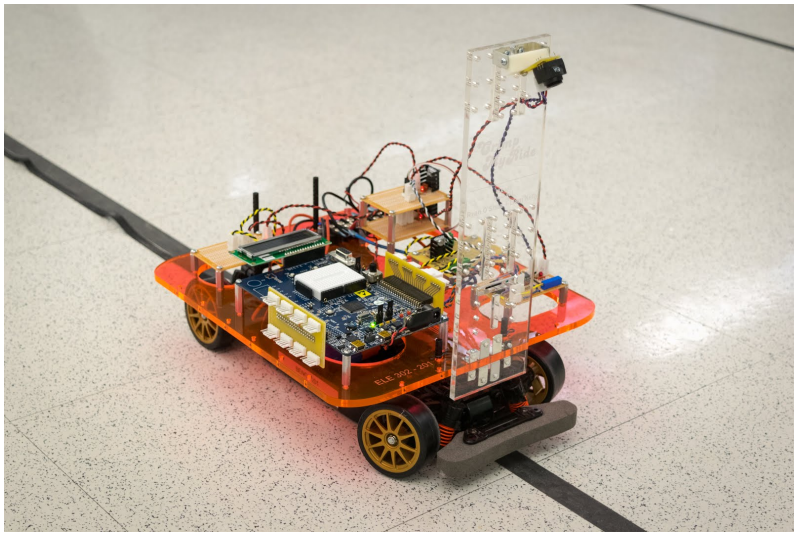
Ryan O'Shea + Liam Kelly
ELE 302
Stage 2: Steering Control
Group 201
April 3, 2015

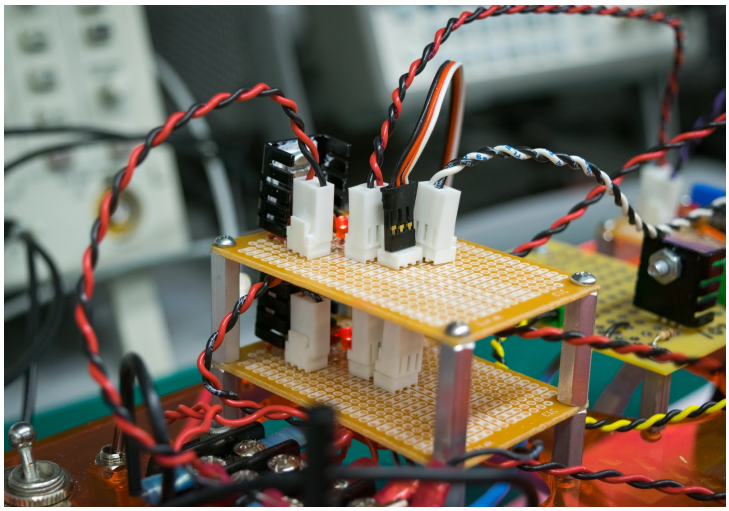
Overview



We took our speed-controlled car and added hardware and software to allow it to optically find and follow a black track on the floor by dynamically controlling the steering. The relevant additions to the car beyond stage 1 were:

- A small camera mounted on a laser-etched acrylic mast, which we used to detect the black line the car was to follow.
- A 'camera board,' which uses a video sync separator and comparator to provide us with more accessible signals to interpret what our camera is seeing. These signals connect to the PSoC.
- A second 'power board,' which is identical to the power board in our first write-up in that it uses a voltage regulator to reduce our 9.6V electronics battery to a 5V power source for electronics on our car except for the PSoC. This second board was added to reduce current load on the first power board.
- The car's steering power and control lines are now being controlled by our hardware. The steering control line is attached to and controlled by the PSoC, and the steering servo motor is powered by our second power board.

Power Board



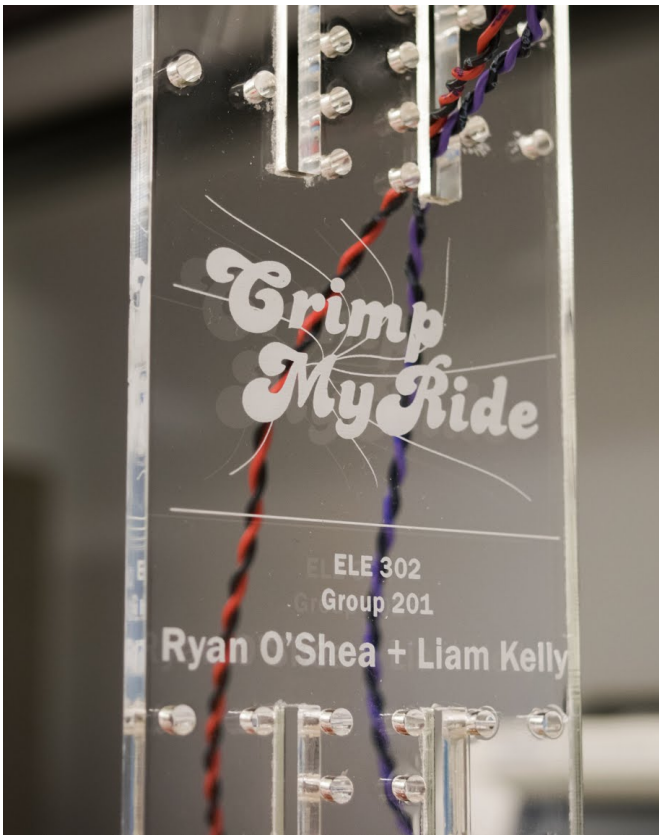
(Right: the two power boards, mounted on top of each other)

As mentioned, we added a second power board to facilitate the powering of more elements. Our wiring changed somewhat from the first stage. In the new design, the Hall-effect sensor (interface) board

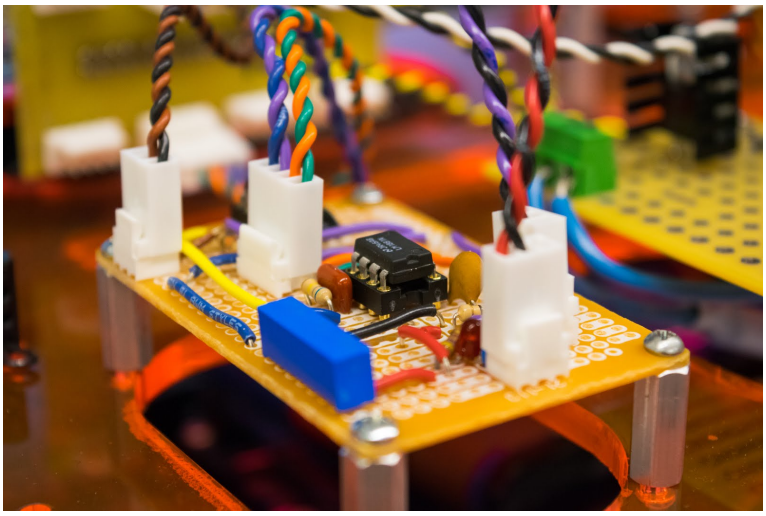
and camera are powered by the original power board, while the steering servo and camera board are powered by the second power board. The arrangement was selected in order to minimize the effect of noise generated by the steering system. We chose to power the camera chip, rather than either of the sensors, from the same board as the steering because we thought the camera chip would be more resilient to noise.

Camera and Mast

We laser-cut an acrylic mast and attached it perpendicularly to the front of our car's acrylic body. On that, we mounted our camera, aimed at the ground in front of the car. Below, we show the mast diagram and the final etched mast:



Camera Board

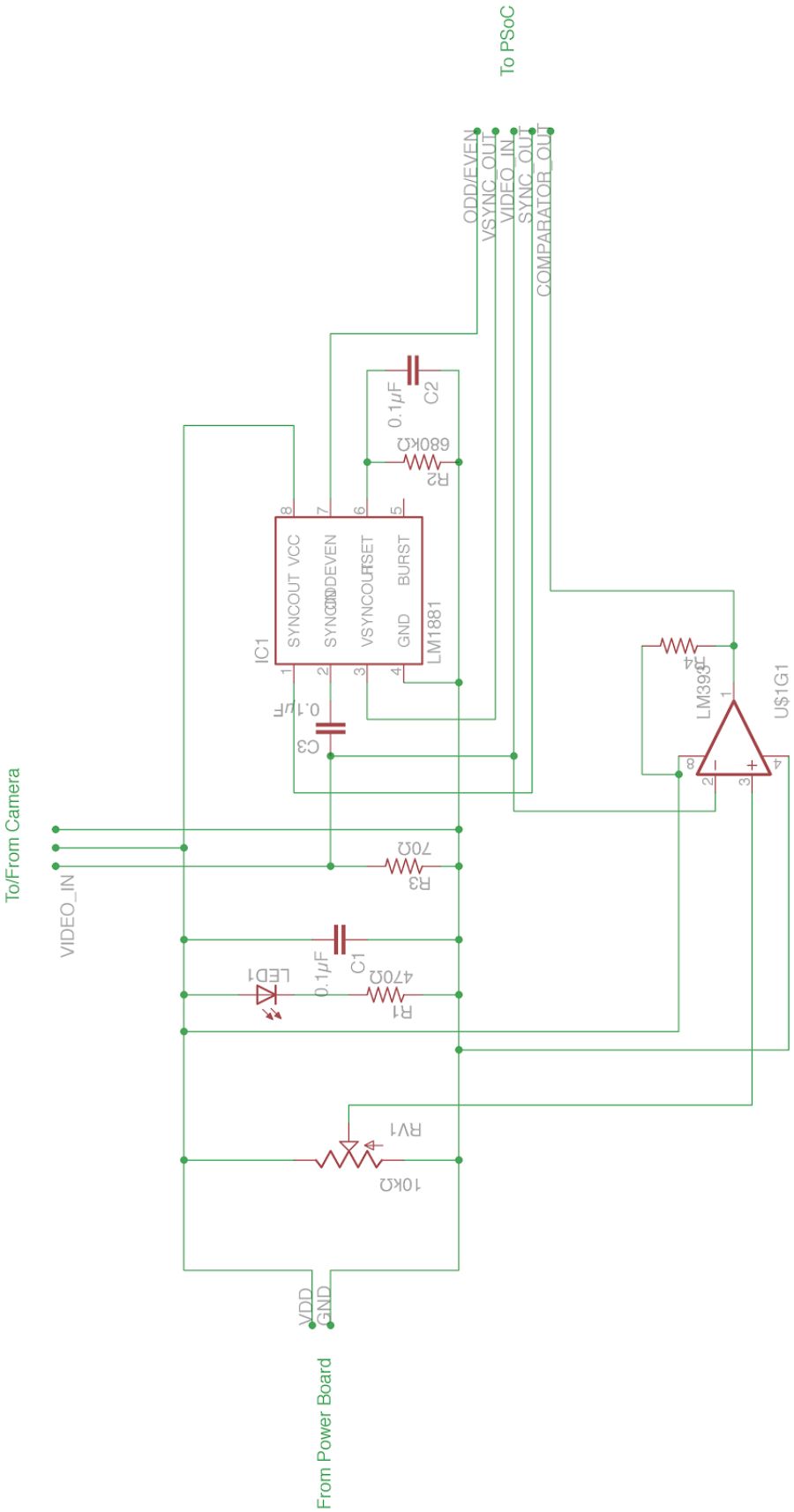


(Right: the camera board. The visible chip is the video sync separator, while the comparator is hidden behind the middle KK connector)

The camera board consists of two inbound sets of wires from the power board and from the camera, as well as a number of outbound wires to the PSoC. From the power board, the camera board receives its VDD and GND signals with a 0.1μF de-spiking capacitor as well as an LED in series with a resistor to show when the board is receiving power. The analog video signal is sent to the LM1881 chip, which separates the video signal into composite sync, vertical sync, odd/even, and burst components. As per the datasheet of the camera, we place a 70Ω resistor to GND from the video signal, and per the LM1881 datasheet we gate the composite input of the chip with a 0.1μF capacitor. We take the odd/even, v-sync, and composite sync outputs of the chip and send those to the PSoC, along with the video signal.

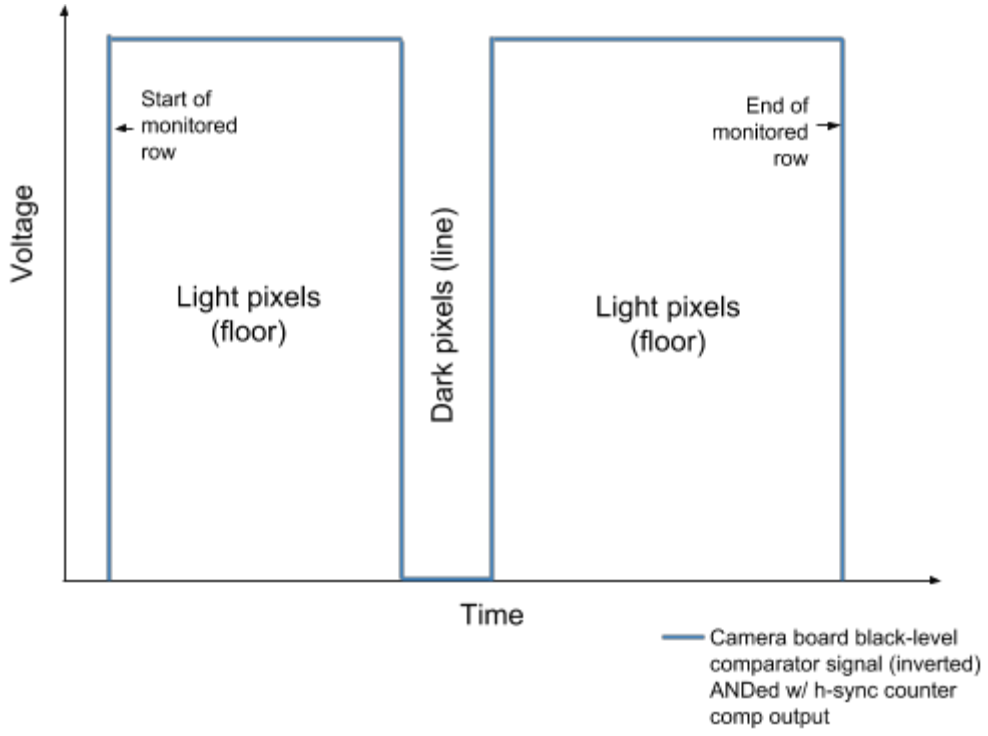
We send one other signal to the PSoC. This is the output of a comparator we installed on the camera board, which indicates whether we are currently looking at a black or non-black pixel. We used a potentiometer to get a value of roughly 540mV, which we used as our reference level for a black pixel. Anything below that level is considered black and anything above it is considered non-black. We placed the video input on the inverting input so that the output would go high (via pull-up resistor) when we were at the edge of a black line. This was a design decision so we could have a rising edge at the start of a black line from left to right. This comparator output was then sent to the PSoC to be used as a digital signal to assist in tracking the black line, as described below.

(Below: EAGLE schematic of the camera board)


To PSoC
ODD/EVEN
VSYNC_OUT
VIDEO_IN
SYNC_OUT
COMPARATOR_OUT
0.1µF
C2
680KΩ
R2
IC1
SYNCOUT
VCC
ODD/EVEN
VSYNCOUT
RSET
BURST
GND
LM1881
0.1µF
C3
LM393
U$1G1
To/From Camera
VIDEO_IN
70Ω
R3
0.1µF
C1
LED1
470Ω
R1
RV1
10KΩ
VDD
GND
From Power Board

Line Detection

Our PSoC was programmed to detect a black line to follow on the floor in the following way. Every 2 frames (meaning every one real frame, since frames are interpolated and sent half a frame per vertical sync), the PSoC reads the analog brightness data for a single row of pixels near the bottom of the image. This is done by via a vertical sync counter, which, after every second vertical sync, resets a horizontal sync counter, which has a comparator output that goes high after seeing 160 horizontal syncs in the current frame. The horizontal sync counter's comparator output is ANDed with an inverted version of the hardware comparator on our camera board. The result of this AND is a signal that stays low while the camera is sending data about the rows of pixels we aren't monitoring, but is free to change while the horizontal sync counter's comparator value is high, i.e. while we are seeing data from the row of pixels we want to monitor. Then, the output of the AND is exactly the inversion of the hardware comparator on the camera board: moving left to right across the image row, it stays high while the pixels in the monitored row are light, drops down when the pixels are dark (the line), rises again when the pixels are bright, and finally falls when the row we are monitoring ends. The signal looks like the figure below:



To detect the location of the line within the row, we use the capture feature of a timer element on the PSoC to measure the 'distance' into the row each of the two rising and two falling edges of the ANDed comparator signal, which correspond to the start of the row, and the relative horizontal positions of the left side of the black line, the right side of the black

line, and the end of the row. These positions are represented as timer values. We calculate the horizontal location of the black line in the frame by taking the midpoint of the line's left and right edges and then calculating its position as a percentage of the full width of the frame (i.e. the time between the start of the row and the end of the row). We represent the line's location as this percentage, an integer from 0 to 99. If we didn't see the interrupts we expected to (i.e. interference kept us from seeing the line clearly, or the line isn't in frame), we ignore the reading and start again in the next frame.

Steering Control

To do our steering control, we implemented another PID controller similar to the one we used for speed control. In this case, our goal was to get the center of the line at what we determined to be 54% of the way across the frame (left to right) to be going straight. When going straight, the output of our steering pin is a 1.52ms square pulse at 100Hz. From the position of the line in the frame, calculated as described above, we calculate an error off the center of the line and use PID control to correct that error. We chose our values such that the car exhibits little to no noticeable oscillation and can follow the curves of the track for a full two laps before coming to a full stop. To come to a full stop, we decided to use a timer. Based on our speed, we figured out how long the car should run to complete two circuits of the track. At the end of those two laps, we shut down the motor PWM signal and end any interrupts from the magnets on the wheel.

All code for both navigation and speed control is contained in Appendix A, below.

Appendix A: Code

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

/* Variables and Constants for Speed Control */
uint32 MAX_TIME = 65535; //Counter period
uint32 FREQ = 100; //Hz
uint32 lastTimeSeen[5];
float CIRCUMFERENCE = 0.66;
float SETPOINT = 3.5;
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;

    // Only integrate if error large enough
    float abserror;
    if (error >= 0) abserror = error;
    else abserror = -1*error;
    if (abserror > epsilon)
        integral += error;
    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];

    float dt = (prevTime - nextTime)/(float)FREQ;

    lastTimeSeen[currentMagnet] = nextTime;
    if (nextTime > prevTime)
    {
        nextTime = nextTime - MAX_TIME;
    }

    speed = (CIRCUMFERENCE / dt);
    lastTimeSeen[currentMagnet] = nextTime;
    currentMagnet++;
    if (currentMagnet == 5)
    {
        currentMagnet = 0;
    }
}

CY_ISR(stall)
{
    uint32 currentTime = SpeedTimer_ReadCounter();
    uint32 idx;
    uint32 oldTime;
    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);
        Motor_PWM_WriteCompare((uint16) pidOutput);
    }
    else // Normal PWM update
    {
        pidOutput = PIDCtrl(SETPOINT, speed, .1);
        Motor_PWM_WriteCompare((uint16) pidOutput);
    }
```

```
}

/* Variables and Constants for Steering Control */
float STEER_MIN_OUT = 110;
float STEER_MAX_OUT = 190;
float Steer_epsilon;
float Steer_preError;
float Steer_integral;
float Kp_Steer = 1.5;
float Ki_Steer = .005;
float Kd_Steer = .005;
float openloop_Steer = 152;
float Steer_SETPOINT = 54;
uint32 MAX_TRACK_TIME = 4294967295;

float Steering_PID(float linePercentage, float wantedPercentage)
{
    float Steer_error = linePercentage - wantedPercentage;
    float Steer_derivative;
    float Steer_output;

    // Only integrate if error large enough
    float Steer_abserror;
    if (Steer_error >= 0) Steer_abserror = Steer_error;
    else Steer_abserror = -1*Steer_error;
    if (Steer_abserror > Steer_epsilon)
        Steer_integral += Steer_error;
    Steer_derivative = Steer_error - Steer_preError;
    Steer_output = openloop_Steer + Kp_Steer*Steer_error + Ki_Steer*Steer_integral +
Kd_Steer*Steer_derivative;

    // Saturation filter
    if (Steer_output > STEER_MAX_OUT)
    {
        Steer_output = STEER_MAX_OUT;
    }
    if (Steer_output < STEER_MIN_OUT)
    {
        Steer_output = STEER_MIN_OUT;
    }

    Steer_preError = Steer_error;

    return Steer_output;
}

CY_ISR(lineSpotted)
{
```

```
    uint32 lineLeft = Line_Timer_ReadCapture();
    uint32 blackLeft = Line_Timer_ReadCapture();
    uint32 blackRight = Line_Timer_ReadCapture();
    uint32 lineRight = Line_Timer_ReadCapture();

    uint32 blackPos = lineLeft - ((blackLeft + blackRight) / 2);

    uint32 lineWidth = lineLeft - lineRight;

    uint32 percentage = (blackPos * 100) / lineWidth;

    float Steer_PIDout;
    char strbuffer[15];

    if (!(percentage > 100 || percentage < 0))
        Line_Timer_ClearFIFO();

    sprintf(strbuffer, "%lu   ", (MAX_TIME - (uint16) TrackTimer_ReadCounter()));
    LCD_Position(1,0);
    LCD_PrintString(strbuffer);

    if (((MAX_TIME - (uint16) TrackTimer_ReadCounter()) > 540) && ((MAX_TIME - (uint16)
TrackTimer_ReadCounter()) < 60000))
    {
        inter_hall_Stop();
        inter_stall_Stop();
        Motor_PWM_WriteCompare((uint16) 0);
    }

    Steer_PIDout = Steering_PID((float) percentage, Steer_SETPOINT);

    Steering_PWM_WriteCompare((uint16) Steer_PIDout);
}


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

    LCD_Start();

    /* Motor initializing */
    Motor_PWM_Start();
    SpeedTimer_Start();
    StallTimer_Start();
```

```
    CyGlobalIntEnable;
    inter_hall_Start();
    inter_hall_SetVector(magnet);
    inter_stall_Start();
    inter_stall_SetVector(stall);

    /* Steering initializing */
    Steering_PWM_Start();
    VSync_Counter_Start();
    HSync_Counter_Start();
    Line_Timer_Start();
    TrackTimer_Start();
    inter_lineSpotted_Start();
    inter_lineSpotted_SetVector(lineSpotted);

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

    Steer_preError = 0.0;
    Steer_integral = 0.0;

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

    // Calculate the speed pid output
    pidOutput = PIDCtrl(SETPOINT, 0, .1);

    // Change the PWM to be the pid output
    Motor_PWM_WriteCompare((uint8) pidOutput);
    Steering_PWM_WriteCompare((uint8) 152);
    /* CyGlobalIntEnable; */ /* Uncomment this line to enable global interrupts. */
    for(;;)
    {
        /* Place your application code here. */
    }
}

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