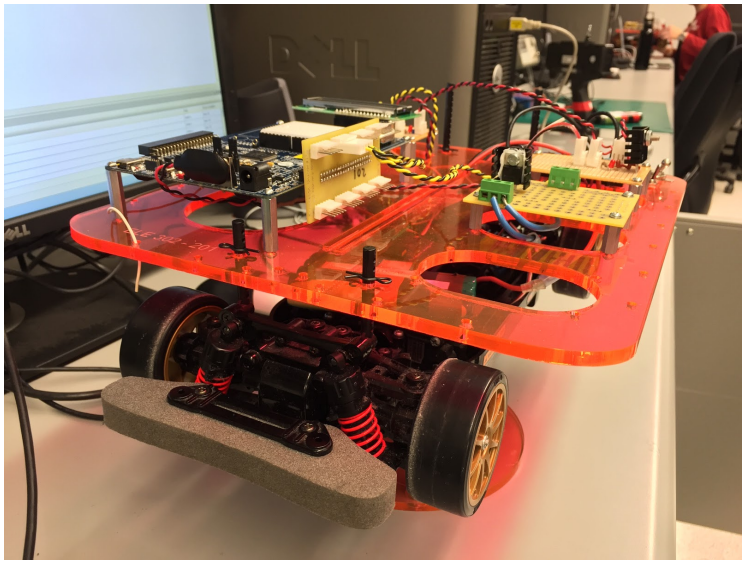
Ryan O'Shea + Liam Kelly
ELE 302
Stage 1: Speed Control
Group 201
March 6, 2015

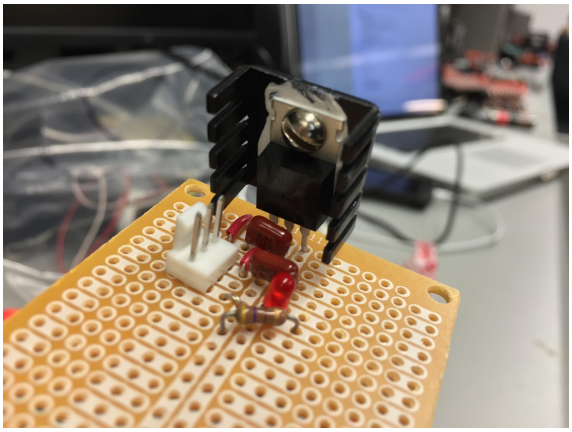
Overview



We implemented a PID speed controller for the hobby car's DC motor. The car was powered by two batteries: 9.6V for electronics and 7.2V for the electric motor. The relevant electronics were:

- A "power board" that used a voltage regulator to bring the 9.6V electronics battery down to a 5V source for all of our electronic components
- An "interface board" that used a Hall effect sensor to detect the motion of one of the car's wheels for the purposes of measuring speed.
- A "MOSFET board" that placed a high-current power-MOSFET in series with the electric motor to allow us to control the amount of power being provided to the motor at any given time by switching the motor on and off via pulse-width modulation (PWM).
- A Cypress Semiconductor PSoC which we programmed to read our car's speed from the interface board and dynamically adjust the motor's output via PWM duty cycle by behaving as a proportional-integral-derivative controller.

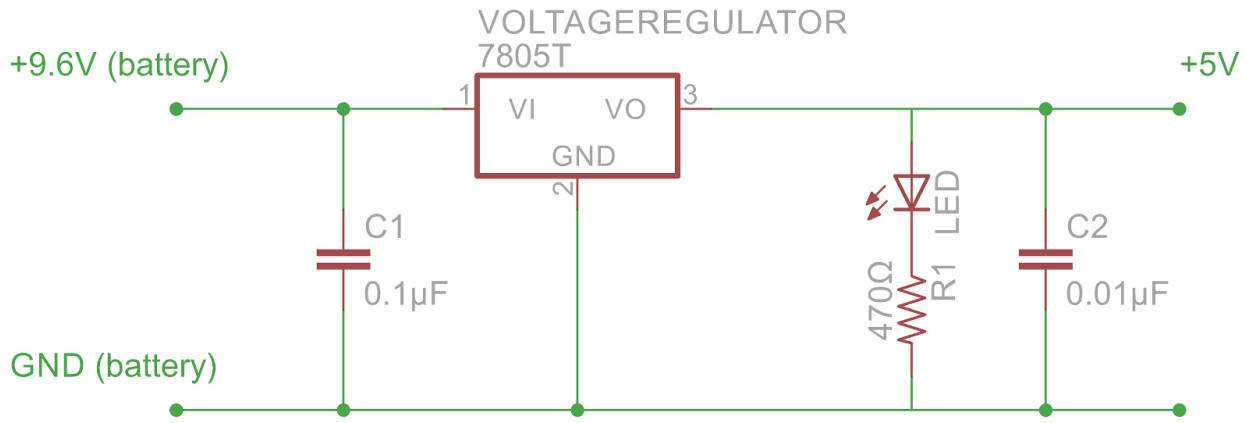
The car also utilized a radio-controlled steering controller for which we supplied power from the power board. This will be replaced in later stages.



Power Board

For the power board (shown incomplete to the left), we have a LM7805 voltage regulator to take in 9.6V from the electronics battery and output 5V. The purpose of this board is to provide power to all the electronics used in our system,

including the Hall effect interface board and steering control and excluding the PSoC. Recognizing that the input voltage signal may be noisy, we added a 0.1μF capacitor between the input of the 7805 and ground, to smooth out the signal the chip was receiving. In addition, to make sure the output was smooth we placed a 0.01μF capacitor to ground. We also placed a 470Ω resistor in series with a red LED to ground on the output to provide a visual cue to show the power board was working. See schematic for a full diagram of the circuit.



This power board is important to regulate 5V going out to all the boards that need it. The 5V signal is used to power the Hall effect sensor, which uses 5V as a reference for signalling whether a magnet is over the sensor itself. The steering control was also powered by 5V from the regulator. The RF control unit would send a signal to be picked up by an antenna, which then used the 5V from the power board to change the direction the wheels were pointing in.

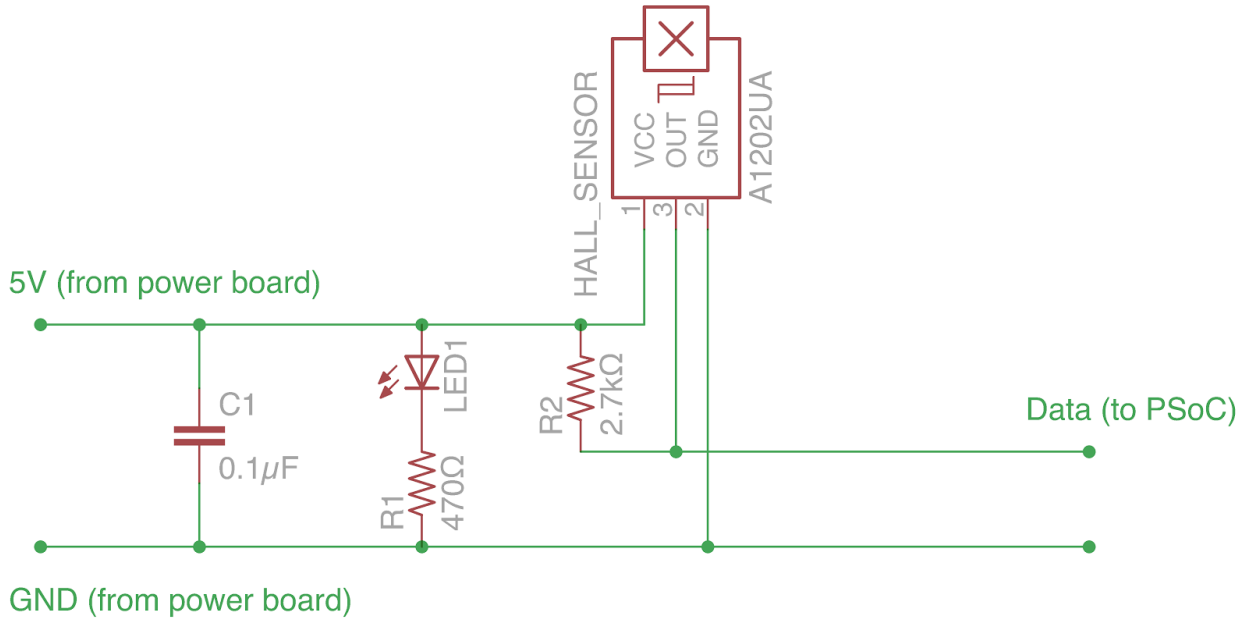
We wired the boards in a star topology, such that no board receives power from any board aside from the power board itself, or from the battery in the case of the PSoC (also, all boards receive ground from either the power board or the battery). In this way, we prevent ground loops caused by noise or other interference to reduce unnecessary power consumption. The Hall sensor interface board is the only instance where power from that board goes off the board, and that is only a short distance to the sensor itself, as described below. In all other instances, power comes directly from the power board or battery.

Additionally, in any case where we transfer power or data to or from a board we use twisted pairs for wires. This reduces noise while travelling through air because of the capacitances between the two wires. We wound the wires tightly in order to create a uniform capacitance between them to reduce the effects of high frequency noise. Any wires between electronics (non-motor and non-sensor) components are 22 gage to be able to allow up to 1A of current. All connections to and from any board are made with KK connectors.

Interface Board

The interface board is the board that receives information from the Hall effect sensor and forwards that information on to the PSoC. This board receives 5V from the power board along with a common ground. To show that power was on the board, we again placed a resistor and LED in series. We also placed a 0.1μF capacitor between the two lines on the board to counteract any noise the voltage may have acquired when being transferred by twisted pair to the board. The 5V and ground are then sent by twisted pair along with a data line to the Hall effect sensor, which is located on the back right wheel. The Hall effect sensor we used was the A1104EUA-T. This sensor displays negative logic, in that it outputs a high impedance value when there is no magnet hovering over it, and a low voltage (0V) when there is. The sensor contains all the logic needed to exhibit this behavior, and we needed to interpret it in the PSoC. To interface with the sensor, we connect 22 gage wire from the board with 30 gage wire that connects directly to the pins of the sensor.

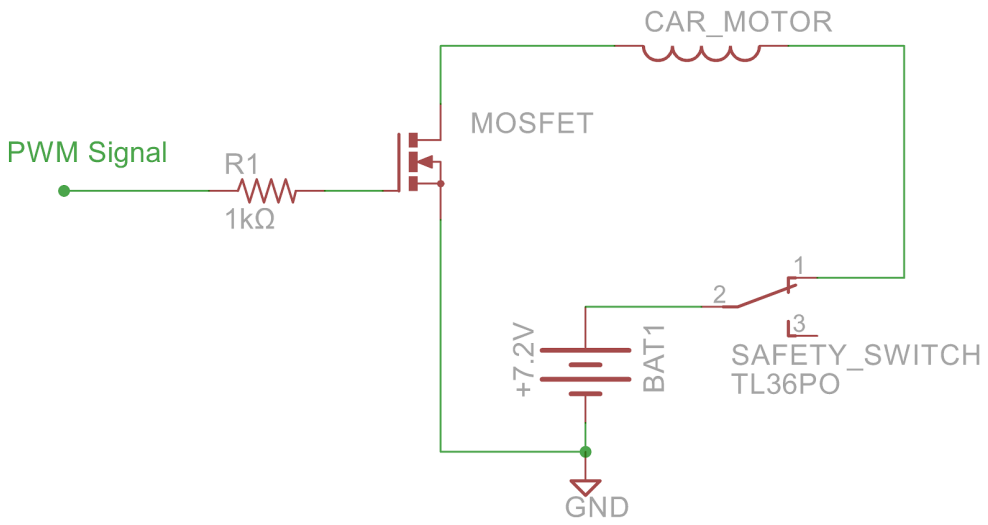
Between where the data line connects from the sensor to the interface board and leaving to go to the PSoC on 22 gage wire, we placed a pull-up resistor so that when there was high impedance output from the sensor, the value sent to the PSoC would be a high voltage (5V) (open collector). We chose a value of 2.7kΩ for this resistor to be high enough that there was not a large power loss when current was flowing but low enough so that the RC constants in the circuit would impact our response times. This results in very clean, well-defined edges, and a signal that is either 0V or 5V. See schematic below for a full diagram of the board. (Note, EAGLE does not have the A1104 sensor, so we used the A1202 in the diagram, which it did have, and has the same pinout).



MOSFET Board

The MOSFET board is used to power the motor on and off using the PWM signal controlled by the PSoC. We used a power-MOSFET (MTP75NO3HDL) to handle the switching because of its low turn-on (gate threshold) voltage (1-2V) relative to the output of the PSoC's digital pins (3.3V) and its high current rating (75A). The MOSFET was mounted to a heatsink designed for its TO-220 package to help with heat dissipation during rapid PWM switching. Due to the high current possible, 14 gage wire is used for the circuit connecting the motor to power and the MOSFET.

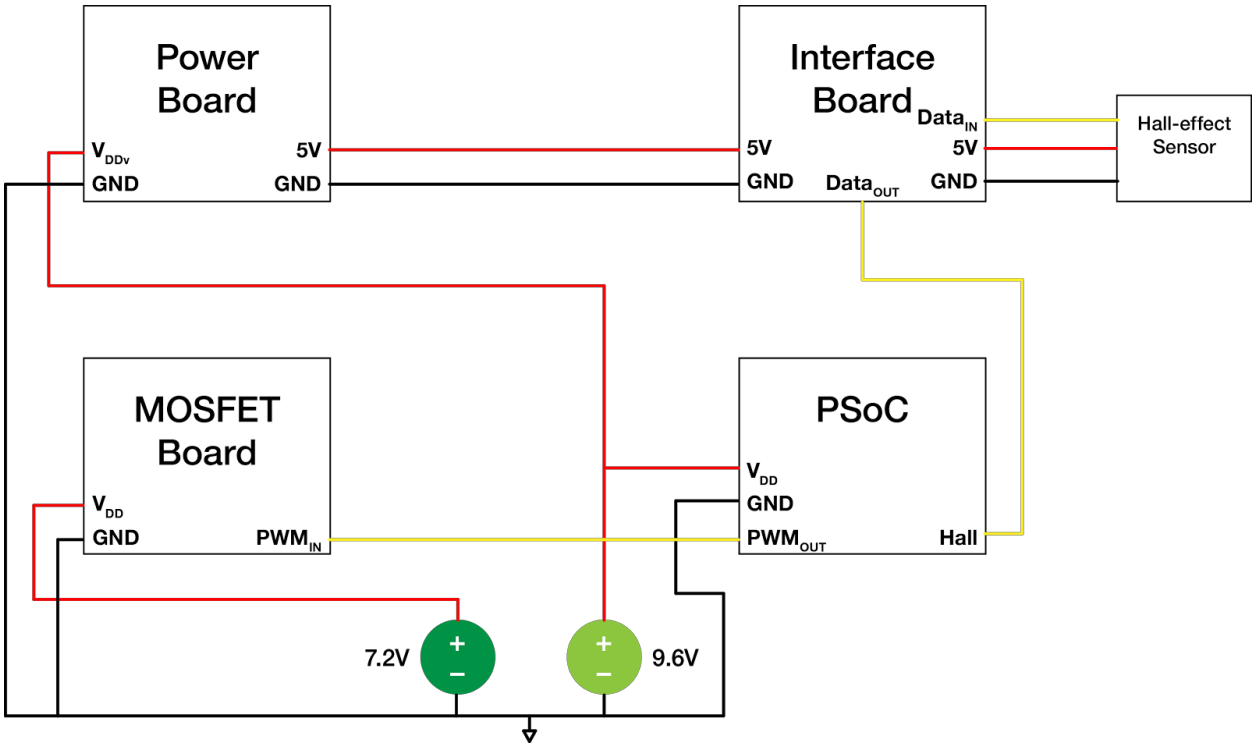
In the schematic, the battery is our 7.2V motor battery. The resistor between the PSoC's PWM output and the MOSFET's gate serves to protect against significant current draw between the PSoC and the motor, which are connected by 22 gage wire. The 1kΩ value allows for relatively fast switching (low time constant) while keeping the current draw fairly low, so as to not damage the PSoC and waste battery power. In the circuit, the safety switch is our emergency circuit interrupt, but when it is closed (as shown in the schematic), the circuit is opened and closed by the PWM signal controlling the gate of our MOSFET. The motor is connected to the drain of our MOSFET to prevent back emf from interfering with the value of our gate-source voltage and thus impeding our ability to reliably control the motor. We used an n-channel MOSFET for faster switching (due to lower propagation delay and lower R_{DS}).



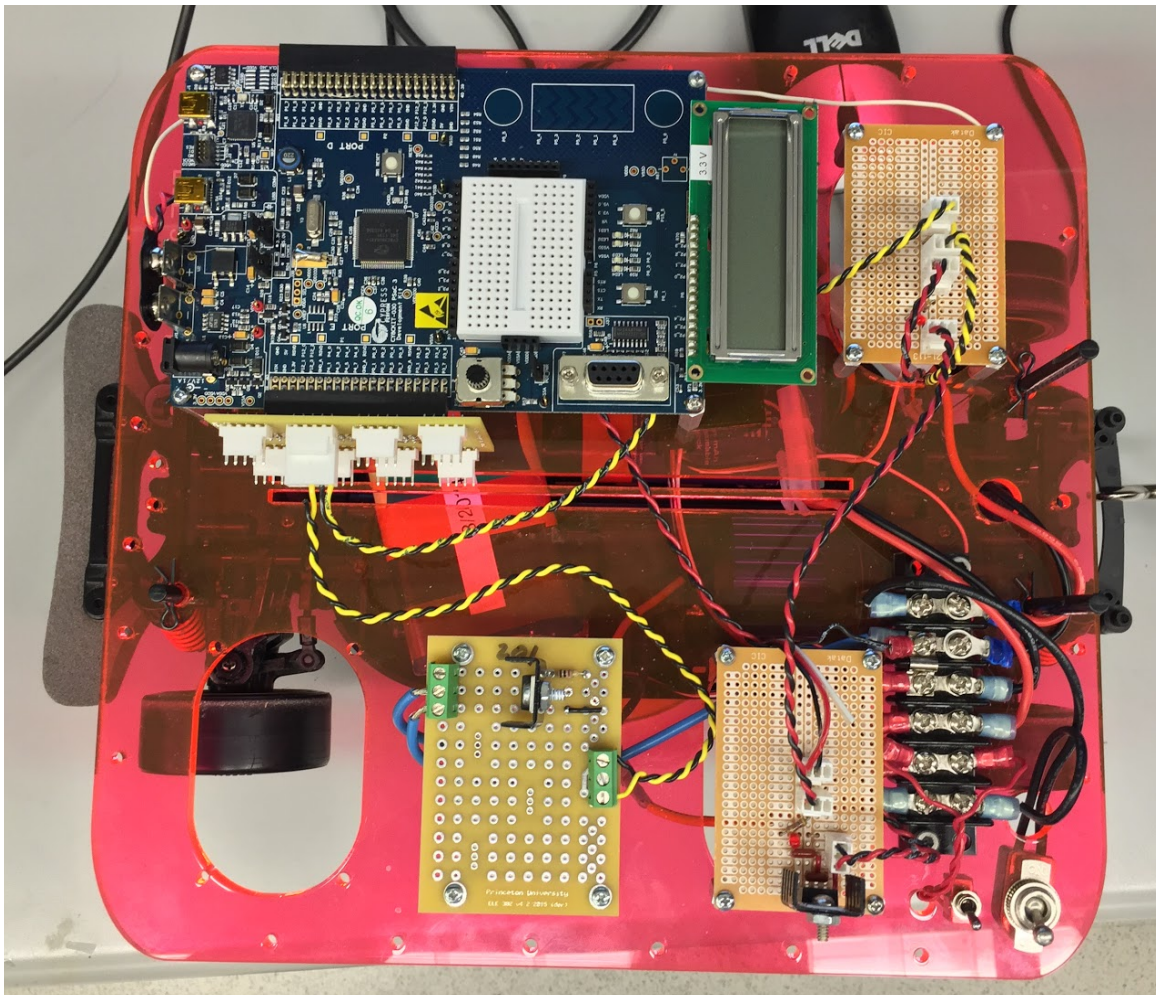
Note: please ignore the part number on the switch.

Topology

Below, we show how the boards mentioned above were connected together with the batteries. The radio steering control is not shown, but simply draws 5V and GND from the power board in parallel with the interface board.



The boards were physically laid out on the car chassis as shown below:



Clockwise from top left: PSoC, Interface Board, terminal block for power connections, Power Board, MOSFET Board

PID Control with the PSoC

To drive the car at a constant speed of 3 feet per second, we first read from the Hall sensor to calculate the speed the car is going at. This is done in the PSoC by creating an array of size 5 that stores the most recent time a magnet has passed over the sensor. The size of 5 was chosen because there are 5 magnets on the wheel, so we know that one full revolution of the wheel will correspond to 5 "ticks" from the sensor. Each tick fires an interrupt that results in a recording of a special counter clocked at 1kHz. We then take the difference between the previous value in the array for the current magnet and the current value, and use that to signify the time that has elapsed for one full revolution. We have an index into the array that we increment (or loop around) each time a tick is seen, so we know we are looking

at the same magnet for the respective tick. Given the time for a revolution, we calculate a speed by dividing the circumference of the wheel by that time difference.

We have an interrupt every 100ms to ensure the car is not stalling, as well as to actually update the PWM duty cycle. We only update the duty cycle when this interrupt occurs so that we do not consume too much CPU time updating the PWM duty cycle every time a magnet crosses the Hall effect sensor and so that our control is updated regularly, independent of the speed of the car. This also helps to ensure that if our car is not moving the PWM duty cycle will increase more and more (due to integral control) to cause the car to begin moving when it is stuck.

Once the speed is calculated, we subtract it from the desired speed (3 feet/s) to get an error value. This is then fed through a PID controller, which in turn changes the duty cycle on the PWM signal to drive the motor. In terms of PID control itself, proportional control gets us close to the desired speed, although it alone can cause some steady state error. Integral control gets rid of steady state error, but introduces some overshoot and can cause oscillation. Derivative control gets rid of oscillations, but can also reduce rise time. We first obtained a steady state approximation for what value our PWM duty cycle should be with an open loop system. We then found a proportional constant based on the Ziegler-Nichols method, and from that method we also derived integral and derivative constants. After getting these approximations, we tweaked them until the speed reacted as we wanted it to given the specifications. We generally found that the integral component was most important in achieving rapid adaptation to changes in our car's output speed.

The code used for our car is included in Appendix A. The top design schematic is included in Appendix B.

Ultimately, our car was able to maintain 3 feet per second with this PID control on both flat ground (within 2% tolerance) and slanted ground (within 10% tolerance).

Appendix A: Controller Code

```
#include <device.h>
#include <stdio.h>
uint32 MAX_TIME = 65535; //Counter period
uint32 FREQ = 100; //Hz
uint32 lastTimeSeen[5];
float CIRCUMFERENCE = 0.66;
float SETPOINT = 3.0;
int currentMagnet;
float speed;

float epsilon = 0.01;
float Kp = 40;
float Ki = 4.5;
float Kd = .05;
float MAX_OUT;
float MIN_OUT = 0;
float preError;
float integral;
float openloop;

float PIDCtrl(float wantedSpeed, float actualSpeed, float dt)
{
    float error = wantedSpeed - actualSpeed;
    float derivative;
    float output;
    char strbuffer[15];

    // Only integrate if error large enough
    float abserror;
    /*sprintf(strbuffer, "%.4f", error);
    LCD_Position(0,0);
    LCD_PrintString(strbuffer);*/
    if (error >= 0) abserror = error;
    else abserror = -1*error;
    if (abserror > epsilon)
    {
        integral += error;
    }
    sprintf(strbuffer, "%.4f %.4f", actualSpeed, integral);
    LCD_Position(0,0);
    LCD_PrintString(strbuffer);
    derivative = error - preError;
    output = openloop + Kp*error + Ki*integral + Kd*derivative;

    // Saturation filter
    if (output > MAX_OUT)
    {
        output = MAX_OUT;
    }
    if (output < MIN_OUT)
    {
```

```
        output = MIN_OUT;
    }

    preError = error;

    return output;
}

CY_ISR(magnet)
{
    uint32 nextTime = SpeedTimer_ReadCounter();
    uint32 prevTime = lastTimeSeen[currentMagnet];

    char strbuffer[15];
    float pidOutput;
    float dt = (prevTime - nextTime)/(float)FREQ;
    lastTimeSeen[currentMagnet] = nextTime;
    if (nextTime > prevTime)
    {
        nextTime = nextTime - MAX_TIME;
    }

    speed = (CIRCUMFERENCE / dt);
    /*sprintf(strbuffer, "%.4f", speed);
    LCD_Position(0,0);
    LCD_PrintString(strbuffer);
    /*LCD_Position(1,0);
    sprintf(strbuffer, "%lu - %lu = %lu", prevTime, nextTime, prevTime - nextTime);
    LCD_PrintString(strbuffer);*/
    lastTimeSeen[currentMagnet] = nextTime;
    currentMagnet++;
    if (currentMagnet == 5)
    {
        currentMagnet = 0;
    }
}

CY_ISR(stall)
{
    uint32 currentTime = SpeedTimer_ReadCounter();
    uint32 idx;
    uint32 oldTime;
    char strbuffer[15];
    float pidOutput;
    if (currentMagnet == 0) idx = 5;
    else idx = currentMagnet;
    oldTime = lastTimeSeen[idx - 1];

    if ((oldTime - currentTime)/FREQ > 0.25) // Stalled
    {
        pidOutput = PIDCtrl(SETPOINT, (float)0, .1);
        sprintf(strbuffer, "PID: %u ", (uint16) pidOutput);
        LCD_Position(1,0);
        LCD_PrintString(strbuffer);
```

```
        PWM_WriteCompare((uint16) pidOutput);
    }
    else // Normal PWM update
    {
        pidOutput = PIDCtrl(SETPOINT, speed, .1);
        sprintf(strbuffer, "PID: %u ", (uint16) pidOutput);
        LCD_Position(1,0);
        LCD_PrintString(strbuffer);
        PWM_WriteCompare((uint16) pidOutput);
    }
}


void main()
{
    /* Place your initialization/startup code here (e.g. MyInst_Start()) */
    int i;
    float pidOutput;
    char strbuffer[15];

    LCD_Start();
    PWM_Start();
    SpeedTimer_Start();
    StallTimer_Start();
    CyGlobalIntEnable;
    inter_hall_Start();
    inter_hall_SetVector(magnet);
    inter_stall_Start();
    inter_stall_SetVector(stall);

    speed = 0;
    currentMagnet = 0;
    preError = 0.0;
    integral = 0.0;
    MAX_OUT = PWM_ReadPeriod()+1;
    openloop = 27*SETPOINT;

    /* initialize the array */
    for (i = 0; i < 5; i++)
    {
        lastTimeSeen[i] = MAX_TIME;
    }

    // Calculate the pid output
    pidOutput = PIDCtrl(SETPOINT, 0, .1);
    sprintf(strbuffer, "%i    ", (int) pidOutput);
    LCD_Position(1,0);
    LCD_PrintString(strbuffer);

    // Change the PWM to be the pid output
    PWM_WriteCompare((uint8) pidOutput);
    /* CyGlobalIntEnable; */ /* Uncomment this line to enable global interrupts. */
    for(;;)
    {
```

```
        /* Place your application code here. */
    }
}

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

Appendix B: PSoC Top Design Schematic