DAVEY JONES' RACECAR

Caitlin Barron, Chris Guarini

Rochester Institute of Technology

***Abstract*- Every year Rochester Institute of Technology holds a small race for autonomous vehicles during its annual festival of engineering and innovation. The cars are programmed by students to follow a clear track, but success doesn't come easily. Teams must overcome everything from simple hardware problems and lighting variations, to working with sensitive code. There are also many opportunities for sudden and unexpected problems that will test their problem-solving skills. Overall, this project serves as a great hands-on learning experience for these computer engineers in training.**

Index Terms - NXP Cup, RIT Cup, Racecar, Autonomous Driving

1. INTRODUCTION

Computer engineering is a relatively broad field, spanning many topics in both electronics and programming. This allows these engineers to work on a wide variety of projects. One of these many possible projects, is the building and programming of autonomous vehicles. In this paper, the vehicle in question is designed to follow a white track with clear black lines as boundaries. The track can also include more complex obstacles like intersections and ramps. This is fairly simple compared to the current developments in full scale autonomous vehicles, but still proves to be a great exercise in developing a solution to a real-world problem.

One interesting aspect to this competition is that each team took a different approach to solving the problem. This means that no team performed the same and made the race even more interesting. of the 11 teams competing, with one team performing in exhibition, all but two were able to complete at least one lap. Times ranged from 25 seconds to just under a minute. Most seemed to be racing at a very high speed and relied on things like drifting and variable speed to keep them on the track. Some teams stuck with the "slow and steady wins the race" technique. This paper describes the design of the racecar from team "Fast XOR Furious", as seen in *Fig. 1* above.

Fig. 1. The racecar from team "Fast XOR Furious"

The code running the cars mainly differed in their steering logic, which used the signal from the camera to detect where on the track they were, and therefore, which way to turn to stay on it. This could be done either by taking the integral of the signal to more clearly find the edges of the track, or simply splitting the signal down the center and using the size of the signal on either side to decide the turn. Each team also needed to find constants to use for speed and turning. These values differ between teams due to slight hardware differences. These relatively small differences greatly affected performance though, adding yet another challenge for students. The authors of this paper chose to use the latter of these two methods

The NXP cars also had physical differences, from slightly different 3D printed parts, or decorations. One of the main challenges for the authors came when, during

practicing on the track the night before the competition, a stack of track sections fell onto the car and broke critical parts. With some help, these parts were fixed or replaced in time for competition. Team "Fast XOR Furious" also chose to decorate their car like a pirate ship and were the only team to add decorations of the sort.

The remainder of this paper is organized as follows. After the introduction, Section 2 overviews the project background. Section 3 introduces methods used in this project. Section 4 presents the experimental results. Section 5 contains concluding remarks, and Section 6 presents the references used.

2. BACKGROUND

The NXP car utilizes several key concepts that were covered in class and used in other laboratory exercises.

The Proportional-Integral-Derivative controller, or PID controller, is used to correct for errors by using a simple feedback system. This uses the *Eq. 1* below.

$$\boldsymbol{R(t) = Kp\left[e(t) + \frac{Kp}{T_i}\int_0^t e(t)dt + Td * \frac{de(t)}{dt}\right]}$$

Eq. 1. The full PID equation

This equation uses current error values, the past error values, and the possibility of future errors to improve output. It is designed to quickly get the car up to max speed and then reduce oscillations, resulting in smoother performance.

The speed of the vehicle was also very important. To win the race, the goal is to go as fast as possible, but this is easier said than done. As discovered during testing, building up too much speed on straight portions on the track makes it harder for the camera to detect turns and also can simply build up too much momentum and go sliding off the track. Going to slow also has issues though, as going to slow would result in stalling when going over ramps. The final plan was to increase the car's speed on straightaways but decrease speed for turns. Logic was also created to vary how much the car slowed for turns depending on how sharp the turn is.

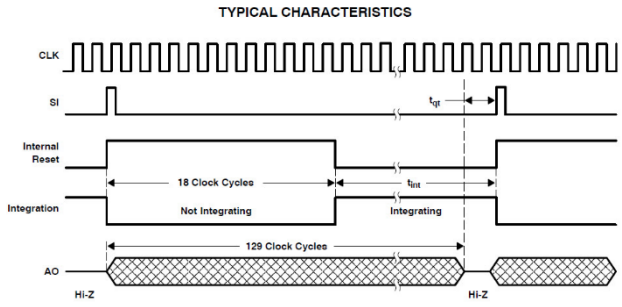
One crucial part of the projects was properly using the Flex Timer Module, or FTM. This was used to generate Pulse Width Modulation, or PWM signals, on interrupts that were used to control the DC motors driving the wheels.

3. METHODOLOGY

3.1. Autonomous sensors

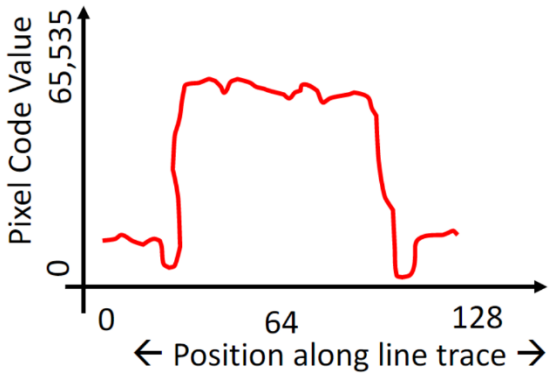
The primary sensor used by every car in the NXP Cup competition is the line scan camera. As described in [3], the camera uses a TLS1401CL sensor to detect the amount of light hitting each of the 128 detectors arranged in a horizontal line. The camera is utilized by the microcontroller through the signals shown in *Fig. 2* where PIT0 sends the SI pulse that triggers camera integration and enables FTM2 to read the line from ADC0, which is connected to the AO pin. The line is stored by the microcontroller as an array of uint16_t that is 256 bytes long (128 16-bit integers).

Fig. 3 shows the general plateau shape output by the camera when the car is centered on the track. The top of the waveform corresponds to the track, the valleys on each side are the black edges of the track, and beyond those is the surface that the track sits on top of.



Source: [2]

Fig. 2. Typical Characteristics of the Line Scan Camera



Source: [3]

Fig. 3. Output of the line scan camera when the car is centered on the track.

Once the microcontroller has read the line from the camera, the function steeringFunction() is used to process the line. *Fig. 4* is a snippet of the file *steering.c* showing the full function. The code is actually quite simple, it loops through the line provided by the camera adding up all values less than the halfway point into the variable left, and all the values greater than the halfway point into the variable right, while also excluding the edges as the pixels on the edge tend to have the most noise. This effectively integrates the left and right side of the line scan. The code then divides the integral of the left side by the integral of the right side, which provides a relative position of where the car is on the track.

Using this integration method, a car in the center of the track will have an equal area under the left and right sides of the line, meaning that the steering factor will be equal to 1. As the car moves left on the track, the numerator gets larger which means that the steering factor becomes greater than one. As the car moves right from the center, the denominator gets larger, which causes the steering factor to be smaller than one but never less than zero.

While there are other ways to find the position of the car on the track, namely using derivation instead of integration, this method was chosen due to a couple of reasons:

A. The effect of noise between pixels is reduced, as 58 pixels are integrated on each side, so a pixel that is over reporting light intensity can be offset by one of the 57 other pixels under reporting its light intensity.
B. The effect of the floor beneath the track is reduced, as the highly reflective track has more impact on the steering factor than the less reflective floor beneath the track.
C. The steering factor is easily used by the control function to determine servo position and motor speed.
D. Intersections are easily handled, as the left and right side of the line should be generally equal, so the car goes straight through without the need for any extra logic.

```
138 void steeringFunction(uint16_t line[128], int MAX_PWM, int TURN_PWM){
139
140   //Values that hold the integration of the left and right hand of the camera
141   int left = 0;
142   int right = 0;
143
144   //Loop through the line output by the camera, integrating the left and right hand sides
145   for(int i = 6; i < 123; i++){
146
147     //Lefthand integration
148     if(i < 65){
149       left += line[i];
150     }
151     //righthand integration
152     else{
153       right += line[i];
154     }
155   }
156
157   //Divide the left by the right, resulting in a factor that will be 1 when they are equal,
158   // <1 when left is larger and >1 when right is larger.
159   float steeringFactor = ((float) left) /((float) right);
160
161   //Call the helper function Steer() to determine steering based on the steeringFactor.
162   Steer(steeringFactor, MAX_PWM, TURN_PWM);
163
164 }
```

Fig. 4. The code for the steering function, which processes the line returned by the camera to determine the car's position on the track.

3.2. Control theory

Once the steering factor has been calculated from the line scan, the Steer() function is called to control the servo and motors in order to steer the car. *Fig. 5* is a snippet from *steering.c* that shows the first part of the Steer() function, which ramps up the speed of the car after initialization and hard turns. The rampUp variable is a global float that is bounded between .25 and 1. For every call of the function, the variable is increased by 1% of the steering factor, which is used to pace the car so that it is able to straighten out after startup and hard turns.

```
56    //Initialize value
57    if (rampUp == 0){
58      rampUp = (float) (.5);
59    }
60
61
62    //Ramp up speed as car goes straight
63    if(rampUp < 1.0){
64      rampUp += (float) ((steeringFactor )* .01);
65    }
66    //Cannot be larger than MAX_PWM
67    if(rampUp > 1.0){
68      rampUp = 1.0;
69    }
70    //Cannot be smaller than 1/4 TURN_PWM
71    if(rampUp < .25){
72      rampUp = .25;
73    }
```

Fig. 5. Code that increases the speed of the car for every call to the Steer() function.

Fig. 6 is the section of the function that handles the variable speed during turns. TURN_PWM is a constant defined in *Constants.h* that sets the minimum motor PWM that the car takes a turn, and thus the minimum PWM of the car. As the name suggests, MAX_PWM is a constant that defines the maximum PWM of the motors. The code takes the difference between the maximum and minimum, multiplies it by the steering factor divided by two. It then adds this value to the minimum PWM to determine the PWM to set both motors. For steering factors larger than 1, it flips the factor around 1. For example, a factor of 1.3 is turned into .7. This keeps the speed consistent between left and right turns, and makes sure that the speed always decreases as the car moves away from the center.

```
76    // Variable speed depending on how far from center of
77    //track, Straighter goes faster
78    if(steeringFactor > 1){
79      maxSpeed = (float) (TURN_PWM + ((MAX_PWM - TURN_PWM)
80        * ((((steeringFactor - 1) * -1) + 1) * .5)));
81    }
82    else{
83      maxSpeed = (float) (TURN_PWM + ((MAX_PWM - TURN_PWM)
84        * (steeringFactor * .5)));
85    }
86
87    //Apply the rampUp variable
88    maxSpeed = ((float) rampUp * maxSpeed);
89
90    //apply maxSpeed to both motors
91    right = maxSpeed;
92    left = maxSpeed;
```

Fig. 6. Code snippet of the Steer() function that decreases motor speed variably by how far away the car is from the center of the track.

Steering the servo is handled by the code in *Fig. 7*, which implements a proportional control algorithm. Following the code in order of operations, 7.25 is the duty cycle that keeps the servo straight. The "center" of the steering factor is moved to 0 by subtracting 1. This means that the steering factor can now be either positive or negative depending on which side of the center the car is on. The range of the servo is from 5.25 to 9.25, so 2 is used to represent that range. 1.75 acts as a K_p of sorts, which provides a gain to the steering factor so that it provides a notable difference in the servo's duty cycle. The sign of this value is flipped through multiplying by -1, which was required due to the value being negative when the car was too far left, when it should be negative when the car is too far right. The value is then added to the 7.25 in order to find the position of the servo.

```
94    //PID servo steering
95    //7.25 is straight, applied when steeringFactor ~1
96    //Variable from 3.75 to 10.75
97    dutyCycle = 7.25  + ( -1 *(2 * 1.75
98    *  (((steeringFactor - 1) ))));
```

Fig. 7. Code that handles the PID steering of the servo.

If the car is taking a sharp turn or is far too close to the left or right side of the track, the code in *Fig. 8* will cut the corresponding motor in order to help with steering. This has the added benefit of causing the car to drift during sharp turns at high speeds. The code simply reads the steering factor, and if it is larger than 1.6 or less than .6 it will cut the left or right motor respectively and set the rampUp variable to .5 so that the car has to take some time to return to full speed. This code also keeps the car straight if the steering factor is between .95 and 1.05, which is a threshold to keep the car straight so that it doesn't wobble with small differences in the left and right side of the line integration.

```
108   //Handle drifting to the right
109   if(steeringFactor < .6){
110     //hard right turn
111     right = 0;//cut right motor
112     left = TURN_PWM;//left motor full turning speed
113     //Drop speed during hard right turns,
114     //will have to ramp up back to full speed
115     maxSpeed = TURN_PWM;
116     //Ramp back up to speed after a hard turn
117     rampUp = .5;
118
119   }
120
121   //Keep straight, controls for noise in the camera
122   if(steeringFactor < 1.05 && steeringFactor > .95){
123     dutyCycle = 7.25;
124   }
125
126   //Handle drifiting to the left
127   if(steeringFactor > 1.6){
128     //hard left turn
129     left = 0;//Cut left motor
130     right = TURN_PWM;//right motor full turning speed
131     //Drop speed during hard left turns.
132     //Has to ramp up back to full speed
133     maxSpeed = TURN_PWM;
134     //Ramp back up to speed after a hard turn
135     rampUp = .5;
136   }
```

Fig. 7. Code that handles drifting and staying straight.

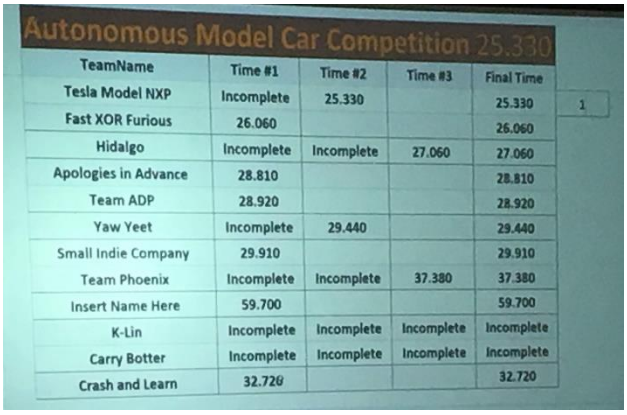
Finally, *Fig. 8* shows the last bit of code in the Steer() function, which makes calls to the SetDutyCycle() and SetServoDutyCycle() functions to set the motor speed and servo position respectively.

```
138   //Set servo and motor values
139   SetServoDutyCycle(dutyCycle, 50, 0);
140   SetDutyCycle(left, right, freq, dir);
```

Fig. 8. Code that uses the values found previously in the Steer() function to set the servo position and motor speeds.

4. RESULTS

The final outcomes of the NXP Cup at Imagine RIT can be seen below in *Fig. 9*.



Autonomous Model Car Competition 25.330

TeamName	Time #1	Time #2	Time #3	Final Time
Tesla Model NXP	Incomplete	25.330		25.330
Fast XOR Furious	26.060			26.060
Hidalgo	Incomplete	Incomplete	27.060	27.060
Apologies in Advance	28.810			28.810
Team ADP	28.920			28.920
Yaw Yeet	Incomplete	29.440		29.440
Small Indie Company	29.910			29.910
Team Phoenix	Incomplete	Incomplete	37.380	37.380
Insert Name Here	59.700			59.700
K-Lin	Incomplete	Incomplete	Incomplete	Incomplete
Carry Botter	Incomplete	Incomplete	Incomplete	Incomplete
Crash and Learn	32.720			32.720

Fig. 9. The final times from the NXP Cup

As seen in the image above, team “Tesla Model NXP” took first place with a time of 25.330 seconds. The authors took second place with a time of 26.060 seconds. This outcome was a bit surprising as, until the day before the race, things were going quite well and suddenly changed. During testing on the day of the race, the car was repeatedly running off of the track either because it took a turn too fast or simply didn’t detect it. At a complete loss for what was suddenly causing so many problems, values were slightly changed, but hope was mostly lost.

During the final event however, after cleaning dust from the wheels one last time, things started working again. Amazingly, the little pirate ship zipped around corners and made it all look easy. During the final race, the maximum speed was set to 80% duty cycle, with a max speed on turns of 50% duty cycle. Through testing, these values were changed slowly to determine the optimal values for this specific system.

This was all done despite the sudden accident where track pieces crushed the car during testing, destroying the mount for the camera pole and snapping off a switch on the motor controller board. With help and some very quick thinking, another pole mount was quickly made, and the switch was soldered to be shorted in the on position. The broken pole mount can be seen below in *Fig. 10*, and the soldered switch can be seen in *Fig. 11*.

Fig. 10. The shattered camera pole mount

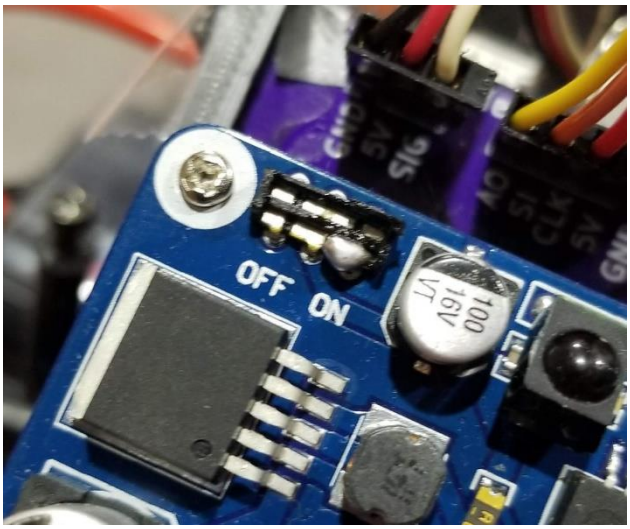


Fig. 11. The shorted switch on the motor controller board

5. CONCLUSION

Overall, the NXP Cup succeeded in challenging the members of team "Fast XOR Furious" and improving their engineering skills. Working so closely with classmates also added motivation for competition as well as allowing students to help each other in times of crisis. It provided a hands-on experience that helped develop skills in using sensors and interfacing with software. Not only did this project give a great educational example of a real world problem, it also reinforced many problem solving and general engineering ideas, helping in professional growth for all involved.

6. ACKNOWLEDGEMENTS

Team "Fast XOR Furious" wishes to acknowledge the contributions from these groups, in no particular order, without which the race would not have been possible:

A. Team "Sorry in Advance" for providing a 3D printed version of the new motor shield mount.
B. The RIT Baja team for machining a custom camera mount after a freak accident the night before the competition.
C. Dr. Raymond Ptucha for instructing this class and guiding teams in the right direction
D. Team "Phoenix" for help with soldering a fix for the motor shield that was required as a result of the previously mentioned accident.

7. REFERENCES

[1] Judge, John J, et al. *Learning Through Racing: Rochester Institute of Technology's Imagine RIT NXP Car Cup*. May 2017, suhailprasathong.com/docs/ieeenxpcup.pdf.
NXP Semiconductors, "Line Scan Camera Use," Doc-1030 datasheet, Jul. 2012 [Revised Apr. 2013].

[2] NXP Semiconductors, "Line Scan Camera Use," Doc-1030 datasheet, Jul. 2012 [Revised Apr. 2013].

[3] R. Ptucha. CMPE-460. Class Lecture, Topic: "Digital Filter Design." Kate Gleason College of Engineering office of Computer Engineering, Rochester Institute of Technology, Rochester, NY, Spring 2019.