ELE 302 – INDEPENDENT PROJECT WRITE-UP

TJ Smith Byung-Cheol Cho
(Bench 207)

Due May 19, 2017

1 Overview

The aim of this project was to build a robot capable of playing basic ping pong. Prior to beginning this project, we established four ideal objectives for the end product:

1. Track a ping pong ball, estimate its trajectory and predict the location where the ball will land
2. Determine (1) when it is the robot's turn to hit the ball and (2) when the ball will hit the net or leave the playing area
3. Move the robot to the required location before the ball bounces twice and hit the ball back over the net
4. Avoid leaving the playing field (area enclosed by table and net)

By Demo Day, we had implemented the hardware and software to achieve objectives 1, 3 and the first half of objective 2, and we had the capability of achieving the remainder of the objectives had we decided to implement them in software.

We used two Pixy cameras (CMUcam5) placed a fixed distance apart on the robot to triangulate the location of a bright red ball in three-dimensional coordinates. Once the camera system had collected enough data points to accurately predict all future positions of the ball until the second bounce, the robot moved to a location where the ball would bounce to the center of a paddle attached to its side, and provided a brief flick to the paddle to return the ball.

The speed and flexibility of movement we needed (we had to move up to 3 feet in less than half a second) required us to abandon the chassis used for our previous speed control and navigation assignments and adopt the large and powerful omni-wheel drive used by Ethan Gordon and Luke Pfleger in 2016. We used an accelerometer and gyroscope for dead reckoning of position (we could no longer use wheel rotation tracking because of slippage). In addition, we continued to use the PSoC 5LP because of its extensive motor control capacity (we required four PWM modules) while adding a Raspbery Pi 3 because of the intensive computation we expected to require for ball tracking and trajectory estimation.

2 Theory

2.1 Trajectory estimation

2.2 Omni-drive

3 Hardware

The movement requirements of our objectives

Figure 1 shows a block diagram of the hardware set-up pertinent to autonomous navigation (see previous write-up for the hardware for speed control, such as the motor board, main driving motor, Hall effect sensor and Hall effect sensor board).

Figure 1: **Block diagram of the hardware setup for navigation.** Black connections indicate ground connections; red connections indicate power (both regulated and unregulated); all other connections indicate data wires (e.g. from the Hall effect sensor board, to the steering servo and the motor board, and from the camera board). Green components indicate power-related components; red components indicate actuators; blue components indicate sensors.

3.1 Steering servo

The steering servo was a Futaba FT-S148, which uses a pulse-width modulated control signal to internally control the motor to achieve the final output angle. From bench testing, we determined that a pulse width of approximately 1.00 ms corresponded to a full left turn, 1.75 ms corresponded to a full right turn, and the mean of 1.375 ms corresponded to driving straight.

The motor had sufficient torque and the car had sufficient turn radius to execute all the turns in the course (the tightest turns were 3 ft in radius, while the car was capable of turning with a radius of 2 ft to 2.5 ft). However, we found that the car had a larger turn radius turning left than turning right, so we slightly increased the range of pulse width times to enable the car to turn enough for the tight left turns, and made some software modifications in an attempt to balance the car's physical imbalance between left and right turns. In addition, we found that there was an approximate 0.5 s delay for the car to transition from a full left turn to a full right turn, another possible source of instability and inaccuracy at higher speeds.

The steering servo is known to draw a lot of current when the motor is running, so it was connected to a separate voltage regulator on the same 9.6 V battery line to avoid drawing current away from the camera, camera board, and Hall effect sensor board. Regulated 5 V voltage output headers were added for the camera board and the servo motor.

3.2 Camera and mast

The camera was a C-Cam-2A, a small CMOS video camera that outputs 260 TV lines interlaced 2 : 1 at 30 frames per second. This corresponds to about 16.7 ms per frame, and about 64 μs per horizontal TV line. To avoid confusion, from this point on, output rows of the camera will be referred to as *rows*, while the black line we are following will be referred to as the *line*.

The camera was mounted on a transparent acrylic mast on the front of the car (see Figure 2) so that the line immediately in front of the car could be seen by the camera. Because of the slight angle at which the camera was mounted on its circuit board, we had to tilt the camera slightly on the mast so that the line appeared at the center of the frame.

The mounting of the camera provided us with an additional parameter for line detection: the angle at which the camera points down at the ground. Pointing the camera to look farther out made control generally better because it helped eliminate oversteering and steering overcorrections that tended to limit high speed following. This is possibly because this effectively contributes to a form of derivative control of line position error: we can effectively predict the line position error when a turn approaches and begin turning in response; similarly, we can predict the straightening out of the line and stop turning so much in response.

3.3 Camera board

We used a video sync separator and a simple comparator circuit on a circuit board to help parse the video signal (see Figure 3 for details of the camera board). The board performs three functions: **(1)** provide power to the camera, provide the 75 Ω load expected by the camera, and pass on the video output to the PSoC; **(2)** detect the beginning of the frame (*vertical sync*) and the beginning of each camera row (*horizontal sync*) and

Figure 2: **Camera mounted on mast.** Labels indicate the mast, the camera and the cable connecting the camera to the camera board.

pass these signals on to the PSoC; **(3)** compare the value of the video signal with a hard-wired threshold to detect, in hardware, the position of the line in each camera row.

There is the usual power indicator light (with a 470 Ω resistor in series) connected to the input 5 V power supply. The camera and video sync separator (LM 1881) power lines have 0.1 μF decoupling capacitors; the video input to the sync separator also has a 0.1 μF de-noising capacitor, and the video line has a 75 Ω pull-down load resistor. The R_{SET} pin is pulled down to ground with 0.1 μF and 680 kΩ in parallel to set the correct timing references for the horizontal and vertical sync separation.

The LM 311 (a high-speed differential voltage comparator) compares the raw video output from the camera with a threshold voltage set using a trim potentiometer. The output of the comparator is an open-collector output, so a 2 kΩ resistor was used as a pull-up resistor. A higher resistance could not be used because the capacitances in the circuit would otherwise cause the time constant of the comparator response to be too slow.

3.4 Programmable hardware

Figure 4 shows the top-design of our programmable hardware in PSoC Creator 2.1.

3.4.1 Modifications from speed control

We renamed our old programmable hardware blocks to make their functions more obvious. Thus the counter that is used for stall detection is now called `Counter_Tick_Timeout` instead of `Counter_1`, and so forth for the rest of the hardware. All of the old hardware is in the top-left of Figure 4. In addition, we have added a new PWM block `PWM_Servo` for controlling the servo motor; since this runs off the old clock with a frequency of 100 kHz, we placed it with the old hardware blocks in the top left corner.

(a) Photograph

(b) Schematic

(c) Circuit diagram

Figure 3: **Circuitry of the camera board.**

3.4.2 Hardware for camera signal processing

The camera data arrives much more quickly than our Hall effect sensor data (60 frames/s and 260 rows/frame, making 15 600 rows/s), so we had to use a much faster, 12 MHz clock to run all of our programmable hardware for the camera.

The signals we receive from the physical hardware are the vertical sync, the horizontal sync, and the comparator output. The vertical sync is high during the frame and low between frames, the horizontal sync is high during the row and low between rows, and the comparator output is (theoretically) high for the line and low for anything else (specifically, when the video signal drops below the hardware threshold, the comparator output goes high).

We detect both the beginning and end of the line (rising and falling edges of the comparator, respectively), using two timers, `Timer_Line_Begin` and `Timer_Line_End`. The timers are on the 12 MHz clock, and are reset by the rising edge of the horizontal sync. Since the reset input is level-based, we implemented a rising edge detector (the NOT gate, D flip-flop, and AND gate between `Counter_Near_Row` and `Timer_Comp_Ignore_1`) to ensure proper resets. Both timers are 16-bit to ensure overflow does not occur (a 12 MHz clock and 15 600 rows/second give roughly 770 clockcycles/row, so 16 bits is more than enough). The only difference between the two timers is that `Timer_Line_Begin` captures on the rising edge of the comparator output, and `Timer_Line_End` captures on the falling edge. Both timers interrupt on capture.

We initially tried to find the line in every row of the camera data, but soon found that the processor could not handle that volume of interrupts. To remedy this problem, we added two counters, `Counter_Far_Row` and `Counter_Near_Row`, which select out a single near (close to the car) row and a single far (far from the car) row from the 260 total rows. The way our camera is mounted, it scans far to near (top to bottom), so the farthest row is the first row and the closest is the 260th. The camera sends synchronizing pulses on either side of a complete frame, which look like half length rows with no data. The vertical sync goes high in the middle of these synchronizing pulses, so we get several "false" horizontal syncs before the actual

rows of the camera start coming. Since it doesn't matter at all that we get the farthest row possible, we just take the 20th row as the far row to make sure we avoid this. We take the near row to be the 240th row.

Each counter is reset continuously when the vertical sync is low, so the counters begin counting once it goes high (i.e., at the beginning of the frame). The vertical sync is much slower than the clock (60 Hz vs. 12 MHz), so we don't need any kind of edge detector. The counters count the rising edges of the horizontal syncs, so they simply count rows. `Counter_Far_Row` is set to compare equal to 20, so its compare output is high only during the 20th row. `Counter_Near_Row` is set to compare equal to 240, so its compare output is high only during the 240th row. These two outputs are OR'ed together, and this is then AND'ed with the comparator output. This effectively makes the comparator output active only during the 20th and 240th rows. `Timer_Line_Begin` and `Timer_Line_End`, then, can only interrupt during these two rows, bringing the number of interrupts per frame down to only 4 instead of 520, which is very manageable. We leave the task of distinguishing between the near row and the far row to software.

The final problem we found (that we resolved in hardware) was that each row has a small buffer region on either side of the actual data. This buffer region is at a flat value equivalent to what the camera would output in full dark. This section is below the comparator's threshold, so the comparator starts high. As soon as the actual camera data starts (assuming the case where the line is not at the edge of the row), the comparator goes back low, causing a false falling edge. Similarly, the end of the row causes a false rising edge. To fix this problem, we use another two timers, `Timer_Comp_Ignore_1` and `Timer_Comp_Ignore_2`, to create a window within the row that selects out only the valid comparator values. Both timers are reset by the rising edge of the horizontal sync (the beginning of the row), and are on the 12 MHz clock. `Timer_Comp_Ignore_1` has a period of 150, and `Timer_Comp_Ignore_2` has a period of 600. Since each row is roughly 64 μs, and the clock frequency is 12 MHz, there are about 770 clock cycles per row, so it is clear that these bounds are near the beginning and end of the row.

The terminal count signal from the timer only goes high for a single clock cycle, so we created a hold element to create the actual window from the two terminal count outputs (the two OR gates, NOT gate, and D flip-flop near the middle of Figure 4). The clock input on the flip-flop will go high on row start, `tc1`, and `tc2`. On the row start pulse, the `d` input will be low, since row start is high, resetting the window. On the `tc1` pulse, `d` will be high, since neither row start or `tc2` are high. This pushes a one into the flip-flop, starting the window. On the `tc2` pulse, `d` will be low, since `tc2` is high. This pushes a zero into the flip-flop, ending the window. This window is AND'ed together with the comparator output, in the same way as the above row selectors, such that the input to `Timer_Line_Begin` and `Timer_Line_End` is now only the valid part of the comparator data. This ensures that without any other noise in the image (like another line), the only interrupts that are triggered are the rising and falling edges of the line on the near and far rows.

4 Software

4.1 Modifications to speed control code

We made several modifications to the structure of our speed control code. In particular, we modularized all of our code to make management easier. All of our serial code is in files `serial.c` and `serial.h`; all of our speed control code is in `speedcontrol.c` and `speedcontrol.h`; `hardware.h` contains all of our external hardware-based constants, including parameters from programmable hardware such as clock frequency or periods of timers, and external constants such as the wheel diameter. The only function left in `main.c` is `parseMessage()`, the function that parses the received serial messages and calls the relevant functions. This was left in `main.c` because it needs to be able to call functions from every other module.

The only substantial change we made to the old code (besides modularizing) was to allow our speed control serial data dump to be toggled via serial command. When navigation is running, it requires very precise timing, and large serial data interfered with this timing.

All of our new code is located in `camera.c`, `camera.h`, `navigation.c`, and `navigation.h`. `parseMessage()` in `main.c` has also been extended to allow for control of all our new functionality.

4.2 Steering control

We control our steering servo with a pulse signal of period 20 ms. The width of the pulse nominally should vary from 1 ms to 2 ms (giving full left and full right, respectively), but we found our true limits to be approximately 1.00 ms and 1.75 ms. Despite this shrinking of range, it maintained quasi-linearity, such that the middle position was still simply the average of these two times. `PWM_Servo` is controlled by the function `Navigation_setSteering`. Most of the function is just performing the required math to scale an easy input (in the range $[-1, 1]$) to the actual output that `PWM_Servo` requires. However, it also scales every negative input (i.e. left-turning input) by a constant (1.15); this is in an attempt to counterbalance the imbalance in our car between left and right turns: our car turns more sharply right than it does left, so unscaled, an input of -0.5 will not cause the car to turn as much to the left as an input of 0.5 will cause it to turn to the right. Of course, at full range (-1 or 1), the car will still turn more sharply to the right than to the left, but this helps in all other cases.

4.3 Camera interface

All of the code to process the camera data is contained in `camera.c` and `camera.h`. `Camera_handleCompRise()` handles the comparator rising (line beginning) interrupt. It simply reads the number of clock cycles from the beginning of the row until the beginning of the line from `Timer_Line_Begin`, and then reads the value of `Counter_Near_Row` to determine whether the interrupt was on the near row or the far row (`Counter_Far_Row` could have been used just as well, as the count values of the two are identical).

`Camera_handleCompFall()` handles the comparator falling (line ending) interrupt, and performs all the final calculations. Like `Camera_handleCompRise()`, it gets the number of clock cycles from row beginning until line end, and figures out which row it is looking at. Then, since we have reached the end of the line, it uses the recorded value of the beginning of the line from `Camera_handleCompRise()` to calculate the width of the line in clock cycles. We experimentally determined the true width of the line in clock cycles for both the near and far rows, and if the measured width does not fall within $\pm 10\%$ of the expected width, `Camera_handleCompFall()` rejects the measurement (note that this requires recalibration whenever the camera angle is changed). Otherwise, it normalizes the location of the midpoint of the line to $[0, 1]$, and stores it in `lineMidNear` or `lineMidFar`, as applicable. We track the number of times in a row we have gotten a bad measurement, and if it passes a set value of misses (indicating loss of the line), we can return an error code (in functions `Camera_getLineMid()` and `Camera_getLineAngle()`. We do the actual error processing in static functions `checkLineMidNear()` and `checkLineMidFar()`). We also check periodically (every 20 ms) to make sure that we have received an interrupt in that time. If we have not, again, we can return an error code. Our navigation code can use the error codes to turn the car off. During most of our testing, we left both these features disabled, as we could kill the car manually, but it would be an important feature for more autonomous applications.

With the calculations of `lineMidNear` and `lineMidFar`, we can calculate a position and an angle of the line relative to the car. The position of the line is simply `lineMidNear`, and the angle (or rather something proportional to small angles) is simply the difference `lineMidNear - lineMidFar`.

4.4 PID control

Our PID control for steering uses the same down-counter `Counter_PID` as for speed control. We updated the period to 0.02 s because controlling steering at high speeds required much faster PID update commands. The interrupt triggered by the counter calls both `SpeedControl_handleTimer()` and `Navigation_handleTimer()`.

The camera also has some variables for error detection (see previous section) that need to be reset periodically, so we also call `Camera_handleTimer()` when the PID counter interrupts.

The PID controller for navigation is very similar to the controller for speed control. The biggest difference is that it controls based on two variables: the line position (x position) and the angle (θ). The function `controlSteering()` receives normalized line position (as a fraction between 0 (left) and 1 (right)) and angle (between -1 and 1 where 0 is parallel to the line and positive means a counter-clockwise rotation of the car is required to return to parallel). Invalid values (such as `linePos < 0` or `theta < -1`) will indicate that something is wrong and that the car should stop: both navigation and speed control are killed in this situation.

For line position, we compute the error from the target line position of 0.5. The implementation of derivative and integral control are essentially identical to that of speed control: we keep a ring buffer and compute the average derivative with

$$\frac{\text{current error} - \text{oldest error}}{\text{buffer size}},$$

and we also compute the integral error with a simple cumulative sum. The code we wrote for PID control also included a double integral term (which was a cumulative sum of the integral error), making this really PI^2D control; we found no need to use either integral terms in our testing and parameter tuning.

For angle, we compute the error (which is simply $-\theta$). Just as before, we maintain a ring buffer and compute the derivative and integral terms. We found no need for the integral term for this term either.

We then compute the new steering position based on our line position error, angle error, and all the associated $\text{PI}^{(2)}\text{D}$ terms in a direct sum over all seven terms. We considered different control methods that would implement the angle error more effectively, but because of time constraints we were not able to explore this further.

We also later implemented a method to adjust the target speed using `SpeedControl_setTargetSpeed()` so that we could drive faster on straight parts of the track and slow down so that we do not lose the line on tight curves.

The final part of the requirement for the navigation project was to turn off the car once it had driven two times around the track. We did this by calling the `SpeedControl_enableDistanceControl()` function that we had already implemented in the speed control assignment

4.5 Serial communication

We continued to use serial communication extensively to debug the camera and navigation portions of our program and to adjust our PID variables. We also moved to using the DB9 header only for bench testing and a wireless XBee serial communication chip for track testing. We changed nearly no code or programmable hardware to move to the XBee, since it relied on the same UART communication as the DB9.

We heavily updated our list of serial commands to improve memorability and consistency, and to expand to control camera and navigation parameters. Our updated commands are listed in Table 1.

5 Results

See Table 2 for the final PID coefficients we used. We updated our coefficients from speed control because we are no longer dynamically updating the target speed to reach the distance in a specific time. We used only the proportional and derivative terms for line position and angle (no integral terms), and unfortunately, despite our efforts to calculate it accurately, the angle contributed very little to our control. Given more time, we may have been able to develop a better system of control to more robustly control steering at higher speeds.

Command	Description
CTx	Change throttle
GS	Get all variables associated with speed control
TS	Toggle speed PID control
TDC	Toggle distance timeout for speed control
TDS	Toggle dynamic speed control
CPSx	Change proportional term for speed control
CISx	Change integral term for speed control
CDSx	Change derivative term for speed control
CSSx	Change steady-state throttle for speed control
CTSx	Change target speed for speed control
CTDx	Change target distance for speed control
TVS	Toggle verbose printout for speed PID control
GC	Get all camera variables
RC	Reset camera variables
CSx	Change steering/servo direction
CMMx	Change maximum permitted line misses
TN	Toggle navigation PID control
CPLx	Change proportional term for line position error
CILx	Change integral term for line position error
CIILx	Change double integral term for line position error
CDLx	Change derivative term for line position error
CPTx	Change proportional term for angle error
CITx	Change integral term for angle error
CDTx	Change derivative term for angle error
CTSNx	Change nominal target speed for navigation
TLE	Toggle line error tracking
TVN	Toggle verbose printout for navigation PID control
A	Abort (kill speed and navigation PID control)

Table 1: **Table of serial interface commands.**

See Table 3 for our final times for one lap. The highest speed we managed to achieve was with a nominal 6.0 ft/s, though by reducing speed while turning corners, our average speed was only about 5.4 ft/s. We were unfortunately unable to consistently replicate this speed, and we performed our demo with a nominal target speed of 4.0 ft/s.

	P	I	D
Speed control	80.0	1.0	10.0
Line position	3.0	0.0	2.0
Line angle	0.5	0.0	0.0

Table 2: **Final PID coefficients**

Nominal target speed	Time for one lap	Average lap speed
4.0 ft/s	28 s	3.8 ft/s
5.0 ft/s	22 s	4.9 ft/s
6.0 ft/s	19.96 s	5.4 ft/s

Table 3: **Time trial results**

Figure 4: **Top-design of the navigation project.**

6 Appendix: full listings

Contents

6.1 `main.c`