CMPE-460 IDE NXP CAR CONTEST 2021

Rohan Patil and Shery George Mathews

Rochester Institute of Technology
Computer Engineering

ABSTRACT

Utilizing various hardware and interfacing techniques, an autonomous car was built, programmed, and raced to place 6th in the NXP Car Contest. A line scan camera mounted on the car took scans of a double lined track which was custom built on the day of the contest. Along with the line scan camera, other hardware utilized were the K64F microcontroller to program the car, a servo to steer the front wheels, two DC motors to drive the back wheels, and motor shield board. Interfacing with the camera data, motors, microcontroller peripherals, and servo was learned and tested during the initial part of the IDE course curriculum. Along with the interfacing of the data, PID control, thresholding, interrupt peripherals, and smoothing were also tested and implemented for the purpose of racing the car efficiently and effectively. The car finished 6th place with a time of 37.16s from its first run.

Index Terms— Autonomous systems, Autonomous automobiles, Micro-controllers, DC motors, Steering systems, Servomotors, H-Bridge, PID.

1. INTRODUCTION

Year after year, robotics competitions take place at universities and professional conventions. These competitions bring forth bright engineers with unique solutions trying to gain any competitive edge to win. The NXP Car Cup Competition embodies this, developing quality computer engineers through Rochester Institute of Technology coursework. Students learn to interface with camera data, motors, and embedded systems.

In total, 12 teams and 3 wild cards competed in this race for the shortest time around the track. The track was setup on the day of the race with roughly 8 minutes of prep and test time to adjust to the track. 8 teams made a successful run around the track, utilizing different optimizations and methods in order to get the fastest possible time. Other teams utilized dynamic thresholding, headlights, and high DC motor duty cycles.

The team set a cap of the left and right DC motors to a 80% duty cycle. This was the decision based on the motor sensitivity and the goal of minimizing slippage off of the track. This approach gave the camera on the NXP car more time to react to sharp turns and quick adjustments.

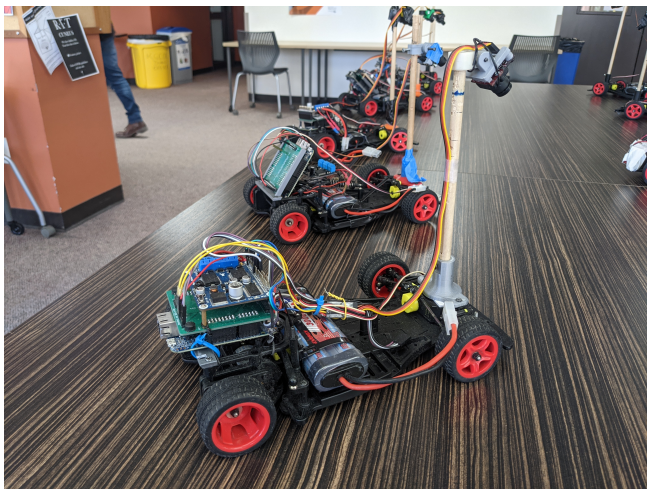

Fig. 1. NXP Car lined up prior to racing the track.

The general goal of the team was to win the race in the simplest and most consistent way possible. The idea was straight forward; high speed on straightaways, lower speed on curves with minimal oscillations.The line scan camera data was used to calibrate and adjust the position of the car on the race track. The bill of materials for the NXP Car is the following:

1. Freedom K64F microcontroller
2. MC33886 Raspberry Pi motor shield
3. K64 adapter shield header
4. Camera mount and wooden dowel
5. ROB0170 robotic kit car chassis
6. MG996R steering servo
7. CJMCU-1401 TSL1401CL linescan camera
8. KIT0166 DFRobot motor kit including 2 DC motors
9. Tenergy 7.2V Battery Pack for RC Car

These materials are put together to construct the car shown in Fig. 1, where the team's NXP Car was lined up moments before the competition.

The rest of this paper is organized as follows. After the introduction, Section 2 overviews the background and theories necessary to for different parts of the car. Section 3 introduces the proposed method. Section 4 presents the experimental results, and Section 5 contains concluding remarks. Section 6 considers acknowledgements, and Section 7 lists all references used.

2. BACKGROUND

Building the NXP Car required knowledge of key theories and ideas that explained the different working parts the entire project system. These theories and ideas were picked up along learning to interface the various car parts with the K64F development platform. The FRDM-K64F development platform was programmed in C on Keil uVision to interface with external peripherals and control all of the car logic. This development platform contains the K64FN1M0VLL12 MCU from ARM® Cortex®-M4 core embedded processor with maximum operating frequency 120 MHz, 1 MB flash memory, and 256 KB RAM [1]. The K64F houses the Flex Timer Module, integral to handle critical features of the NXP Car such as processing the camera data and generating PWMs to drive the DC motors and control the steering servo.

A team in a previous year's NXP Car Cup competition controlled their servo motor with quadratic servo tuning. This was a unique approach found from other teams who had computed the servo offset linearly. The team found that approach worked initially but when going at higher speeds this caused the car to turn hard when coming out of turns and sometimes made the car to turn too much to normalize itself [2].

The basic strategy on the current NXP Car for a fast race time included controlling the DC motors independently to initiate different speeds on individual tires, going slower while turning. To aid in this speed control, proportional integral derivative (PID) control was utilized.

2.1. Flex Timer Module

The Flex Timer Module (FTM) is a peripheral on the K64F microcontroller used to toggle the clock and serial input signals to begin a line scan on the camera as well as control the duty cycles of the DC motors and steering servo. The FTM can be used for input capture and output comparison as well as generating PWM signals. This timer is ideal for motor control and timing associated with external inputs and outputs [3]. FTM2 handled the camera driving logic along with the Periodic Interrupt Timer (PIT) controlling the integration period. When the PIT overflowed, it triggered an interrupt on the FTM, resulting in another line scan capture from the camera. The falling-edge "capture" read from the ADC module, fed in from the analog out on the camera. FTM3 controlled the duty cycle for the servo, and FTM0 controlled the duty cycle for the 2 individual DC motors. Both of these FTM controlling the servo and DC motors alike, navigated the steering, speed, and braking of the NXP Car.

2.2. PID Control

Proportional Integral Derivative (PID) Control uses error in a closed feedback loop to control and regulate the speed of the car. An error value is continuously computed and applies a correction, being the proportional, integral, or derivative. Integral control helps reduce steady state error. Proportional control attempts to get the minimal desired response with maximum allowed oscillation. Integral control helps reduce steady-state error. Derivative control helps bring the process into control faster. Fig. 2 displays the proportional, integral, and derivative components that utilize error in this feedback loop.

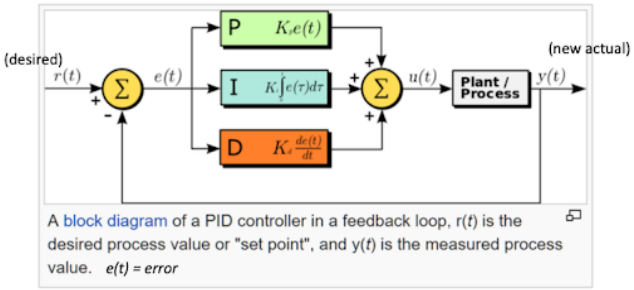

Fig. 2. PID control block diagram [4]

Equation 1 is an extension of Fig. 2. u(t) is the is the desired error adjustment summation of the P, I, and D terms.

$$u(t) = K_p e(t) + K_i \int e(t)dt + K_p \frac{de}{dt} \qquad (1)$$

2.3. DC Motors

In order to move the car forwards (or backwards), two individual DC motors were utilized to drive the back wheels of the NXP Car. The K64F standard 3.3/5V signals are a limiting factor to drive the DC motors at a higher voltage. An H-bridge is a circuit which allows one to both control the direction of a DC motor, and allows the usage of separate control and driving voltages [5]. Fig. 3 shows the H-bridge circuit used for each DC motor in order to control them independently of one another. The PWM signals generated by the micro-controller are amplified to actually drive the motors. The motor driver board attached to the motor shield has built in H-bridges.

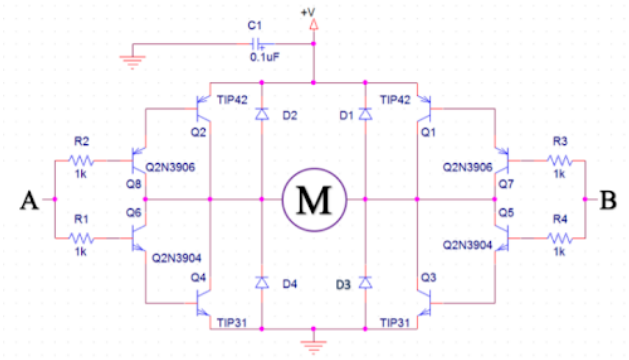

Fig. 3. DC motor control H-bridge circuit

3. PROPOSED METHOD

The NXP Car configuration is composed of various moving parts. In order to control the working speed and direction of the various motors, a motor shield allowed control from the same peripheral, the K64F microcontroller. Fig. 4 shows the wiring diagram between the the DC motors, camera, servo motor, and motor shield. The motor shield and adapter board sits on top of the K64F micro-controller and fully covers all headers. The pinout for the motor shield adapter is shown in Table I. Every pin on the header is connected to a corresponding pin on the K64. Additionally, Table II shows the camera, servo header pins and the the K64F pins.

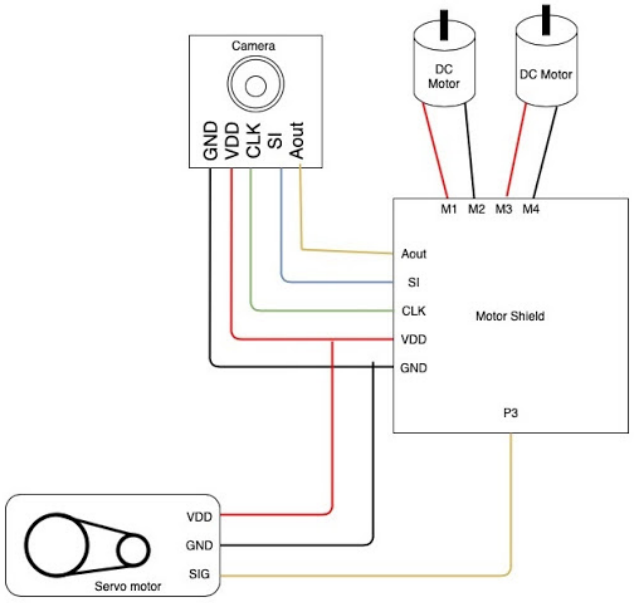

Fig. 4. Wiring diagram with servo motor, camera, DC motors, and the motor shield

Along with the motor shield, custom car parts were necessary to continue progress on the car. Provided to the team, custom 3-D printed camera mount and hinge were used to adjust the camera position. A custom 3-D printed board mount was to be used to contain the microcontroller and motor shield while the NXP car was driving. However, the team decided to make a simplified mount resting on the chassis frame of the NXP car using zip ties tied on the chassis and boards. This can be seen clearly in Fig. 1.

Once the various interfaced hardware was provided and setup, the main objectives to ensure a properly working NXP car were to process the line scan camera data, have an efficient centering algorithm, and implement variable speed and braking. The team was faced with challenges in regards to each of these objectives, where changes and sacrifices were made to uphold a working strategy of racing the car in a custom track on the day of the NXP Car Cup Competition.

Table I. K64 and Motor Shield Adapter Pin Associations [6]

Motor Shield Pin	K64 Pin
P0	PTB18
P1	PTE24(IR)
P2	PTC9
P3	PTD0
P4	PTE25
P5	PTB11
P6	PTB10
P7	PTB19
P21	PTC10
P22	PTC1(A1)
P23	PTC2(A2)
P24	ADC1_DP0
P25	PTB2(EN_A)
P26	PTB3(EN_B)
P27	RST
P28	PTC3(B1)
P29	PTC4(B2)
SCL	PTA2
SDA	PTA1
IDSC	DAC0_OUT
CE1	ADC1_DP1
CE0	PTB20
IDSD	PTC11
SCK	PTD3
MISO	PTD2
MOSHI	PTD1
Rx	PTC16
Tx	PTC17

Table II. Shield Header Camera and Servo Pin Associations [6]

Header Pin	K64 Pin
SIG	PTC8
CLK	PTB9
SI	PTB23
AO	ADC0_DP0

3.1. Processing Line Scan Camera Data

The raw camera data was difficult to discern between black and white due to the levels of noise that is noticeable on the top plot of Fig. 5.

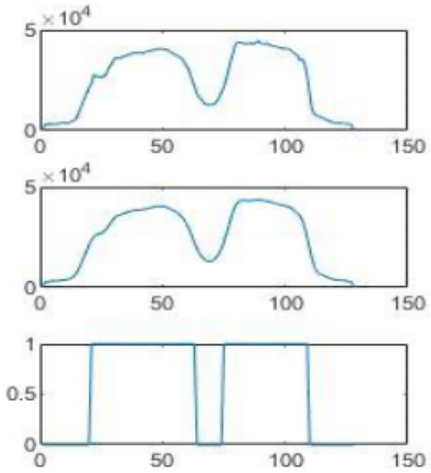

Fig. 5. Matlab line scan camera raw, smoothed, and square data plots

The plot shows how the camera data looks like when the line scan camera detects a black line on the center of the race track. The middle plot is the smoothed version of the top plot. This smoothing was done by taking 5 data points and taking an average. This is done in (2). The bottom plot in Fig. 5 was created by checking the data over or under a threshold for a square wave of absolute values. This helped eliminate noise created by small sudden changes on the camera. This threshold was adjusted to make the signal clean. The smoothed data was then scaled down by 900 before finding the left and right threshold values in the race track to bring down this values under 100 so that it could be typecast to an integer data type to get consistent values for left and right threshold under 100. i.e if two adjacent camera data was 65549 and 64800, there is a big difference of 749 between the values. Scaling down by 900 and typecasting to integer data type yields 72 for both values. Thus, the camera data will be better smooth and stable.

$$\begin{aligned} smooth[i-4] = ((line[i-4] + line[i-3] \\ +line[i-2] + line[i-1] + line[i] + line[i+1] \\ +line[i+2] + line[i+3] + line[i+3])/900) \end{aligned} \quad (2)$$

The raw camera data was stored in an array of length 128 named ”line”. The data points in this array were then smoothed by taking an average of five consecutive data points and storing it into another array of length 128 named ”smooth”. Data points 20 to 50 from the array ”smooth” was averaged to find the left threshold. Data points from 74 to 104 was averaged to find the right threshold. The thresholds were calculated to be 30 for left and right.

3.2. Centering Algorithm

The goal of the team was to win the race in the easiest way possible. The idea of track following was straight forward. High speed with none to little oscillations on straight ways and lower speed along with differential steering on curves. The line scan camera data was used to calibrate and adjust the position of the car on the race track. The initial method for following the white track was with hard coded left and right threshold values. If the camera was within 2 inches of the ”center” of the track, the servo would adjust to the straight position, and the DC motors would spin at an 80% duty cycle. This made the car go straight at a high speed in a straightway. When the car approaches a turn, the servo adjusts the front wheels to left or right based on the right and left threshold values. If either of the values is lesser than the provided threshold values, the servo turns all the way to the left if the left value is greater than the right, and vice-versa. This logic worked out the way it was expected. However, sharp turns made the car drift too much from the center, and the car started oscillating too much before getting back into the center region. This was because the differential steering was not included, and the servo turned all the way to the right or left. The PID control algorithm did not solve the oscillation issues. The team tried many different combinations of proportional, integral and differential constants. PID controller worked better with the proportional constant alone. However, the team was not able to get rid of the oscillation.

This issue was fixed with a new method by taking the absolute value of the difference between the right and left value and saving it as the front wheel error and then adding and subtraction to the servo duty cycle to adjust the front wheel accordingly depending on how far the car is off from the center. If the car is close to the left side of the track the servo duty cycle will be the front wheel error added to 6.2. Similarly, if the car is close to the right side of the track the servo duty cycle will be the front wheel error subtracted from 6.2. The car front wheel is straight when servo duty cycle is 6.2. This helped to get rid of the oscillation as the servo adjusted

its position based on the error instead of turning all the way to the right or left. Scaling down and averaging the camera data to find the left and right thresholds resulted in an integer number and thus the absolute value of difference between the left and right value (the front wheel error) resulted in zero when the car was roughly in the center region of the track. This helped to reduce the oscillation significantly and hence no other logic was required to get the car running within the race track smoothly. The logic used to make the car follow the white track is shown in the flowchart in Fig. 6.

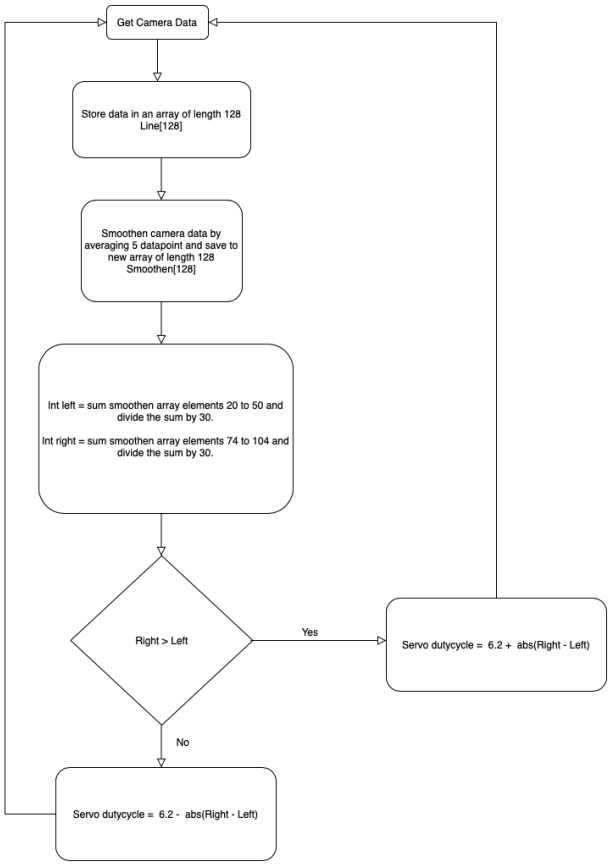

Fig. 6. Centering code logic block diagram

It was found that the absolute value of the difference between the right and left values (front wheel error) was 30 at both right and left extremes. The team utilized this pattern and came up with a formula to calculate the servo duty cycle error at different positions on the track. (3) gives the servo duty cycle error.

$$error = k_p|right - left| \tag{3}$$

The k_p value used to adjust the servo position was $1/30$. This k_p value was chosen because at 5.2% duty cycle the servo turns all the way to left, and at 7.2% duty cycle, the servo turns all the way to the right. The center position of the servo was 6.2. Therefore, at left and right extremes, the maximum value to be added to or subtracted from the 6.2% duty cycle is 1%. i.e. if the car is closer to right side of the track the servo duty will be adjusted based on (4),

$$servopwm = 6.2 - error \tag{4}$$

When the car is closer to left side of the track, the servo duty will be adjusted based on (5),

$$servopwm = 6.2 + error \tag{5}$$

When the car is roughly at the center the abs(right - left) will be zero. Thus the servo duty cycle will be 6.2 and hence the front wheels will be straight. The PuTTY terminal display of the left and right values calculated from the camera data when the car is on the left side, center and right side of the track is shown below in Fig 7.

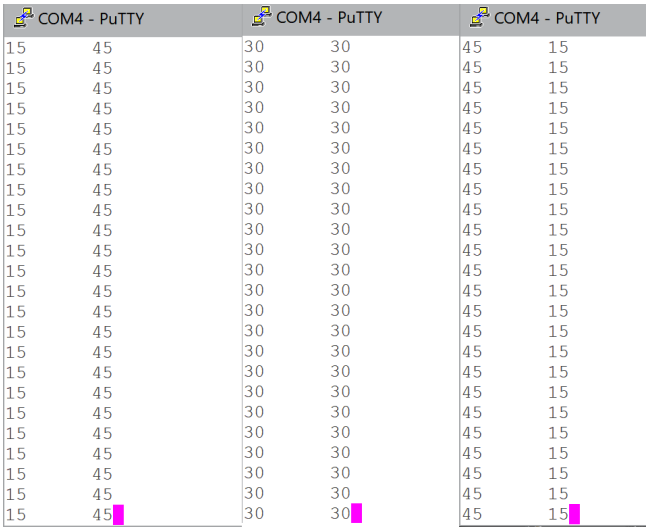

Fig. 7. Left and Right values from the camera data when the car is on the left side, center and right side of the track

At a right turn, the DC motor on the right side of the car spins at 40% duty cycle and the left motor spins at 70% duty cycle making a sharp right turn and vice-versa.

3.3. Variable Speed and Braking

Under certain conditions, the team would change the duty cycle of the DC motors in an effort to change the speed. This includes both slowing down and braking.

Turning at high speeds would cause the NXP Car to drift outside of the track, so implementing logic to slow down during turns would be the first effort in managing the speed of the car. Before implementing PID, the duty cycle of the DC motors would increase to speed up when the servo was in the dedicated center position (straight). Using PID, we could adjust the left and right DC motor duty cycles respectively to

slow the corresponding wheel down to come closer to the inside of the turn.

The last approach that was taken to manipulate the speed was a constant braking logic. The NXP Car would still leave the track on a sharp turn after a long straightaway, so measures were taken to brake more frequently. A braking system was added to the car code. The car braked every three seconds when it was on a straight way. While making a sharp turn, the speed was significantly reduced so that the car would be able to stay on the track more consistently.

4. RESULTS

During the actual race, the car speed was significantly reduced to avoid the risk of being disqualified and the car was running at a duty cycle of 40. The slow speed along with the braking system resulted in longer time to finish the race. However this safety measure helped to finish the race on the first attempt out of the allowed three. The car finished the race in 37.16 seconds and the team ended up in 6th position.

4.1. PID Tuning and Other Adjustments

The motor shield had a design flaw which is the absence of a pull up resistor to be connected between the base and emitter of the transistor in the motor shield board. This causes the DC motors on the car turning on randomly during micro-controller reboots. During a micro-controller reboot while the car was on the racetrack resulted in the motor turning on at 100% duty cycle due to the absence of the above mentioned pull up resistor and the car flew straight to a portable white-board in the IOT room at RIT destroying the micro controller mount and the wooden camera holding dowel. As a result the team had to zip tie the micro-controller board along with the motor shield to the car frame itself which came into the advantage of the team as the zip tied boards are stronger compared to the previous 3D printed micro-controller holder. Cutting down the wooden camera dowel height after the crashing also helped the team better as the camera only sees necessary track data at lower height as compared to unnecessary carpet data collections at bigger heights.

Carpet detection was turned off for the actual race as there are chances the camera sees the carpet on a sharp curve at low light zones which could stop the car on the track and result in a disqualification. The authors decided it was better to let the car run a low speed rather than stopping as there are higher chances for the car to get back into the track before completely leaving the track or being stopped.

4.2. Final Results of the Race

During practice sessions the authors noticed many other teams with faster cars failing at low light zones in the IOT room. The authors decided to reduce the duty cycle of the DC motor in the car to reduce the speed. This resulted in the car going at a lower speed as compared to what it was before. The new duty cycle was 40% whereas the previous setting was 70%. Slowing down the car was risky as many other teams had faster cars, reducing the authors chance to finish the race in the top seven. However, this speed adjustment helped the team to finish the track in the first attempt itself compared to many other teams failed at the low light zones and disqualified. The author's car finished the race in 37.16 seconds and the team ended up in 6th position. Table III displays the final results of all the student teams.

Table III. NXP Race Spring 2021 Final Race Times

Teams	1st Run (s)	2nd Run (s)	3rd Run (s)	Final (s)
Michalski and Skafal	x	x	x	x
Myers and Buchheister	32.97s	-	-	32.97
Patil and Mathews	**37.16s**	-	-	**37.16**
Wells and Dombrowski	x	x	x	x
Yakobchuk and Yakobchuk	x	x	38.21	38.21
Au and Larson	x	x	27.33	27.33
Burnette and Carile	x	x	27.33	27.33
Nicolais and Nichols	x	x	x	x
Blust and Simmons	x	24.68	-	24.68
Meyerson and Poliwoda	21.93	-	-	21.93
Crane	x	x	25.45	25.45

4.3. Future Improvements

The braking mechanism is one area that would warrant future improvements for significantly faster speed. Currently, the NXP Car slows down during turns, and brakes with a 1.7s delay every few seconds to avoid reaching too high of a speed. Rather than constantly brake with a timed delay, a hardware timer can be utilized to issue braking on straightaways after a certain length of time. This way, the only time the car slows down is at a speed where it has no chance of slowing down fast enough for a turn and for sharp turns. This improvement would increase the time the DC motors can run at a higher duty cycle for a faster speed over a longer period of time.

5. CONCLUSION

Completion of the NXP Car Project and Contest has provided a hands on engineering experience in building, testing, debugging, and fixing an autonomous car. Learning to interface sensors and digital electronics with embedded systems is relevant to a wide variety of engineering applications. The success of this project provided these skills and highlighted critical engineering principles necessary in the completion of any project. The ability to come up with practical solutions in adversity of new problems with limiting constraints was exercised through the course of this project.

6. ACKNOWLEDGMENT

The authors express their sincere thanks to CMPE-480 IDE course instructor Louis Beato, teaching assistants Xavier Brooks, Eri Montano, Connor Henley, and Brunon Sztuba for all the help and suggestions that helped to improve the design methodology and implementation which helped this project to reach this outcome.

7. REFERENCES

[1] *FRDM-K64F Freedom Module User's Guide*, rev 0.1, NXP Freescale Semiconductor Inc., 2014, pp. 1-1.

[2] Syed Tousif Ahmed, Jeff Barker, and Christopher Ranc, "Savage Valhalla: Building a Fast Autonomous Car," in IDE NXP Car Competition, pp. 4-5.

[3] *Interface Digital Electronics Laboratory Manual*, Rochester Institute of Technology Department of Computer Engineering, Rochester, NY, Spring 2021.

[4] L. Beato. (2021). Control Systems [PowerPoint slides].

[5] L. Beato. (2021). Locomotion [PowerPoint slides].

[6] Michael Baumgarten and Matthew Toro, *K64 Motor Shield Adapter*, Rochester Institute of Technology Department of Computer Engineering, Rochester, NY, Spring 2021.