🚗 ECE 302 Navigation Report

1. Information

- **Bench Number: 409**
- **Name(s)**: David Lee (dl1506@princeton.edu), Seungho (John) Lee (jl7339@princeton.edu)
- **Date**: 11/9/2025

2. Statement of Objectives

1. Laser cut a mount for the camera to be attached to the car and implement the video board to process information sent by the camera for navigation.
2. Using PID control, maintain navigation control to complete two laps around a track of black tape without human intervention.

3. Overview of Key Subsystems and Components

[Used the same car as speed control]

- Video Board: Powered by 5.0 V, the video sync separator (LM1881) takes in the composite video input from the camera and outputs a horizontal sync and vertical sync that is sent to the PSoC along with the original video input.
- Camera + Mount: Placed in the front of the car, used to identify the black lines in front of the car for navigation. The camera is powered by 5.0 V and the signal from the camera is sent to the video sync separator.

4. Schematic Diagram of Circuits

*Note: Circuits were hand drawn and labeled because we no longer have access to CircuitLabs

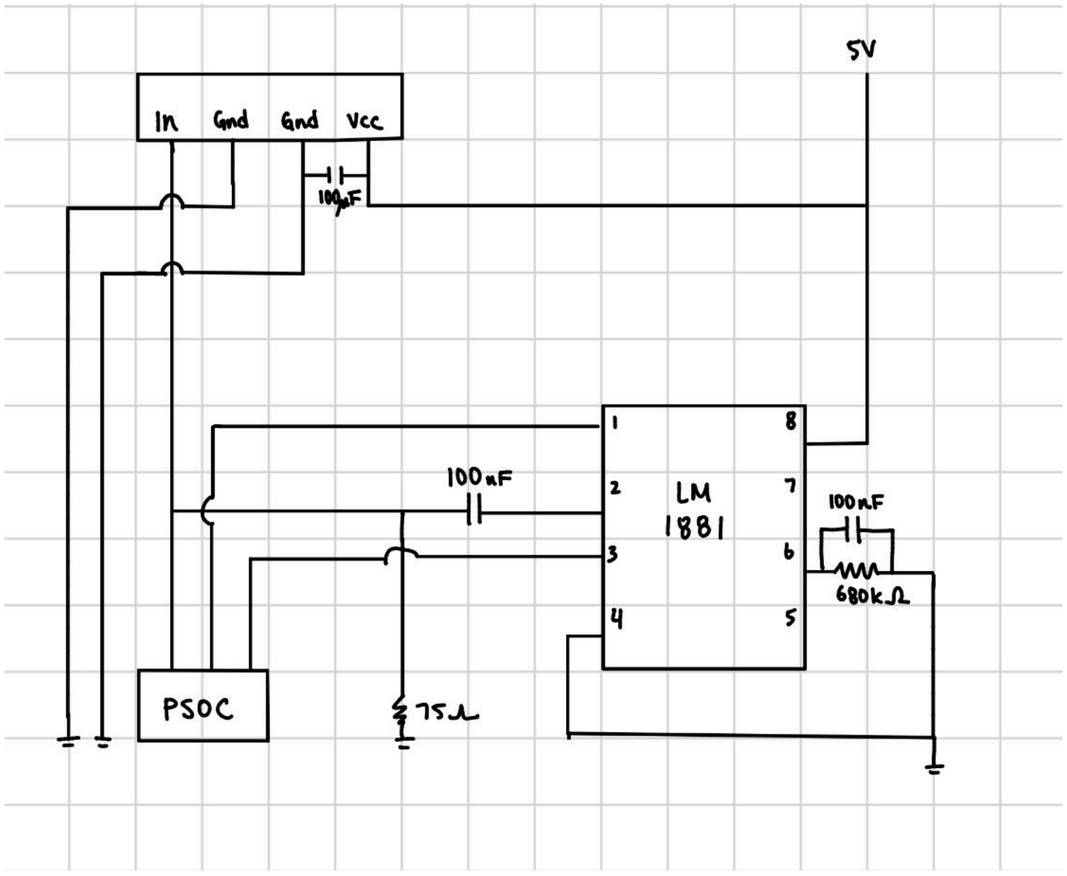


Figure 1. Circuit diagram of the video board

5. Data and Results

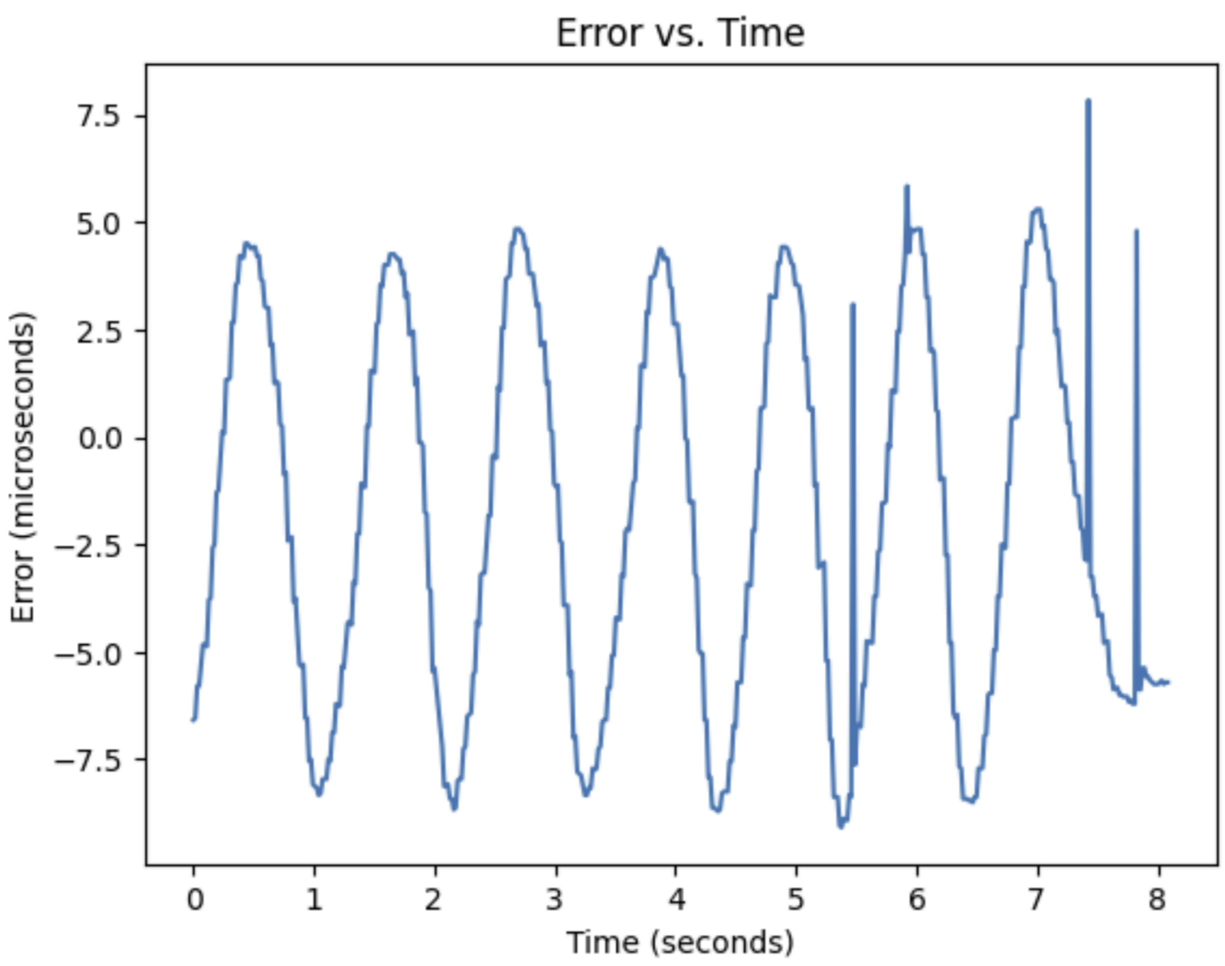


Figure 2. Graph of error versus time when going along a straight path. The timer received a clock frequency of 24 MHz so 1 second is equivalent to 24 million clock cycles. The graph fluctuates in a sinusoidal waveform as the car swerves left and right by a small amount due to the proportional and derivative controls.

Raw Video(Yellow) vs Composite Sync (Blue)

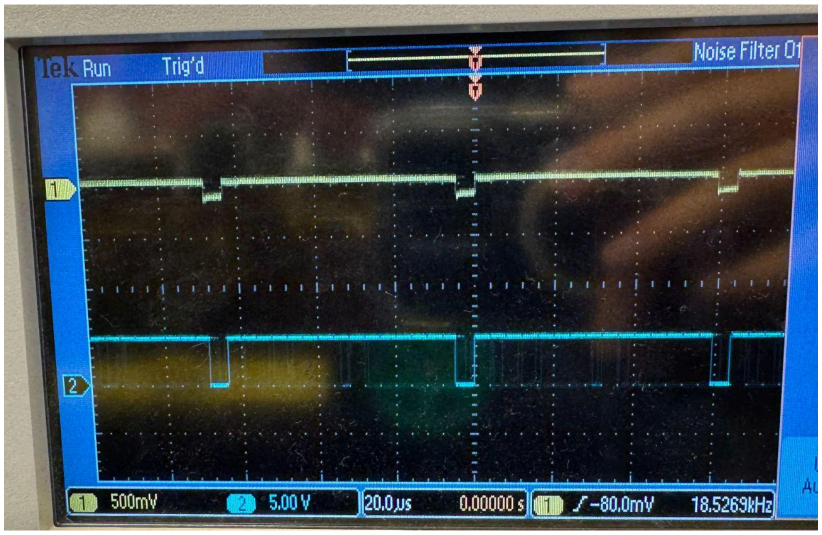


Figure 3. The scope traces showing raw video and composite sync. The yellow trace is the full video waveform (the camera was not in the presence of a black line). The composite sync pulses are indicated by the dips which go below the black level all the way down to 0 V. The blue trace is the composite sync output, which is simply a reproduction of the signal waveform below the composite video black level, with the video completely removed. Therefore, we should see the corresponding low pulses and a clean signal, which we do in the image above.

Raw Video (Yellow) vs Vertical Sync (Blue)

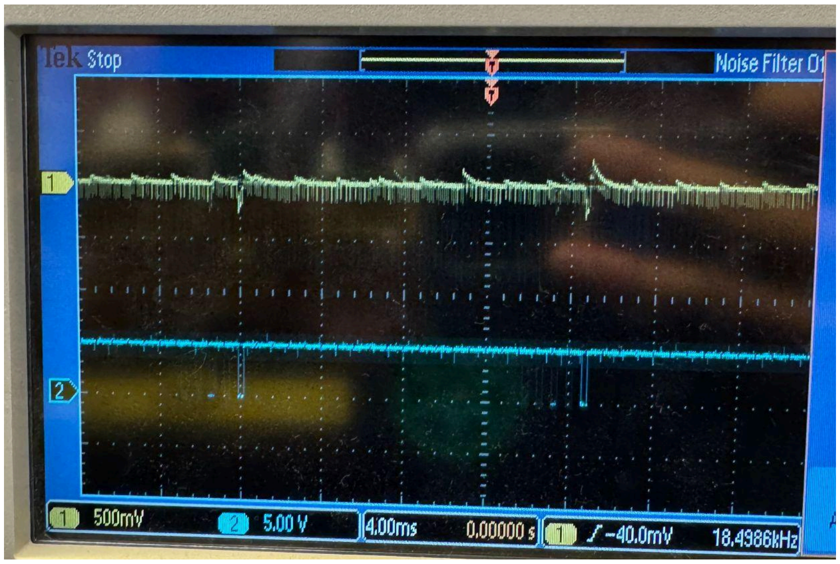
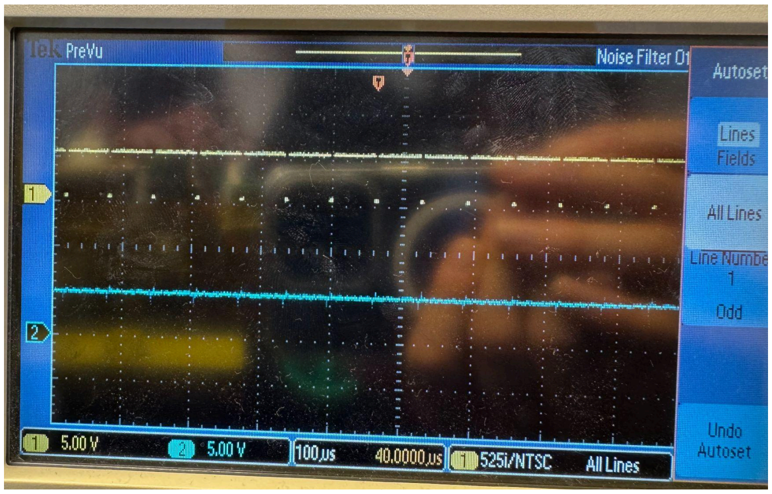


Figure 4. The scope traces showing raw video and vertical sync. The yellow trace is still the video signal but zoomed out much farther so we can see both the horizontal sync pulses (the small negative pulses that are in between the large negative pulses) and the vertical sync pulses (the more prominent negative pulses that dip farther down than the hsync pulses). The vsync pulse indicates the end of a video frame. The blue trace is the reproduction vertical sync input, where the sync separator separates the vsync from the video signal so we see the

corresponding negative pulse at every end of the frame and the signal in between the two pulses are clean (does not include the raw video signal and the horizontal sync pulses).

Composite Sync (Yellow) vs Vertical Sync (Blue)



In this image, we are comparing the csync (in yellow) and vsync (in blue). This is a little zoomed in because we were having trouble zooming out without being able to see the pulse dips in the csync if we wanted to see the pulse dips in the vsync. But essentially, we know that there are 525 lines in one frame (this may depend on the region), so we will see 525 pulse dips before the blue trace shows a pulse dip to indicate the end of the frame.

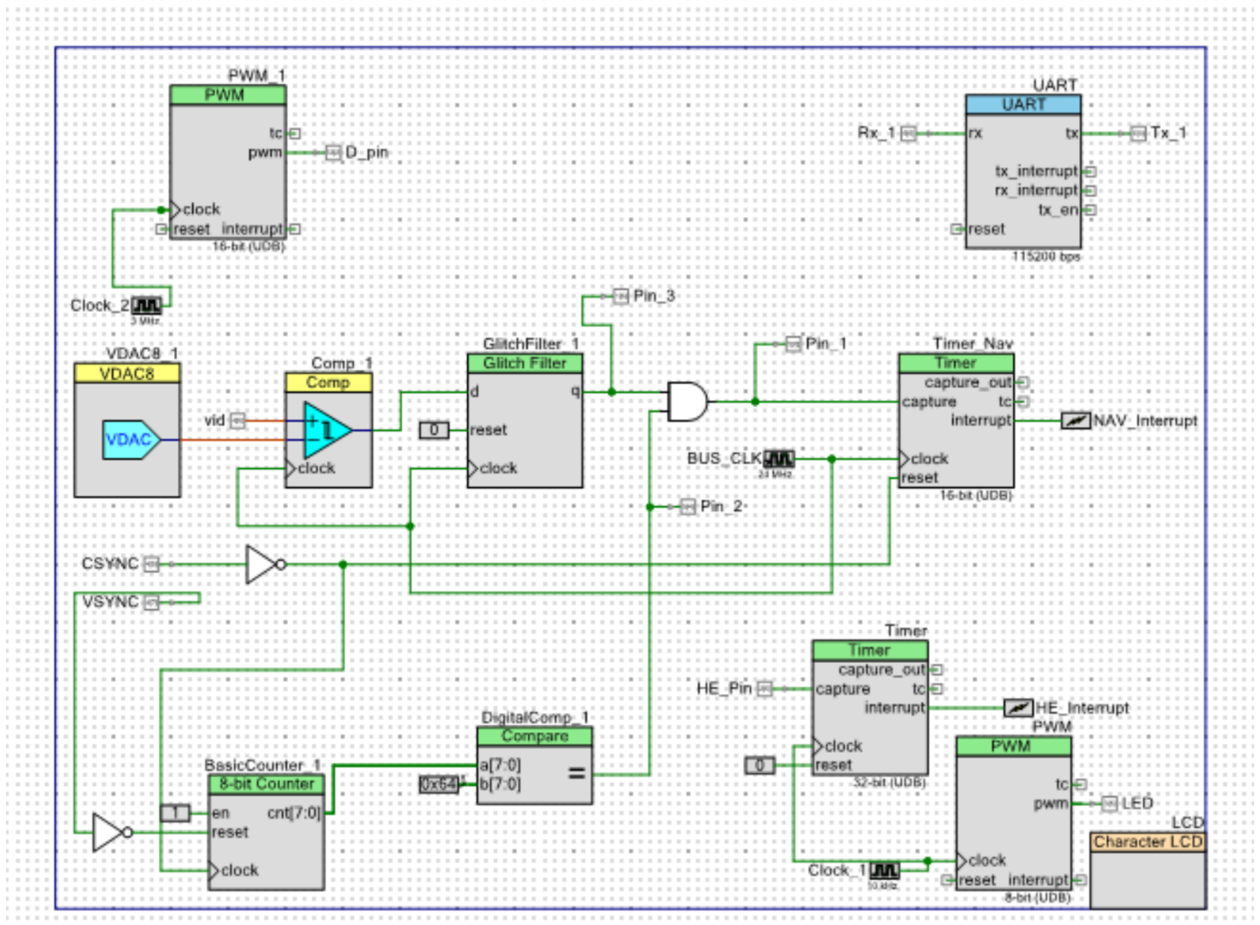
6. Challenges & Improvements

There were quite a few hardware challenges that our group encountered during this navigation milestone. Initially, we realized that some components of our car were becoming too hot and burning up. The burning came from two places. The first was our hall effect sensor which we had to redo, but when we resoldered we realized that we had used the wrong resistor value (much lower than prescribed). The second burning came from the microcontroller on the PSOC which prevented our program code from downloading onto the PSOC. So we replaced our PSOC but then the motors of our car would not run. The issue was that when we made the replacement, the output voltage for our PWM was 3.3V instead of 5V. We learned that there was a hardware component on the PSOC that controlled whether 3.3V or 5V were to be outputted so after this fix our car finally functioned.

With the code itself, we did not have too much trouble implementing the PID control. We initially ran with just proportional control and tried adjusting the Kp value but realized that the car did not navigate very well on tougher turns. It would turn too widely and ultimately the black lines would be out of sight for the camera to see. After adding derivative control, our car was able to turn more sharply as it adjusted to faster angular changes in the black lines and we did not have to add integral control. We found that our Kp value had to be larger than our Kd value and that the adjustments made for our Kd were not drastic (large Kd values made our car veer back and forth in straight paths). One of the improvements that we can make for our future that might be helpful for our final project is to account for the case where the frame no longer sees a black line. We should create some sort of search component so that it can stop and find the black line (or an object) for the future.

This is a general hardware improvement, but if we can shorten the wires on our car it will make it look neater and less cluttered. We also definitely improved debugging our car, especially with using the oscilloscope to identify whether certain inputs were receiving the right voltage and signal and targeting where the car was failing.

7. Code



```c
/* ========================================
 *
 * Copyright YOUR COMPANY, THE YEAR
 * All Rights Reserved
 * UNPUBLISHED, LICENSED SOFTWARE.
 *
 * CONFIDENTIAL AND PROPRIETARY INFORMATION
 * WHICH IS THE PROPERTY OF your company.
 *
 * ========================================
*/
#include "project.h"
#include <stdio.h>
#include <math.h>

// Speed Control Variables
// Target speed: 3.6 feet per second
#define TARGET_SPEED_FTPS   3.6f
// Cheel_radius: 1.25 inches (later, convert to feet by dividing by
12)
#define WHEEL_RADIUS        1.25f
#define PI                  3.14159265f

// Circumference in feet = (wheel radius in feet) * (2 * pi)
// Circumference of 1/5 of the wheel in feet (because there are 5
magnets
static const float WHEEL_CIRC_FT = (2*PI / 5) * (WHEEL_RADIUS /
12.0);

// Speed difference between the actual speed and target speed
double error;

// Captured time on the previous falling edge of the hall effect
uint32 last_time;

// Captured time on the current falling edge of the hall effect
uint32 curr_time;

// Time difference between last_time and curr_time
double elapsed_cycles;

// Speed of the car in feet per second
double speed;

// Proportional control constant
```

```
double kp = 30;

// Integral control constant
double ki = 5;

// Accumulated error over time
double int_error;

// Base compare value
double base = 50;

// Adjusted compare value
double pwm;

// Navigation Variables
// Captured time on the falling edge of the camera input
uint32 nav_capture_time;


// Target time elapsed from the start of the current CSYNC row until
the sensor detects the black line
double nav_time = 623;

// Actual time elapsed from the start of the current CSYNC row until
the sensor detects the black line
double difference_time;

// Time difference between the measured time (difference_time) and
the target time (nav_time)
double nav_error = 0;

// Proportional control constant
double kp_nav = 0.4;

//
double pwm_nav;
double new_nav_time;

// Maximum cycle (timer set to 16 bits)
double max_cycle = 65536;


double conversion_constant = 7.3194;
double prev_nav_error = 0;
double derivative;
```

```
// Derivative control constant
double kd_nav = 0.08;

// Speed Control
CY_ISR(inter)
{
    // Captures current time (in cycles)
    curr_time = Timer_ReadCapture();

    // Time difference between last falling edge and this falling
edge of the hall effect
    elapsed_cycles = last_time - curr_time;
    last_time = curr_time;

    // Car's current speed in feet per second
    speed = (WHEEL_CIRC_FT / elapsed_cycles) * 10000;

    // Proportional control error
    error = TARGET_SPEED_FTPS - speed;

    // Integral control error
    int_error += error;

    // Adjusting the compare value using proportional and integral
control
    pwm = base + (kp * error) + (ki * int_error);

    // Applying new compare value
    PWM_WriteCompare((uint) pwm);

    // Clears the timer interrupt flag
    Timer_ReadStatusRegister();
}

// Navigation Control
CY_ISR(navigation)
{
    // Captures current time (in cycles)
    nav_capture_time = Timer_Nav_ReadCapture();

    // Time elapsed from the start of the current CSYNC row until the
sensor detects the black line
    difference_time = max_cycle - nav_capture_time;
    prev_nav_error = nav_error;
    // Proportional control
    nav_error = difference_time - nav_time;

    // Derivative control
    derivative = (nav_error - prev_nav_error) / difference_time;

    // Converts timing output to a compare value for steering servo.
    new_nav_time = nav_time + (kp_nav * nav_error) + (kd_nav *
derivative);
    pwm_nav = conversion_constant * new_nav_time;

    // Applying new compare value
    PWM_1_WriteCompare((uint) pwm_nav);

    // Writing the live error values to the UART
    char buffer[64];
    sprintf(buffer, "%d\r\n", (int) nav_error);
    UART_PutString(buffer);

    // Clears the timer interrupt flag
    Timer_Nav_ReadStatusRegister();

}


int main(void)
{
    CyGlobalIntEnable; /* Enable global interrupts. */

    /* Initializations */
    UART_Start();

    // For speed control
    PWM_Start();
    Timer_Start();

    // For navigation
    Timer_Nav_Start();
    VDAC8_1_Start();
    Comp_1_Start();
    PWM_1_Start();

    // To navigation interrupt
    NAV_Interrupt_Start();
    NAV_Interrupt_SetVector(navigation);

    // To hall effect interrupt
    HE_Interrupt_Start();
    HE_Interrupt_SetVector(inter);


    for(;;)
    {
    }
}

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