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 = EXPECTED_SPEED - gcurSpeed;
        //Accumulate past errors for Ki
        gki_error = gki_error+error*time_diff_s;
        // Discard saved error from acceleration as it becomes less relevant
after starting
        if (gspeedMeasurements == 28) gki_error = 0;
        //Calculate the duty cycle based upon Kp, Ki, and Kd
        duty_cycle_buffer = THREE_FT_DUTY + Kp*error + Ki*gki_error +
Kd*error/time_diff_s;

        //LCD output for debugging
        LCD_ClearDisplay();
        LCD_Position(0,0);
        sprintf(buffer, "%f", error);
        LCD_PrintString(buffer);
        LCD_PrintString("//");
        sprintf(buffer, "%f", Kd*error/time_diff_s);
        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;
        duty_cycle = duty_cycle_buffer;
```

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

        PWM_WriteCompare(duty_cycle);
    }
    //debug code
    //PWM_WriteCompare(100);
    //LCD_ClearDisplay();
    //LCD_PrintNumber(gcurSpeed);
    //gcurSpeed++;
}

int main()
{
    //initialize all modules
    CYGlobalIntEnable;
    LCD_Start();
    LCD_Position(0,0);
    LCD_PrintString("ELE302 Carlab");

    HE_TIMER_Start();
    HE_ISR_Start();
    HE_ISR_SetVector(HE_inter);

    PWM_Start();
    PWM_CLK_Start();

    for(;;)
    {
    }
}
/* [] END OF FILE */
```

Results and Further Work



With our PID control implemented, our car was able to traverse a flat course of 36 feet in 12.12 seconds, a 1% error off desired. On uphill our car achieved 12.4s and on downhill 12.01, which were errors of 3.33% and 0.0833% respectively. Ultimately, with just proportional and integral control our car performed better than expected.

Further work can be done by introducing derivate control for a smoother drive. Additionally we noticed that our back wheels were more loose compared to other groups and thus had less friction, making uphill and downhill travel much harder. Once we fix the wheels, we will have to redo our Kp, Ki, and Kd constants, so as it stands, we will maintain the status quo as it is more than sufficient.