Navigation Report

Callista Chong: cc0225@princeton.edu
Mayan Wasu: mw8270@princeton.edu
Group 308

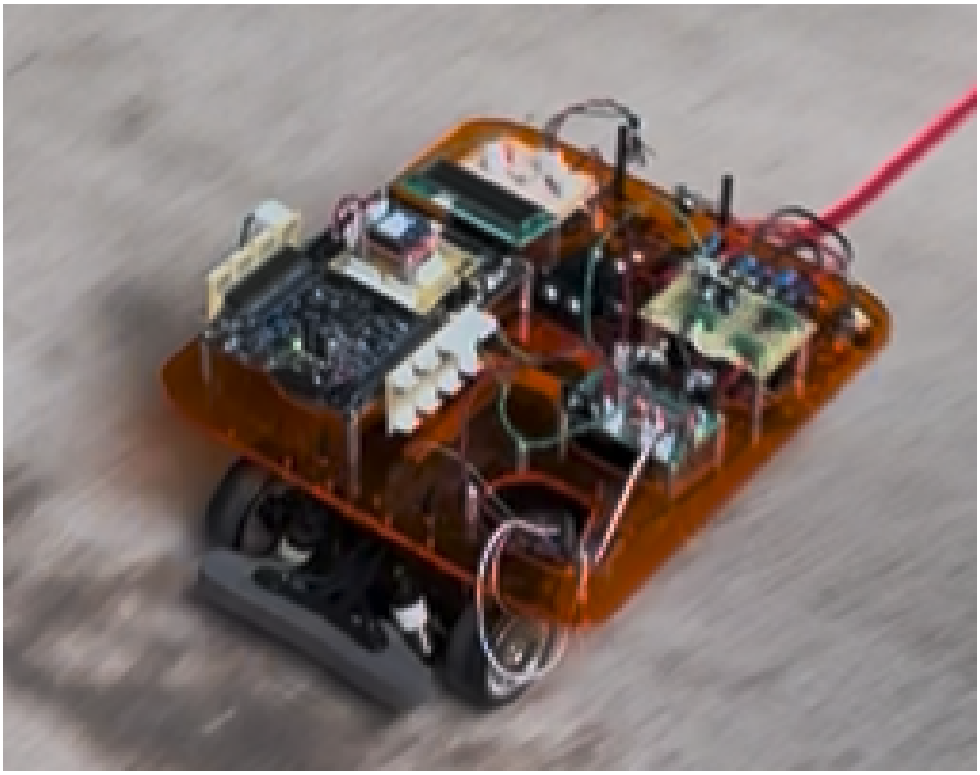

Figure 0: The Chongasu-Mobile

Statement of Objective

The primary objective of the Navigation portion of this project was to modify our existing car to autonomously navigate two full laps of a closed-loop track in under 60 seconds as accurately as possible. Achieving this goal required integrating new hardware components, including a digital camera, video board, and mast, as well as modifications and extensions to our existing Speed Control PID software. At a high level, we track the timing of the drop in voltage that happens when the black line that the car is following is detected along one horizontal line in the video frame; we track the difference between this measured time to a reference time (taken when the car is centered on the line) to adjust the steering of the car to keep it aligned on the track via basic PID control. These additions enabled the car to successfully interpret real-time visual information for navigation, while maintaining adequate speed control. Modifications were successfully implemented, and the car passed functionality tests.

Overview of Key Subsystems and Components

Our system relies on PID control for navigation of the car. As mentioned above, when the camera detects a black line, there is a corresponding voltage drop. The lines of the camera output frame are read from left to right; so, when the black line is positioned to the left of the camera, the voltage drop happens sooner than if the car was centered on the track. Similarly, when the black line is positioned to the right of the camera, the voltage drop happens later than if the car were centered. We empirically found the ‘baseline’ time - the time that it takes for the voltage drop to occur when the car is centered on the line. By comparing the actual time with this baseline time, we can calculate the error. We then use this error to modify the PWM signal to the servo, which adjusts the car’s steering and keeps it centered on track. This section will discuss in depth the newly integrated hardware components in our car that assist us with this navigation control implementation.

Components

1) Camera

A PTC08 serial camera module was mounted above the front edge of the car using a laser-cut, clear acrylic mast. Specifications and design details of this mast are discussed in later sections. The camera has a downward view of the track immediately ahead of the car. The camera outputs a standard NTSC composite video signal, which contains visual data as

well as synchronization information embedded in the analog waveform. This waveform is transmitted directly to the LM1881 video sync separator, where the horizontal (H-Sync) and composite (C-Sync) timing pulses are extracted.

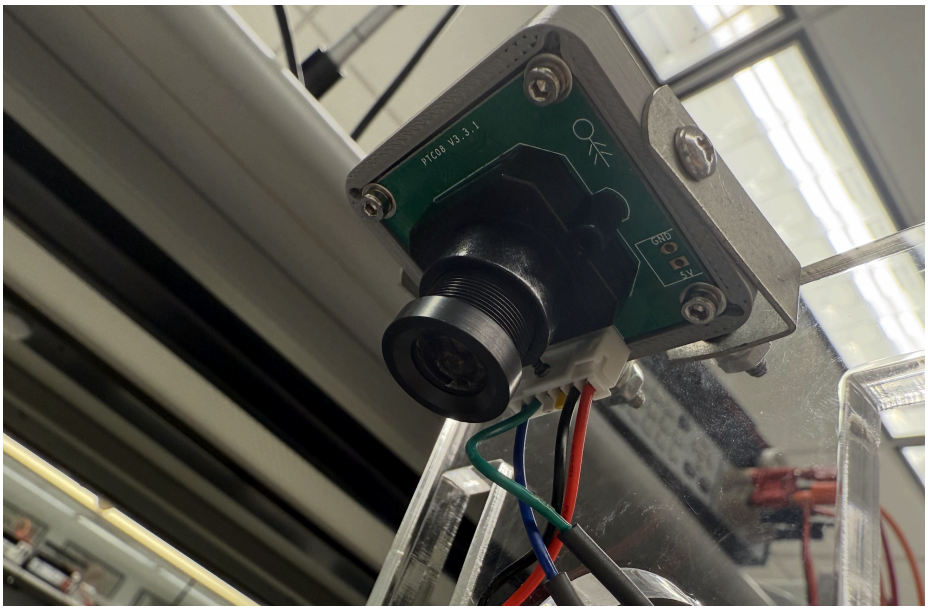

Fig. 1 PTC08 serial camera module

2) Video Board

To extract timing information from the composite video signal output generated by the camera, an LM1881 video sync separator circuit is used. The chip isolates the composite (C-Sync) and vertical (V-Sync) synchronization pulses from the NTSC video stream, which allows the PSoC to synchronize each frame and line in real time: the C-Sync falling edge triggers a timer interrupt for line indexing, and the V-Sync rising edge resets the line counter for each new frame. This operation allows for consistent frame-based navigation control.

The video board circuit takes in 5V from the voltage regulator board. An LED was placed to indicate sufficient voltage along the positive rails. This 5V rail powers the camera and the LM1881 chip. The camera's raw video signal is sent to the LM1881 through a capacitor with

a small resistor to ground. The capacitor blocks DC so that only the video contents pass, and the resistor establishes a stable DC ground reference so that the node doesn't float**.** The LM1881 separates the C-Sync and H-Sync signals and then transmits them to the PSoC. The RSET pin is connected to ground through an RC network that simply defines internal timing and noise filtering of the chip. This helps to keep sync pulses relatively clean regardless of the noisy analog video.

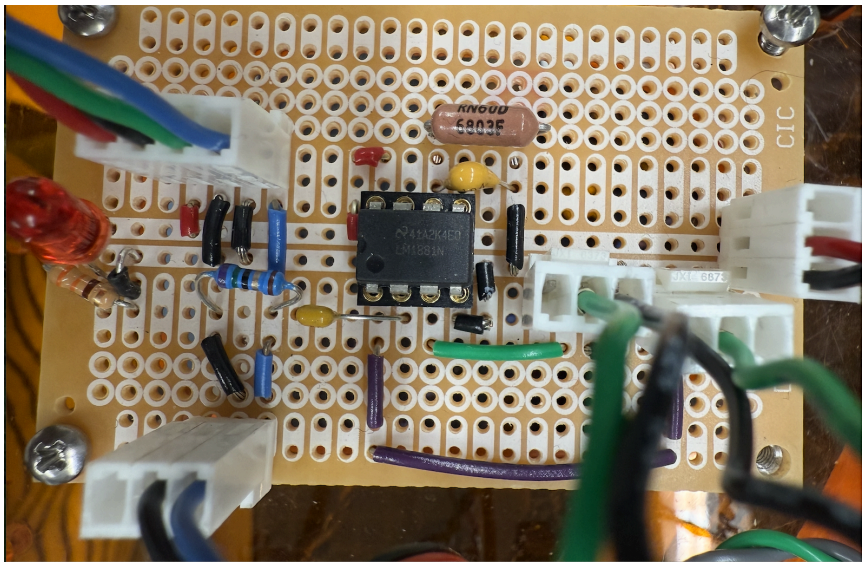

Fig 2. Photo of Video Board

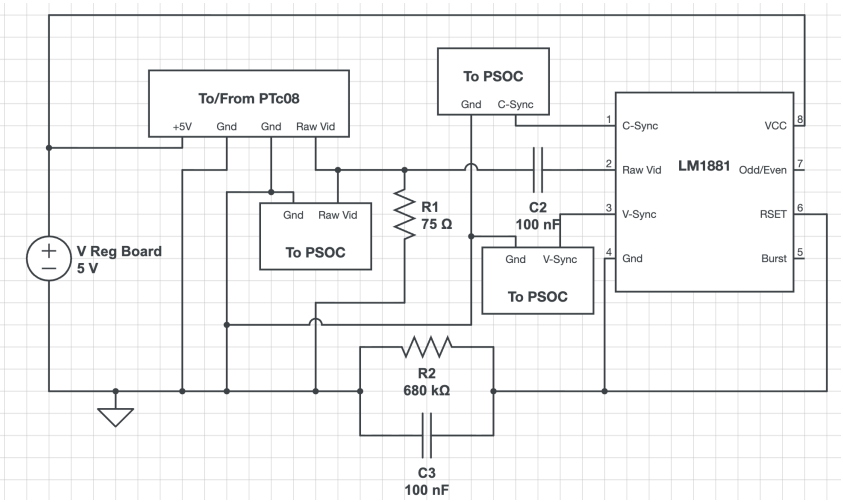

Fig 3. Schematic of Video Board via CircuitLab

3) Mast

The clear acrylic mast attaches to the front chassis using four brackets for peak stability. Holes were added to mount the brackets and camera, and cutouts were made to manage wires. The mast holds the camera 11 inches above the chassis and mounts the camera such that it has a clear downward view of the track immediately ahead of the car.

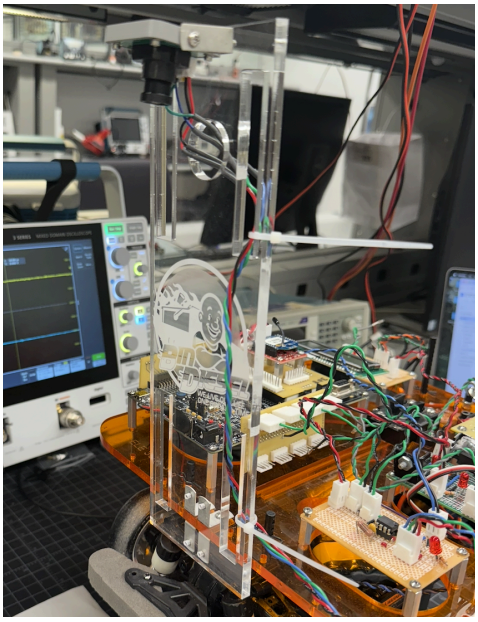

Fig 4. Photo of mounted mast

System Validation Data

To validate the connections of the video board, camera, and mast, scope traces of the C-Sync, H-Sync, and raw video are shown.

1) Raw Video (yellow) and C-Sync (blue)

C-Sync indicates the beginnings of each new transmitted line. Each unit of yellow trace is one horizontal line. Traveling across the scope (forward in time), each next unit is the next line down on a frame of video. We can observe the C-Sync trace going to zero at the end of each line, and rising back to 5V at the start of the next line, indicating correct synchronization.

Fig.5 Composite Video vs. C-Sync Trace

2) Raw Video (yellow) and V-Sync (blue)

V-Sync indicates the beginning of a new frame. At the end of each frame, the raw video trace sends a series of pulses that deviate from the normal line signal information. During this period on the scope, we can observe the blue V-Sync sinking to zero, and then rising up to 5V again at the start of the next frame. This indicates correct synchronization.

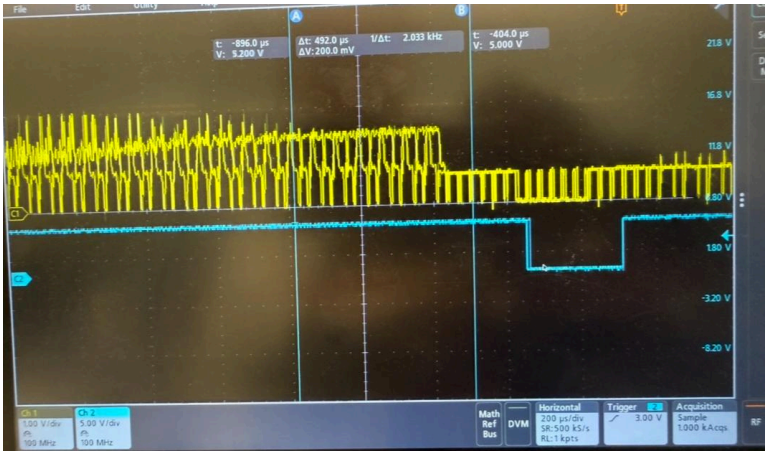

Fig.6 Composite Video vs. V-Sync Trace

3) C-Sync (yellow) and V-Sync (blue)

Pictured is the C-Sync sending inverted pulses on the leftmost side of the screen to indicate a new frame at the same time as V-Sync taking 0V. Then, C-Sync follows the analog trace and inverts back, and V-Sync takes 5V at the start of the new frame. This indicates correct synchronization.

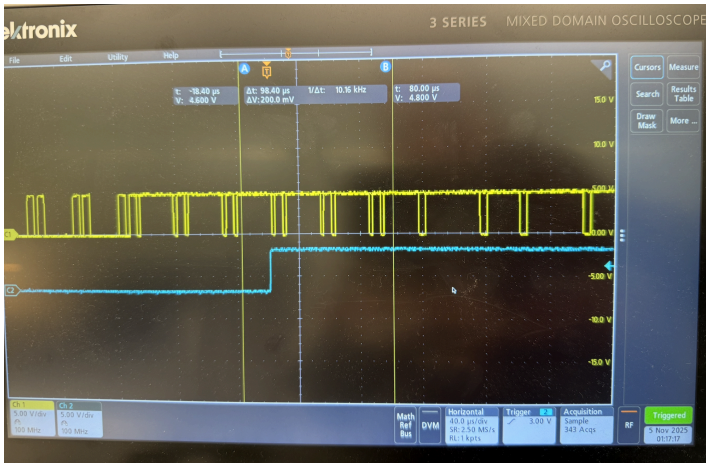

Fig.7 C-Sync vs. V-Sync

Conclusion

The provided scope traces prove that the video board is successfully supporting the functionality of the LM1881 video sync separator; each scope trace shows correctly synchronized signals derived by the LM1881 from the original analog signal. In the analog trace, we can also observe a clear and consistent voltage high level when the white paper is detected, and a voltage low where the black line is. So, we know that the camera is feeding lines and frames correctly, seeing the environment reliably, and interacting with the video board as expected.

Algorithm

The final piece connecting the navigation system is the algorithm. In PSoC Creator, we expanded our speed control project to include simple PID control for navigation purposes. The product of this algorithm is a continually revised PWM [3000, 6000] sent to the servo to adjust the direction of the wheels. The base PWM aligns the wheels perfectly straight, then we add a proportional error term. Tuning was sufficient without the addition of an integral or derivative term.

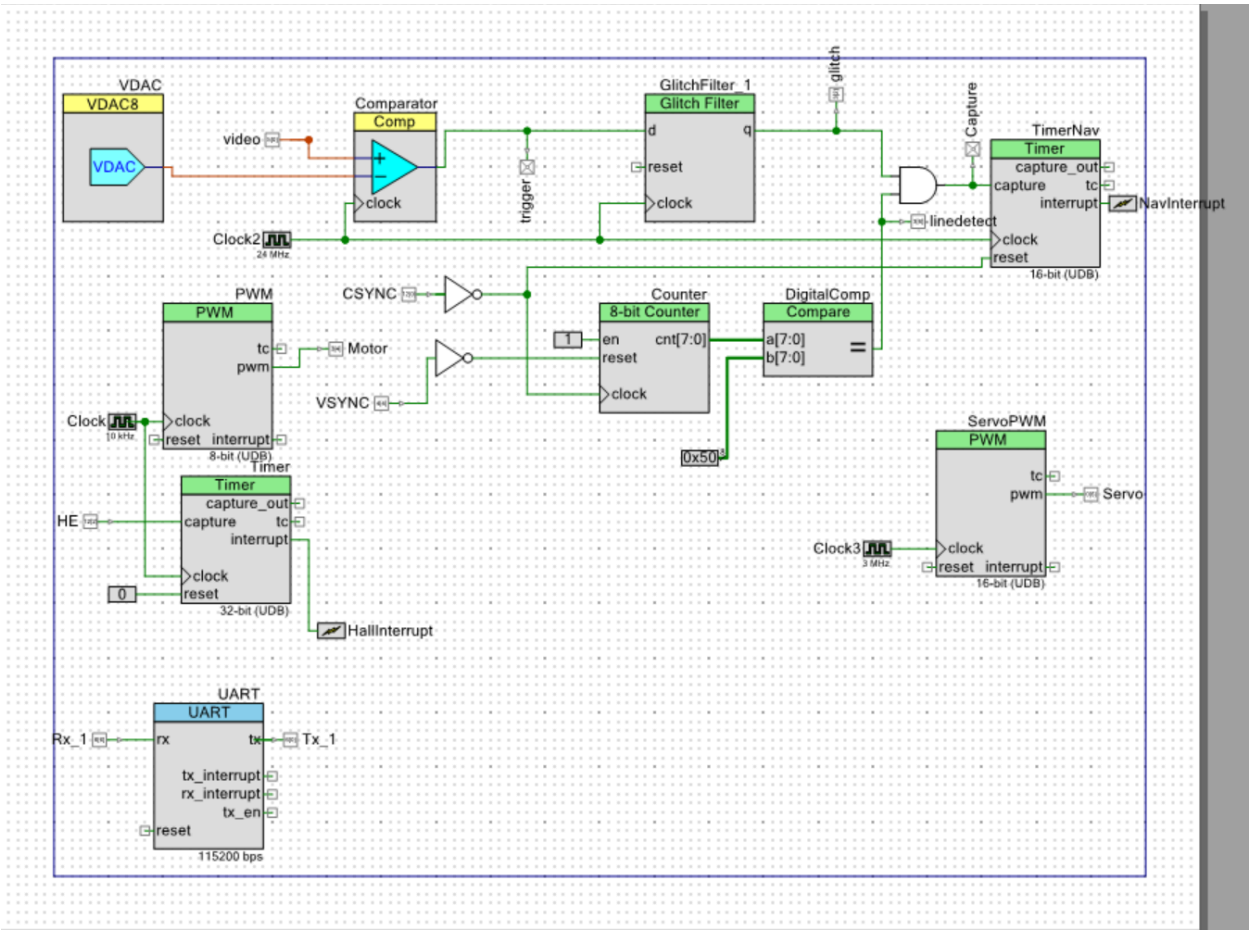

Fig.8 Top Design (integrating both speed control and navigation)

The interrupt measures the elapsed ticks from the start of a line to when the first edge of the black tape is detected. The elapsed ticks when the car is aligned perfectly straight on the tape are used as the target time from which the error is calculated.

In order to do this, we first needed to determine whether a pixel was black or white by comparing it to a threshold voltage. The digital-to-analog converter (DAC) takes in an input value, which is then converted to a constant voltage. We determined the threshold value empirically using oscilloscope traces by choosing the midpoint between the voltage given by a black pixel and the voltage given by a white pixel. The analog comparator helped us to compare the outputs from the DAC against this threshold value. When we are above this threshold value, the output is high (5V) and if we are below threshold, the output is low (0V). So, the comparator was outputting digital signals that were high for white pixels and low for dark pixels.

We then imposed a glitch filter in order to reduce noise and to prevent ‘hallucinations’, where small specks of dark pixels can result in false positives, where the car believes it is viewing the track. The glitch filter would only output the value at q when the input at d has been constant for a set number of clock cycles (32 samples), helping us ensure that the dark pixels seen corresponded to the black tape of the track and not ‘noise’.

We also had to select the particular line from the frame that we wanted to analyze. This is because the PSoC is not powerful enough to quickly analyze every line of every frame. So, in order to select a particular line, we used a counter and a digital comparator. We chose line 80 to get data that was sufficiently far ahead of the car while maintaining a high level of resolution. The counter is reset on every V-sync rising edge (new frame) and is incremented on each H-sync rising edge (new line). When we hit line 80, the digital comparator outputs a high signal, and the interrupt is triggered. This allows us to only start measuring ticks when we are at line 80.

This output and the output q from the glitch filter are fed into an AND gate. The output of this AND gate is only high from the beginning of line 80 until we detect the black line of the track, because once the black pixels are detected, the output q of our comparator goes low. Our timer is set to capture on a rising edge, signifying that we have read to the end of the black line.

We can figure out the error of the car by finding the difference between our target time and the elapsed time. This error term is multiplied by a proportional scalar, k_p, and then added back to the base PWM and sent to the servo.

The results of our navigation control on a straight path are shown. There are a few important things to note about our algorithm's performance:

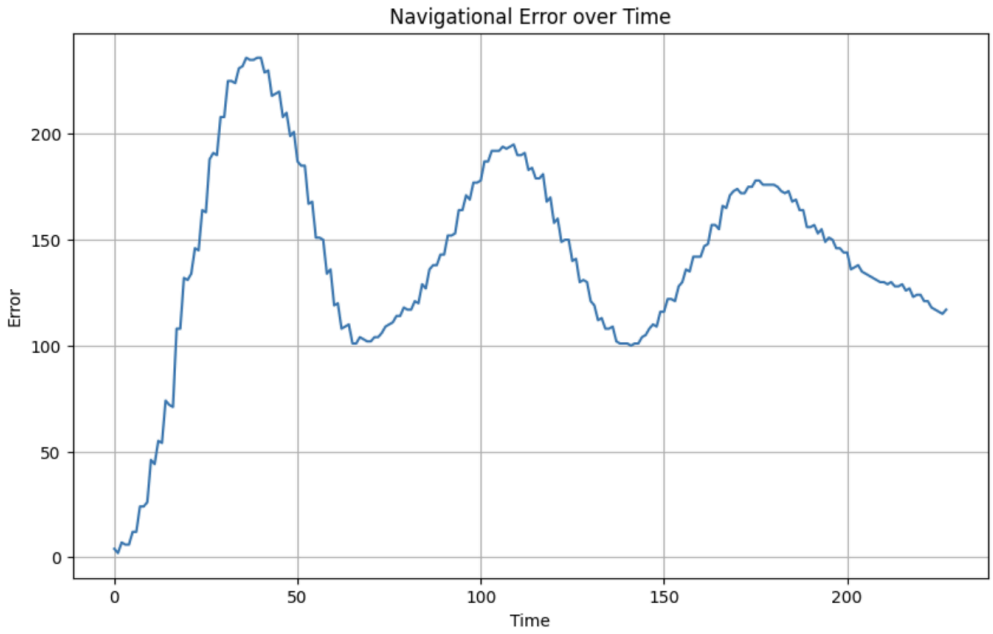

Fig. 9 Error vs. Time

Most obviously, on a straight path, the oscillations center around 150 ticks, not zero. The base PWM was continuously adjusted to center on zero, but on the track, drifted up to ~150. After some analysis and debugging, we realized the most significant thing contributing to this drift was an unstable camera. We measured our target time while the car was stationary, but during motion, the connections between the camera itself and the metal elbows kept loosening, which caused the camera to swing slightly to the right over time. We attempted to adjust the base PWM to preemptively account for this instability, but found that performance was most consistent with the allowed offset. Although a non-zero reference is not ideal, our car's operation still surpassed operation thresholds.

Additionally, there are some obvious oscillations observed. This is to be expected using PID control, especially with only proportional control implemented. However, we can observe oscillations dampening as time goes on, indicating that k_p was tuned sufficiently. Unfortunately, there was no available straight track long enough to observe this trend further.

Discussion of Challenges and Design Choices

Designing the video board was a lengthy process due to the high density of connections required by the LM1881. We attempted to minimize solder bridges due to the solder's

lower conductivity, though this is not ultimately that important for this level of operation, and connected things intuitively to make debugging easier. This was ultimately very useful, as we encountered several issues that required careful debugging: a burned resistor on the input line caused us to lose sync detection, and a number of poorly soldered connections along the vertical sync path caused intermittent signal loss, which prolonged debugging time.

Once the software was integrated, to isolate problems, we added compare and reset pins to probe intermittent nodes to visualize signals at key points. This ended up revealing that the signal was lost along the V-Sync path, leading to another hardware fix. Another key problem was adjusting edge-trigger selection in the PSoC. Initially, the system used the falling edge of the C-Sync pulse to trigger the navigation interrupt, but elapsed times were not consistent, and edges were missed. Switching to the rising edge fixed the inconsistency issues. Correcting these issues restored clean, periodic sync pulses to the PSoC.

Mechanically, we faced stability challenges with the camera's metal mount. Since the right screw hole was not secure, the camera angle would drift. In conjunction with expected flexion and shifts from the mounting system, we ended up with zero-error reference shifts. This problem was mitigated to the best of our abilities by tightening mechanical junctions and adjusting the base PWM, and performance was sufficient in testing.

Future Improvements

As we move into our final project, our first goal is to zero out the navigation error reference point. We plan to integrate sound-based angle-of-arrival estimation, which will need a very reliable navigation framework. One important step we can take is to adjust the way the camera is mounted so that it is stable. One option is to add glue to the screws to prevent them from loosening. We can then re-tune our servo base PWM to reclaim zero error reference.

Another area of focus for us would be our connections between different components. Currently, we have a lot of wires that are longer than they need to be, and some of them are tangled. By wiring and securing cables in a more systematic way, we would be able to debug more easily, reduce the chance of any possible hardware errors, and consequently improve overall system performance as well as the speed at which we can build our final project.

Commented Code

```
/* ========================================
 *
 * 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>
#include <stdlib.h>


#define CLOCK_HZ 10000UL // SpeedTimer clock (10 KHz -> 100 µs ticks)
#define TIMER_MAX 4294967295 // 32-bit timer (2^32 - 1)
#define MAGNET_SPACING_FEET 0.12438 // Distance per HE pulse in feet (2pi * 1,3/16inches)/5
#define TARGET_SPEED_FPS 4.0   // Desired speed in feet/sec


#define PWM_MIN 1
#define PWM_MAX 230


// variables we need to track times from HE sensor
volatile uint32 oldSeen = TIMER_MAX;
volatile uint32 newSeen;
double elapsedTicks;


// variables needed for speed and errorvolatile double speed;
double speed;
volatile double error = 0;
volatile double prevError = 0;
volatile double integralError = 0;
volatile double derivativeError = 0;


// variables for PID - the proportional constant and the integral term
double kp = 60; // Should be 60
double ki = .004; // Should be 0.004


// Initial PWM value (base speed) to MOSFET
double pwm;
double pwmBase = 8;


// Used to help print to debug and collected data
char print_speed[32];
char print_error[32];


char dbg[64];
```

```
// ********************************************************************************

// initializing the error and the PWM values for the Servo
int16 nav_err;
double nav_err_acc = 0;

double nav_olderr = 0;
double nav_derr;

double nav_pwm_old = 4500;
double nav_pwm_base = 4500;
double nav_pwm;

int16_t NavelapsedTicks;

// variables we need to track times from camera
int16 time;
int16 target_time = 375;


// variables for PID (Nav)
double nav_kp = 2.33;
double nav_ki = 0;
double nav_kd = 0.4;

// Slew limit for servo command to suppress jittery jumps (in PWM counts)
#define NAV_SLEW_LIMIT 40


// Hard bounds for the servo turning
#define SERVO_MIN 3000
#define SERVO_MAX 6000

// for debugging
char print_elapsedTicks[32];
char print_naverror[64];
char print_navpwm[64];



// ********************************************************************************

// This interrupt will trigger for navigation
CY_ISR(inter2)
{
   // time at interrupt trigger (black line detected, signal rising edge)
   time = TimerNav_ReadCapture();

   // time since the horizontal line began (since timer was reset to MAX = 65535 8-bits)
   nElapsedTicks = (double)(65535 - time);


   // Signed lateral error (ticks): positive one side, negative the other
   // target_time is the tick value you measured when centered over the tape
   nav_err = (double)target_time - nElapsedTicks;


   // Simple PI-D terms (discrete)
   nav_err_acc += nav_err;                          // integral
   double nav_derr = nav_err - nav_olderr;          // derivative (per frame)
```

```
   // Raw servo command centered around base (straight wheels) value
   double nav_pwm = nav_pwm_base
                  + nav_kp * nav_err
                  + nav_ki * nav_err_acc
                  + nav_kd * nav_derr;


   // Saturate to safe bounds
   if (nav_pwm > SERVO_MAX) nav_pwm = SERVO_MAX;
   if (nav_pwm < SERVO_MIN) nav_pwm = SERVO_MIN;


   // Slew-limit: cap the step size vs previous command (reduces spikes from noisy detects)
   double delta = nav_pwm - nav_pwm_old;
   if (delta > NAV_SLEW_LIMIT)  nav_pwm = nav_pwm_old + NAV_SLEW_LIMIT;
   if (delta < -NAV_SLEW_LIMIT) nav_pwm = nav_pwm_old - NAV_SLEW_LIMIT;


   // Apply to servo
   Servo_PWM_WriteCompare((uint8)nav_pwm);


   // Bookkeeping
   nav_olderr   = nav_err;
   nav_pwm_old  = nav_pwm;


   // Clear interrupt source
   TimerNav_ReadStatusRegister();
}


*/

//
*****************************************************************************************************
***********************************

// This interrupt will trigger every time magnet passes HE sensor
CY_ISR(inter){

// captures the time from the timer and calculates the elapsed time from the previous capture
newSeen = Timer_ReadCapture();


// Calculating how many ticks elapsed between magnet sighting
elapsedTicks = (double) oldSeen - (double) newSeen;


oldSeen = newSeen;


// speed is calculated based on the elapsed time and the distance between magnets on the wheel
speed = 1243.8 / elapsedTicks;
error = 4.0 - speed;

/*
// print for XBee Data Collection
sprintf(print_speed, "%f", speed);
UART_PutString(print_speed);
UART_PutString("\r\n");
*/
```

```
// prevents error being added to the accumulated error if car stationary for a long time
if(elapsedTicks <=5000){
   integralError += error;
}


prevError = error;


// new pwm is calculated based on the error
pwm = pwmBase + kp * error + ki * integralError;


// bounds the pwm values to the upper and lower limits
if (pwm>PWM_MAX)
pwm = PWM_MAX;
if (pwm<PWM_MIN)
pwm = PWM_MIN;
// the new pwm signal is sent to the motor board
PWM_WriteCompare((uint8)pwm);
// clears interrupt
Timer_ReadStatusRegister();
}


//
*****************************************************************************************
**********************************

CY_ISR(testing)
{
     // time at interrupt trigger (black line detected, signal falling edge)
    UART_PutString("************************\r\n");
   time = TimerNav_ReadCapture();

   // time since the horizontal line began (since timer was reset to MAX = 65535 8-bits)
   NavelapsedTicks =  (65535 - time);

   nav_err =  NavelapsedTicks - target_time; // if +ve, we need to go right, if -ve, need to
go left (PWM goes up)


    // print for XBee Data Collection

   sprintf(print_naverror, "%hd \r\n", nav_err);
   UART_PutString(print_naverror);
   //sprintf(print_elapsedTicks, "elapsedTicks: %hd \r\n", NavelapsedTicks);
   //UART_PutString(print_elapsedTicks);



   nav_err_acc += nav_err;                          // integral
   nav_derr = nav_olderr - nav_err;




  nav_pwm = nav_pwm_base + nav_kp * nav_err; // + nav_ki * nav_err_acc + nav_kd * nav_derr;
```

```
  // sprintf(print_navpwm, "Nav PWM: %.3f\r\n", nav_pwm);
  // UART_PutString(print_navpwm);

// Saturate to safe bounds
   if (nav_pwm > SERVO_MAX) nav_pwm = SERVO_MAX;
   if (nav_pwm < SERVO_MIN) nav_pwm = SERVO_MIN;



   ServoPWM_WriteCompare((uint16)nav_pwm);



   nav_olderr   = nav_err;
   nav_pwm_old  = nav_pwm;


   // Clear interrupt source
   TimerNav_ReadStatusRegister();

}


int main(void)
{
// Initialize components + Enable interrupts
CyGlobalIntEnable;
Timer_Start();
PWM_Start();
HallInterrupt_Start();
UART_Start();
HallInterrupt_SetVector(inter);




TimerNav_Start();
ServoPWM_Start();
VDAC_Start();   // your VDAC instance name
Comparator_Start();
NavInterrupt_Start();
NavInterrupt_SetVector(testing);


// Main control loop
for(;;)
{


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