Group 309
Ryan Zimmitti & George Kopf
rz6542@princeton.edu & gk3946@princeton.edu

For speed control, navigation, and the final project, your team should upload two files to Canvas (one report per group):

- Lab report (PDF)
- A .zip file containing all code & designs (including .cydsn for PSoC, .ino for Arduino, etc.)

Statement of Objective:

Statement of Objective (5 pts.) Your car must go as fast as you can around a track. For completion (25% of your overall grade for this part), your car should complete two laps around the track in under 60 seconds

Overview of Key Subsystems & Components:

Overview and brief explanation of key sub-systems and components (30 pts.)

Camera Board

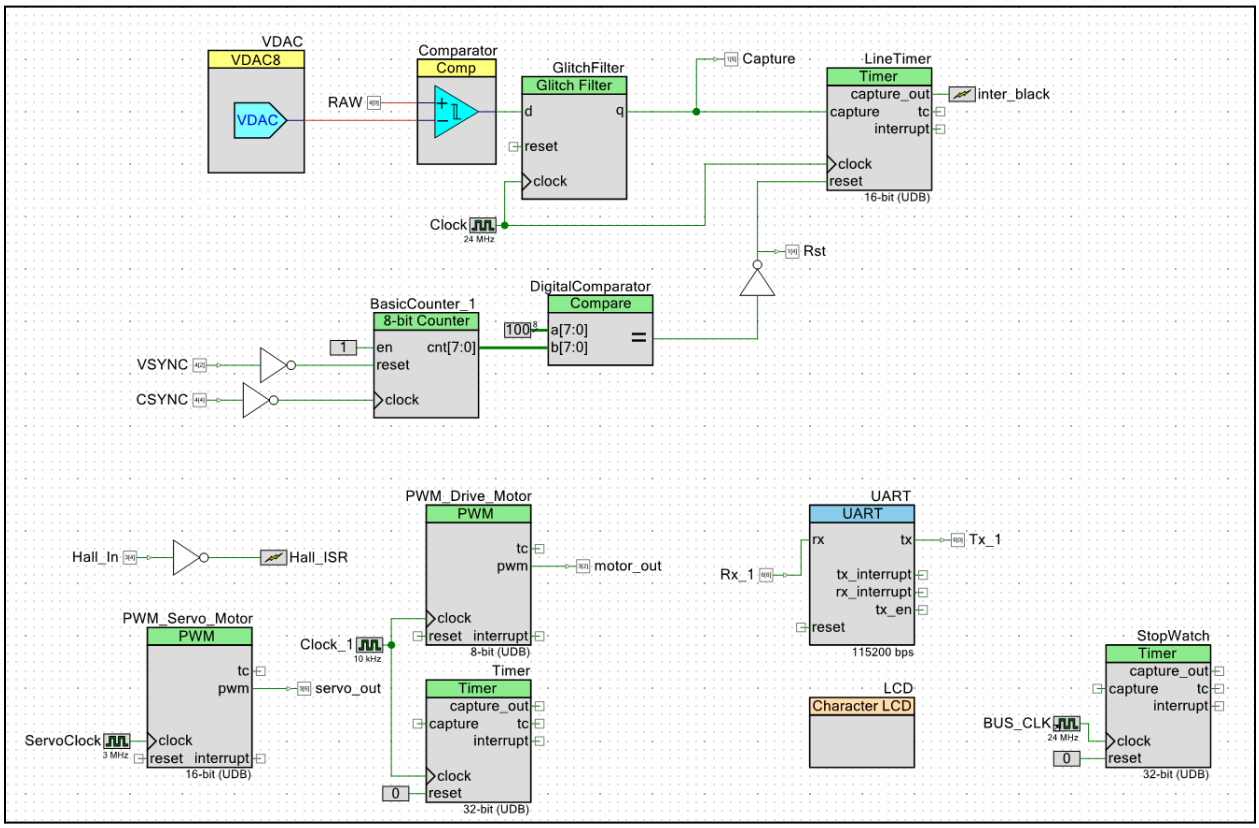
The camera we were originally given was a PTC08 v3.3.1, but for our final completion of the demo we switched to the C-Cam-2A, the older model of camera (this helped reduce noise). Functionally, however, they operate the same for our purposes. The camera takes in raw footage and sends out a condensed signal with layered information determining frame, line, and color timing. To get the timing of this signal, we feed it into our LM1881, the video sync separator chip, which splits the signal into composite and vertical sync pulses which are normally high, and pulled down for each pulse.

The camera receives 6V power from the voltage regulator board. It outputs an analog signal that is pulled down by a 75 ohm resistor (impedance matching). Then, the signal is sent to two places: the PSoC directly, and the LM1881 video sync separator chip through a 0.1 microfarad decoupling capacitor on pin 2. The LM1881 needs a capacitor and resistor on Rset pin 6 to set the line scan frequency as specified in the schematic (0.1 microfarad cap and 680k resistor in parallel to ground). Pin 1 and 3 send out two signals to PSoC: the composite sync (CSYNC) and vertical sync (VSYNC).

The LM1881 takes the camera output and splits it into three output signals. The raw video footage signal sends a continuous series which represents lines across the camera's view. In each, it can be seen how the voltage of the output correlates to what the camera is looking at. White is high voltage, black is low. However, just this footage signal does not give our system the context it requires to map those voltage changes to what the camera is seeing. That's where the other two LM1881 output signals come in. The CSYNC signal is a pulse that drops at every new line. The VSYNC signal is a pulse that drops at every new frame. Combined, we can use

these signals to determine exactly what line any given bit of the raw footage signal happens within. We use VSYNC to identify a new frame, then count the CSYNCs to identify which line we are at.

Top Design & Software



Our Top Design differs slightly from what most other groups did. Functionally, they should be equivalent, but we switched things up as part of our debugging process to reduce delay in the system.

In order to count video lines, CSYNC inputs into the clock of an 8-bit counter. The counter increments every CSYNC signal, which effectively counts each line in the footage as it passes. This count outputs into a digital comparator block which compares the counter value to the number 100. Thus, when we are at line 100, the digital comparator sends a positive signal. VSYNC inputs into the reset of the 8-bit counter, so every new frame we reset the count and start over from 0. The output of the digital comparator hits a NOT gate that feeds into the reset of the 16-bit timer LineTimer. Thus, LineTimer is continually being reset *except* when the output of the digital comparator is high, which means LineTimer only increments while we are at line 100 in the footage. This section of analog logic effectively allows us to only activate LineTimer while the camera footage is at line 100 of each frame, which lets us standardize which part of the frame we base our line following off of.

To identify the black line, the raw video is sent into a comparator that measures its voltage against an 8-bit VDAC. VDAC stands for Voltage Digital-Analog Converter, and the

VDAC converts a digital number into a corresponding analog voltage value. We set the output of VDAC to 700 mV, which was chosen based on the range of voltage values the raw video signal outputs when looking at the line on the floor. The idea is that when looking at the white floor, the raw video voltage should be higher than the VDAC value, while when looking at the black line, the voltage should be lower than VDAC. Thus our comparator outputs high when looking at white, and low when looking at black. This feeds into a glitch filter block set to 36 samples (which is 1.5 us at 24 MHz). The glitch filter blocks signals that don't hold steady for at least 1.5 us. This helps weed out noise in the video signal, so the car doesn't dive off the track chasing phantom voltage spikes/drops. The output of the glitch filter feeds into the capture input of LineTimer (set to rising edge). This logic allows the car to accurately trigger when it sees the black line.

In order to identify our position relative to the black line, these two halves combine to allow us to determine the car's position relative to the line. *Only* when the video signal is at line 100 do we stop resetting LineTimer, which starts counting the time from the start of the line. Then, *only* when the raw video voltage drops below VDAC's 700 mV (indicating the line), the comparator will be set low (along with glitch filter), then the voltage rises right as the line is leaving the camera view, which is when LineTimer's capture is triggered. This triggers an interrupt that records LineTimer's value in that moment. The amount of time recorded represents where the line is in the frame of view of the camera. We determined the center value of our camera to be 1150 (*setPointDiff* in the code). This is the amount of time it takes to pan across line 100 and reach the black line when the car is manually centered to the black line on the floor (the center value was closer to 900 with the PTC08 camera, but when we switched to the C-Cam-2A it changed – this is likely due to changes in angle when we remounted the replacement camera).

The simple logic is that if LineTimer reads less than 1150 when the capture triggers, the line is further to the left of the camera's frame of view (or in other words, the car is on the right of the line). If LineTimer reads greater than 1150, the line is further to the right (the car is on the left of the line). We can use this error (LineTimer output - 1150) to direct the car back towards the line, converting negative and positive error into respective instructions to turn left or right).

PD Control

Once we convert from camera signals to an error value that represents how far to the left or right of the line we are, we can use that as the basis for our PD control loop.

The proportional portion of our control is based on error, and directly changes in strength based on how large error is. When the error is negative, we turn left. When the error is positive, we turn right. The greater the absolute value of error, the larger the turn. Our proportional gain has to be high enough to effectively respond to turns in time, so we ended up using a Kp of 2.7. When we have too small of a proportional gain → the car is too sluggish and drives past turns before it can follow, losing the line. When we have too large of a proportional gain → the car is

too jittery and experiences growing oscillations as it drives until it swerves off the line and loses it.

The derivative portion of our control is based on change in error over time. We used positive derivative (error - previous error), which causes our derivative term to resist the proportional portion. When the change in error is big, we resist more strongly. This helps dampen oscillatory behavior, which can be a problem when moving at higher speeds with larger proportional gains. As our oscillations grow, the resisting force of the derivative term grows too, slowing down our response and keeping oscillations in check. Our final Kd was 0.2. When we have too small of a derivative gain → the car struggles with oscillations. When we have too large of a derivative gain → the car is sluggish to respond and misses turns. This seems like the inverse of the proportional gain, but due to the way that proportional control increases in relation to error size while derivative control increases in relation to change in error, they are not 1:1. We can use these different properties to tune the car so that it is responsive enough to turn but dampens oscillations to an acceptable level as they appear.

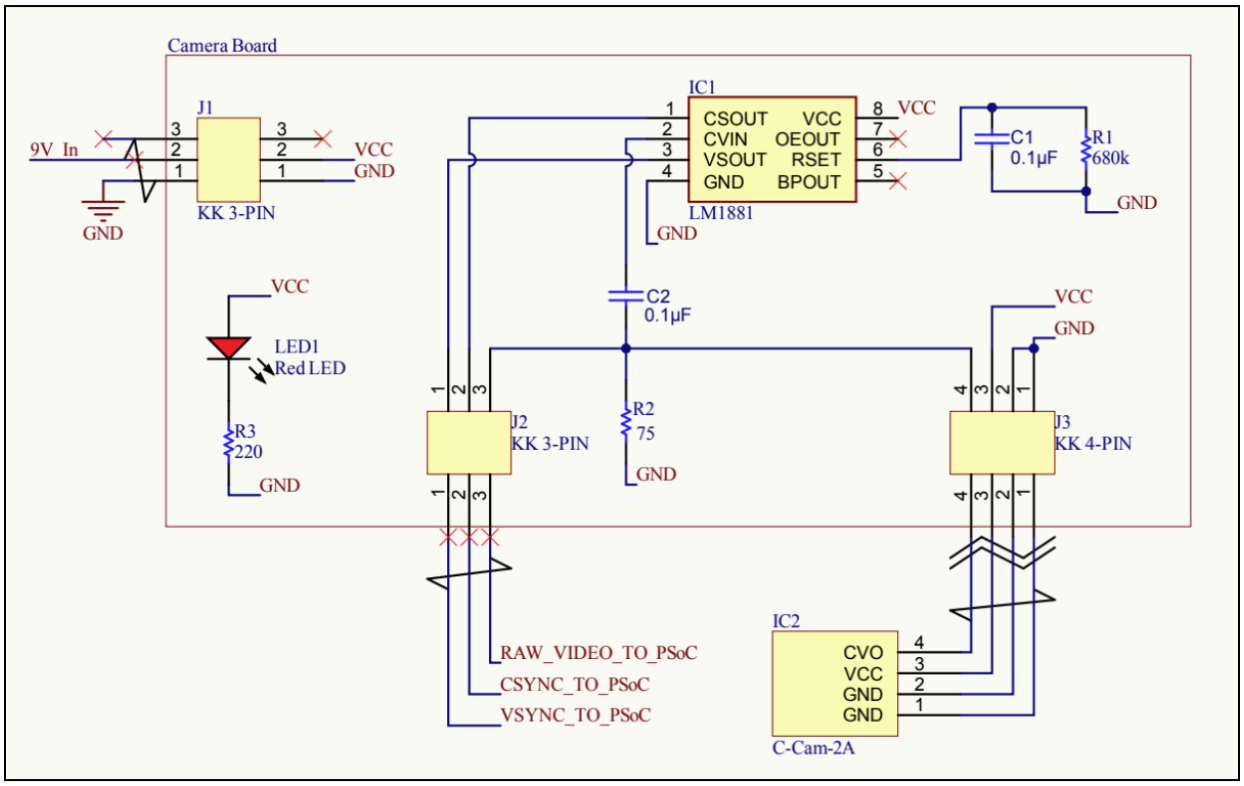
Servo Motor

In order to actually convert our PD control into physical turning, we modulate the PWM signal being sent to the servo. The servo has a 20 ms period, but only the first 2 ms is used by the servo to make changes in angles. Based on how much of the first 2 ms is a high signal vs a low signal, the servo moves from -90^{o} to +90^{o} (from our measurements, 1 ms high is -90^{o}, 1.5 ms high is 0^{o}, 2 ms high is +90^{o}). This allows us to convert from a PWM signal to an angle.

The PWM block for the servo motor runs on a 3 MHz clock with a 60000 unit period (this is 20 ms). Thus, 1 to 2 ms ranges from 3000 to 6000 units. 4500 units means the servo sets to 0^{o}, which corresponds to driving straight forward. The PD control output is centered on 4500 (so if all gains are 0, the car just drives forwards). When the error is negative, control output is less than 4500, and we drive left (below 0^{o}). When the error is positive, control output is more than 4500, and we drive right (above 90^{o}).

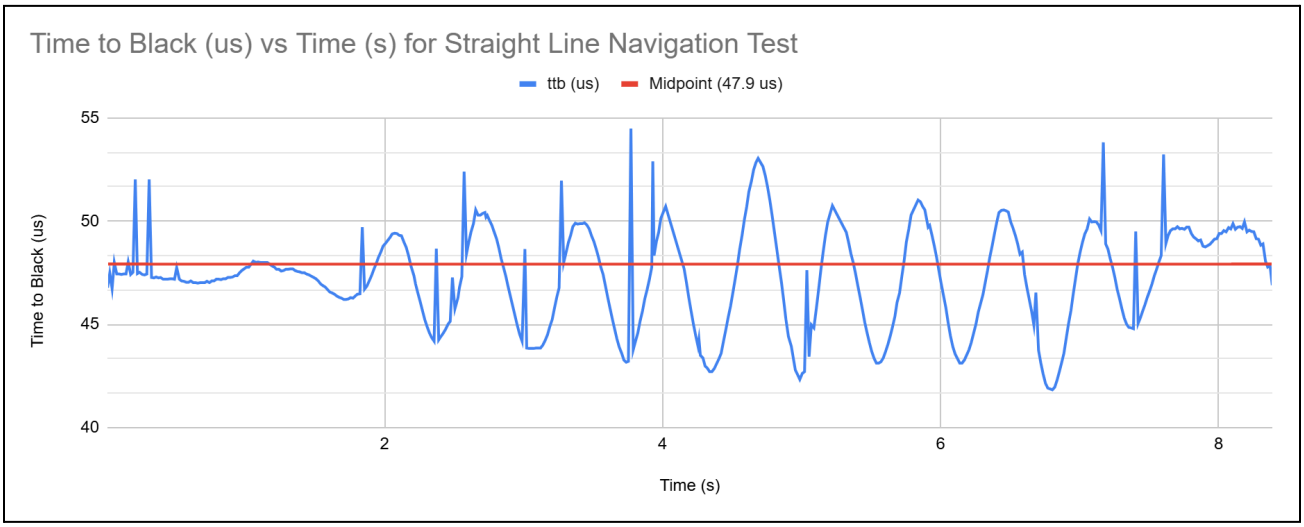
Schematic Diagrams:

Schematic diagram of circuits (computer-drawn, try CircuitLab, etc.) (15 pts.)



Data:

Data showing that it is working as you expect. Include explanations for this. (25 pts.)
Plot showing error (how far your car is from the center) versus time when going along a straight path.



This is a plot of data collected from our XBEE on a straight line, it oscillates slightly as the kd ki hasn't been tuned perfectly, but does not lose the line. The error can be compared to our straight-aligned default value of 1150 / 0.0416 (the period of a 24 MHz clock) = 47.84 us, meaning our oscillations are wiggling ± ~10% on either side of the line. These oscillations were small enough to consistently pass the Navigation course.

There are periodic spikes downwards into strong negative error, which we attributed to bursts of noise getting through the glitch filter. This is actually the lowest amount of noise we managed to get, and was low enough to not disrupt our line following at 3.7 ft/s. When we raised the speed of the car above that, we believe these noisy spikes caused oscillations that were too big to recover from, and we would only finish the course successfully about half the time.

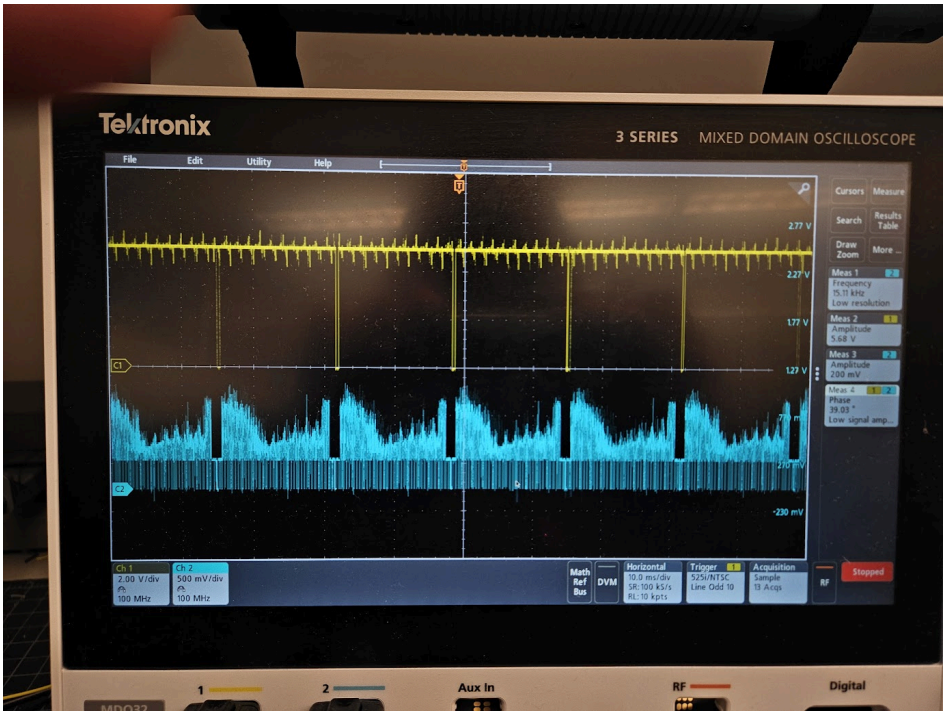
Scope traces:

Raw video & CSYNC

This image shows the raw video footage in Channel 1 (yellow) and the CSYNC signal in Channel 2 (blue). Each CSYNC pulse represents the start of a new line of the camera's video signal.

The actual data of the video signal is everything between CSYNC pulses (not including blanking intervals). This signal is analog with the voltage proportional to the shade of the picture the camera sees for the given line. Darker shades (black) correspond to lower voltages (~400mV), while higher shades (near white) correspond to high voltage (1V). On the oscilloscope we see the voltage waver from left to right across time (temporally), and this represents the way the camera scans each horizontal line from left to right spatially. We placed a piece of paper with a black line in the center under the camera for this image, and you can clearly see each line in the video signal identifies the black line by the Channel 1 voltage dip that aligns near the center of each CSYNC block.

Raw video & VSYNC

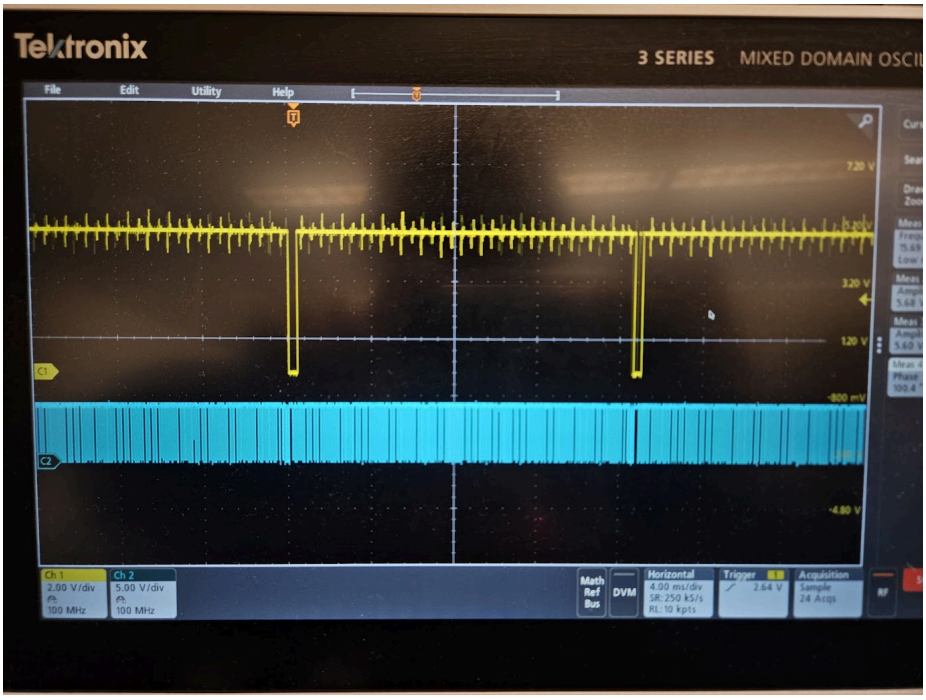


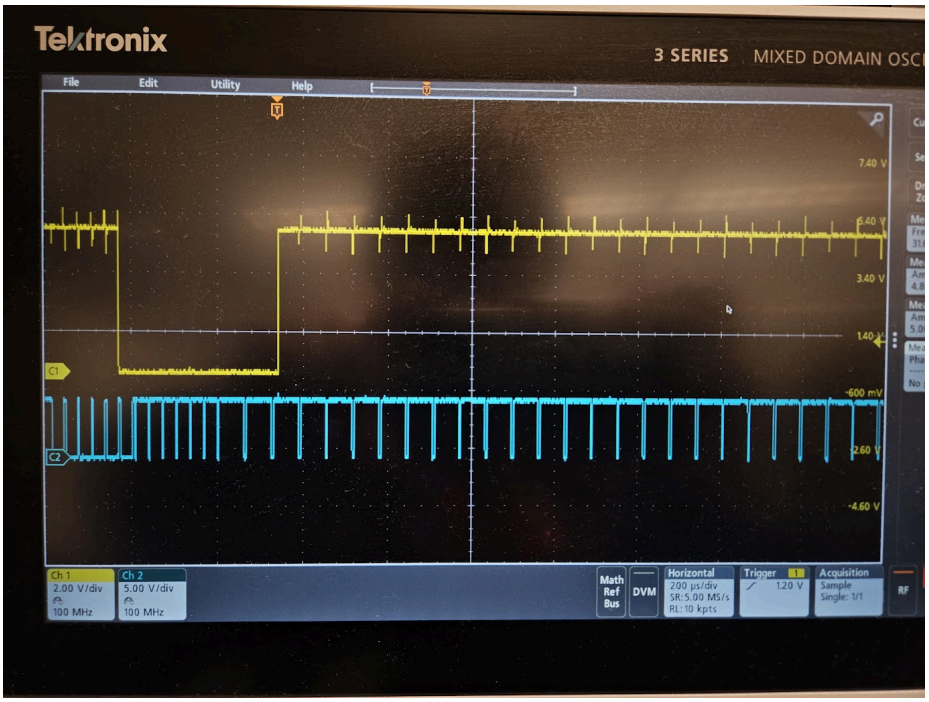
In this image, the raw camera footage signal is in Channel 2 while VSYNC sits in channel 1. Each VSYNC pulse represents a new frame of the footage.

You can see that each chunk delineated by VSYNC pulses corresponds to a burst of camera signal. This burst contains all of the information within a single frame of the video feed. Thus, our system can use VSYNC to identify when a frame has ended and reset our logic for the next frame.

If you were to “zoom in” on the horizontal scale, you would be able to identify many signal bursts like what was shown in Figure 2, because many CSYNC pulses fall within each VSYNC pulse.

CSYNC & VSYNC





These two images both show the same thing at different scales (the first image has a horizontal resolution of 4 ms/div while the second has 200 us/div). This is to make it easier to see what both signals are doing because, as mentioned above, CSYNC and VSYNC operate at very different time scales.

We can see how each chunk delineated by VSYNC pulses corresponds to a large block of CSYNC pulses (the camera datasheet lists the default resolution as 320x240, which would imply that there are 240 CSYNC pulses per every VSYNC pulse).

Using both CSYNC and VSYNC in combination, PSoC can accurately determine where any given part of the signal is in the various packets of raw footage the camera is sending. VSYNC tells us which signals are part of which frame, and CSYNC tells us which signals are part of which line.

Discussion:

Discussion of challenges and design choices you made (10 pts.)

Challenges:

Unlike our project one, project two was plagued from the start with challenges from circuit construction and software strangeness, to hardware-level failures beyond our control. On the circuit side, one of our first issues was not being able to find our VSYNC signal. Initially we were convinced the error was in software, but after debugging and improving our code to a level we thought could not fail, we determined the issue was somewhere else. We decided the board could be the problem so we redid our entire camera board. We checked signals from our hardware and saw it had good outputs, but when we checked the side of the KK connector going into PSoC, we noticed there was an incoherent, gibberish signal. After replacing the KK and wires, the VSYNC was finally being read by PSoC.

After getting a decently tuned car without speed control, the next challenge was combining speed control and navigation. This turned out to be difficult as well, because initially in speed control we had all the logic in the main for loop. This interfered with the triggering of the navigation interrupt, so we switched all of the speed control logic into the Hall Effect interrupt. We also modified the top design to make more sense.

Another curious part of our software was the UART appearing to be too slow. When we printed values, it would take 20+ seconds to display them on the output channel. It turns out that George's account was somehow configured wrong with regards to port inputs (across multiple computers strangely), as the issue was resolved when using the XBEE exclusively with Ryan's account.

The issue that took us the longest to resolve was a strange "jittering" of our wheels that seemed to stay no matter what we changed. We spent many hours and late nights attempting to tune the issue away and eventually had to ask for help. Our car stumped TAs and Professors for hours as we chased leads ranging from the motor introducing signal interference to a broken PSoC, but we could not fix the issue. Over 40 man hours went into the last few days before the due date. Finally, at midnight on Thursday before demo day, Professor Valavi suggested a camera replacement and top design change, and immediately our car worked on the track, passing the demo within the half hour after that switch. We aren't sure whether it was the camera or the top

design change that fixed things, but it was likely either a hidden top design error that we changed when we altered the logic or a faulty connection with the camera somewhere that we missed while redoing the camera board and KKs.

Design Choices:

The design choices for the camera board are fairly straightforward. The camera needs 6V power from the power board, two grounds, and a signal wire. The camera outputs an analog signal that needs to be immediately pulled down by a 75 ohm resistor for impedance matching. Then, the signal is sent to two places: the PSoC directly, and the LM1881 video sync separator chip through a 0.1 microfarad decoupling capacitor on pin 2. The LM1881 needs a capacitor and resistor on Rset pin 6 to set the line scan frequency as specified in the schematic (0.1 microfarad cap and 680k resistor in parallel to ground). Pin 1 and 3 send out two signals to PSoC: the composite sync (CSYNC) and vertical sync (VSYNC).

One design choice we made that changed our top design significantly was triggering our interrupt for finding the line using a timer LineTimer that was normally reset, but not reset when the correct line was found based on a digital comparator using a line counting 8-bit Counter. Then the VDAC would be fed into a comparator and checked with the raw video output (then through a glitch filter) eventually going high on a low enough raw signal value. This signal was sent to the capture pin on the LineTimer which fed the interrupt subroutine. This ensured that the LineTimer was not constantly running and removed the redundant AND gate used in our original design.

Our mast was designed simply to the specifications given by Radd during the training, with through holes and threaded holes and additional rectangular cutouts for the camera wires to be routed through.

Improvements:

What could be improved (10 pts.)

The car sometimes struggles in the dimmer side of the room, losing the line (especially if it sees someone's shadow on that side of the course). At the lower light levels, the average voltage readout from the camera is lower. However, VDAC is a constant term. When we compare them, if the entirety of the raw video output drops by one or two hundred millivolts, that could be low enough to trigger the digital comparator and initiate line following logic. We solved this by carefully tuning the VDAC value on the oscilloscope (we placed the oscilloscope on the ground and measured the tape because the paper line has a slightly different albedo which changes the voltage reading) so that our car could properly distinguish the line and the floor in bright and dim environments.

Theoretically, the line following could be improved even further by adding headlights to the car. This would allow us to choose a VDAC value specifically optimized for the headlight brightness, which would make the car more consistent. By normalizing the light level in the

camera's field of view, the system could be even more effective at line following no matter the environmental conditions.

Additionally, the mast holding up our camera would slightly wobble when we drove, even when the screws were fully tightened. Acrylic is not the stiffest material (especially because we have a single unsupported sheet), and these small vibrations could throw off where our car thinks the line is, which would contribute to noise and oscillations. Potentially, a second panel of acrylic affixed perpendicularly to the mast would add a lot of structural integrity. We could also replace the entire mast with a 3D printed structure that has broad support geometries.

Given more time, we could attempt to perfect kp and kd at higher speeds. The gains we demo'd with work very well at 3.7 ft/s, but at 4+ are too sluggish to make turns. Given more time, we could certainly tune our car to its limits in terms of speed.

Furthermore, we could capture a second line of the camera footage, likely by using a digital comparator to trigger off a second value (say, 80) from the 8-bit counter triggering off CSYNC. With two lines being recorded every frame, we could compare the amount of time to reach the black line between them and accurately calculate the angle of the line. Instead of basing our alignment on a measured standard value (1150), we could use the angle to determine the straightness of our car. This may prove to be more consistent, especially if the measured standard we used is slightly off or changes (camera adjustment).

Adequate comments in your code (5 pts.)
See submitted code ✅