Savage Valhalla: Building a Fast Autonomous Car

Syed Tousif Ahmed, Jeff Barker, and Christopher Ranc

***Abstract*—The purpose of this project was to use a kit of hobbyist scale car components and a micro-controller to program an autonomous car to race around a track for the IDE Car Cup competition. During the competition, the car was required to follow a double line track and complete the course in the fastest possible time. To accomplish this, the car used two individually controllable DC motors, a Futaba servo for steering, a one-dimensional infrared line scan camera for line detection, and two controller boards. The controller boards included the Freescale K64F micro-controller, which controlled the steering and driving logic, as well as processing of data from the line scan camera, and a motor controller, originally designed to pair with the Freescale KL25Z micro-controller, which controlled power delivery to the DC motors and steering servo. Using interrupt driven processing and Proportional-Integral-Derivative (PID) car control, the car was successfully able to complete the competition track in 17.683 seconds, securing 2nd place in the competition. The best time that was recorded after the competition on the same track was 17.302 seconds, beating the winning lap time of 17.329 seconds.**

I. Introduction

Prior the buyout of Freescale by NXP, and the subsequent buyout of NXP by Qualcomm, the car cup competition was an officially sponsored event designed to engage students from different schools in friendly competition through programming of an autonomous car for racing around a track. Competing teams would use a collection of car parts and a Freescale microcontroller to form the car. In prior iterations of the competition, a Freescale KL25Z microcontroller was used as it paired with the bundled motor controller board such that the boards could be directly attached together. Most current teams utilize the Freescale K64F microcontroller for its improved performance and hardware capabilities. The primary programming interface used was the Keil uVision IDE.

After building up C code for basic control of the car components, including the DC motors, steering servo, and line scan camera, control theory and line detection algorithms were implemented to enable the car to follow two outside lines of the track, where prior tracks utilized single line following. Ideas from control theory included PID control for speed of the motors and steering, differential motor speed for improved turning, non-linear steering for improved control at high speeds, and braking.

II. Hardware

The cars used for the competition were assembled from a kit of hobbyist car parts, along with a micro-controller, motor controller board, and 3D-printed mounting components, as seen in Figure 1.

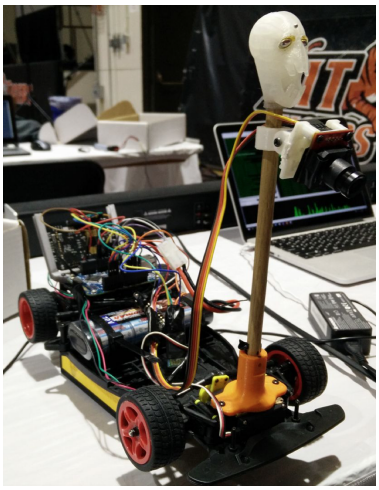


Figure 1 : Assembled Car during Race Day

Specifically, these components include:

- Freescale K64F microcontroller
- Freescale K25Z compatible motor controller
- Futaba steering servo
- 2 x DC motors
- Texas Advanced Optical Solutions infrared line scan camera
- Wide angle lens
- Bluetooth module
- Competition car chassis with suspension, steering linkage components, and wheels
- 7.2V battery
- 3D printed components
    - Motor controller mount
    - Camera pole mount
    - Camera holding mount

The suspension, steering linkage components, and wheels were attached to the main chassis for form the body of the car. The Futaba steering servo was attached to the front portion of the chassis and to the steering linkage to control the direction of travel of the car. The DC motors were mounted at the rear of the chassis to drive the car forward and backward. The camera pole mount was placed above the steering servo, into which a wooden dowel rod with the line scan camera attached to the camera holding mount were inserted. The DC motors and steering servo were connected to the motor shield, which was mounted with a 3D printed mount to the rear of the car chassis above the DC motors. The line scan camera, as well as certain connections from the motor controller, were connected to the K64F microcontroller, which was attached to another portion of the motor controller mount.

S.Ahmed is with the Computer Engineering Department, Rochester Institute of Technology, Rochester, NY 14623, USA. e-mail: sxa1056@rit.edu

J.Barker is with the Computer Engineering Department, Rochester Institute of Technology, Rochester, NY 14623, USA. e-mail: jdb2438@rit.edu

C. Ranc is with the Computer Engineering Department, Rochester Institute of Technology, Rochester, NY 14623, USA. e-mail:cxr3473@rit.edu

A. K64F Microcontroller

The primary mechanism for implementing the autonomy of the car was the Freescale K64F microcontroller. The K64F utilized an ARM Cortex M4 processor running at a max clock speed of 120MHz, with 1MB of flash storage and 256KB of RAM [2]. Hardware modules on the microcontroller included two user push buttons, one RGB LED, as well as chip specific hardware, including multiple FTM timer modules, each of which were capable of generating Post Width Modulation (PWM) signals, PDB and PIT timer modules, a 16-bit analog-to-digital converter for sensor data conversion, and GPIO pin control. The K64F was capable of taking advantage of additional hardware such as ethernet, digital-to-analog conversion, and micro-SD card storage expansion, but these features weren't utilized for the competition.

The main components with logic controlled by the K64F were the Futaba steering servo, the two DC motors, and the line scan camera. To control the steering servo, a single channel on the FTM3 timer module was used to generate a PWM signal to control the angle the servo needed to turn to, which would dictate the direction of travel of the car. To control the two DC motors, four channels of the FTM0 timer module were used for PWM signal generation. Four PWM signals were needed in order to allow for fully independent control of each motor, since each motor used two separate leads, and applying a PWM signal to a particular lead on the motor would cause the motor to spin in a particular direction.

To control the line scan camera, multiple hardware modules were used. These included the FTM2 timer to generate the clock signal and SI trigger to cause the camera to begin a line capture, the PIT timer to trigger the FTM2 timer, the ADC module to convert the output data from the line scan camera, and two GPIO pins for the clock and SI trigger connections to the line scan camera.

B. K25Z-compatible Motor Controller FRDM-TFC

The FRDM-TFC shield [3] is a utility board which was designed to be used with the KL25Z microcontroller. It can drive the up to two DC motors, the servo and has several interfaces such as two push buttons, 4 DIP switches etc. This project only utilized the DC motors and servo power delivery utility of the TFC shield. The TFC shield was mounted on a 3D printed fixture to avoid displacing the board when the car's moving.

C. Steering Servo

The primary steering mechanism used was a Futaba servo. The servo used three inputs: PWM (to determine the amount of turning), VCC (5.0V-6.0V), and GND. In order to control the amount of turning of the servo, the PWM input was connected directly to the output port corresponding to FTM3 timer channel 4 on the K64F microcontroller. The maximum PWM frequency that could be used to control the servo was 50Hz, although this frequency of servo updating did not impact the granularity with which the servo could be controlled.

D. DC Motors

Driving of the car was controlled by two brushed DC motors. Each motor had two leads to which a PWM signal would be in order to determine the direction of rotation of the motor. Since the motors required a larger current to be driven than what the PWM signals from the K64F microcontroller could supply, the PWM driving signal connections were connected to the inputs to two full H-bridge circuits on the motor controller.

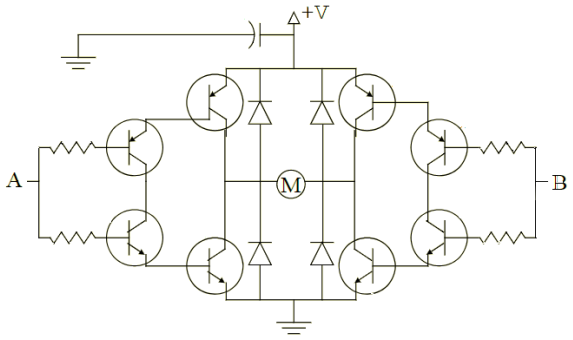


Figure 2 : H-Bridge Motor Control Circuit

These H-bridge circuits, similar to the circuit in Figure 2, would amplify the PWM signals from the microcontroller and drive each motor individually, since two full H-bridges were used, in independent directions and at independent speeds, depending on which lead the PWM signal was applied to, and depending on the frequency of the PWM signal, respectively.

E. Line Scan Camera

The line scan camera [4] has a CMOS sensor array of 128 pixels. A wide angle lens was attached to the camera to get a greater field of view of the track ahead. The camera takes in two input signals, Clock (CK) and Serial Input (SI) and produces an Analog Output (AO) signal.

The input signals to CK and SI were given from the GPIO pins in the K64. SI determines the start of capture and CK controls the pulse to obtain 128x1 array of data. The AO output is a voltage value between 0 and V_{dd} and hence, was conditioned with a Analog to Digital Converter in the microcontroller.

Several angles and heights for placing the camera on the wooden dowel were tested. The optimal angle and position was found through trial and error and it was fixed throughout the project by making a marking of the positions on the fixtures.

F. Bluetooth Module

As an added debugging tool, a bluetooth module was used to send the camera data wirelessly to a laptop or cellular device, where the data could then be sent to a plotting program. Using the bluetooth module allowed for quick debugging of what the line scan camera was detecting for the track and the lines of the track, all without being physically tethered to the car, which allowed for debugging while the car was running.

In order to implement the bluetooth module, the same UART initialization code which was used to implement UART data transfer over USB was modified to change the transmit and receive inputs of the UART module on the processor to use two of the physical pin outputs on the K64F instead of the USB connector on the board. These ports were connected to the transmit and receive inputs of the module, as well as 3.3V and GND from available ports on the K64F being connected to VCC and GND on the module.

After powering and initializing the module, the device would be displayed as 'HC-06' when viewing the bluetooth connections on a device such as a laptop. Connecting to this device would open up upstream and downstream connections to the module which would be displayed as COM ports on the host computer to which a terminal program, such as PuTTY, could be connected to send and receive data to and from the module.

G. Fixtures

Different 3D printed parts were used in the car, to support several of the working components, such as supporting the line scan camera and its angle, and mounting the TFC shield. The 3D models were supplied as parts of the kit.

During testing and debugging, some of the parts broke as a result of crashes and mishandling, and thus, they were reprinted using the CURA software and a LulzBot Mini 3D printer. Moreover, the fixture for the line scan camera was marked with a pen, when optimizing for the line scan camera angle. A 3D printed part inspired from the movie "Mad Max: Fury Road" was mounted on top of the line scan camera for beautification purposes.

III. Proposed Software Solution

The algorithms that determined midpoint, speed, and direction for the car were performed using functions managed by the FTM hardware interrupt that drove the car's servo. The frequency of it was the lowest of the interrupts being used so it permitted the functions to update optimally without impeding the speed of the code.

A. Line Processing

The signal from the camera was fed to an averaging function that smoothed the camera data to help better identify the track. The averaging implemented was a five point weighted average. Weights of .1, .2, and .4 for the averaging were chosen from a student groups paper in the Freescale version of the Cup that had used a similar camera [1]. The averaging equation for creating the new array can be seen below in Equation 1.

$$avg[i] = .1l[i-2]+.2l[i-1]+.4l[i]+.2l[i+1]+.1l[i+2] \quad (1)$$

When smoothing the line data the function also would take a running average of the smoothed line values. The purpose of the running average was to be used as a dynamic threshold for the next part of the algorithm that was binary conversion. The function that did this would utilize the dynamic threshold to create a new array composed of zeros and ones. Ones were placed where the average array values were greater than or equal to the dynamic threshold, while all other values were set to zero. This should remove any considerable noise and turns the line data into a square pulse and makes it easier to determine the edges seen by the camera. The dynamic aspect of the threshold was to permit easy transitions to tracks in different lighting, as the line values would change depending on the lighting.

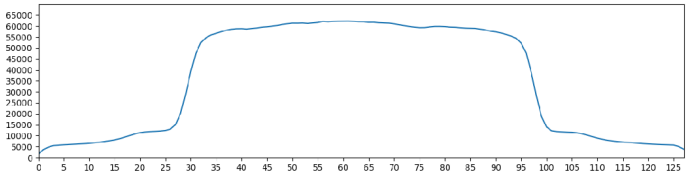


Figure 3 : Python Plot of Smoothed Line Feed

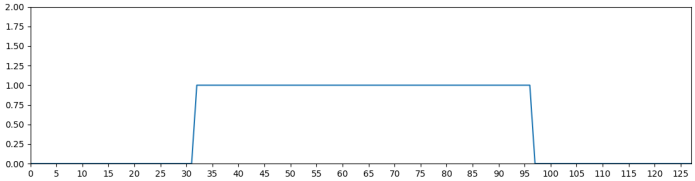


Figure 4 : Python Plot of Binary Conversion

Figures 3 and 4 both represent the white portion of the track that the car camera sees. The First figure being the feed after being smoothed and the second being the binary conversion. The rectangular shape of the binary conversion made it easy to determine left and right edges of the white that the car could see. This was by finding the indices of ones with zeros to the left or right of them, where a 01 would be a left edge and a 10 would be a right edge. The midpoint between these indices indicated as to where the car was on the track. Before the midpoint could be officially determined error checking was implemented to see if any noise was large enough to make it through and become a pulse.

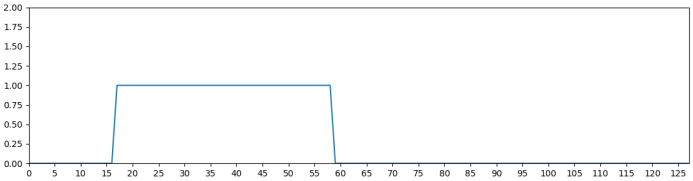


Figure 5 : Python Plot Approaching Left Side of Track

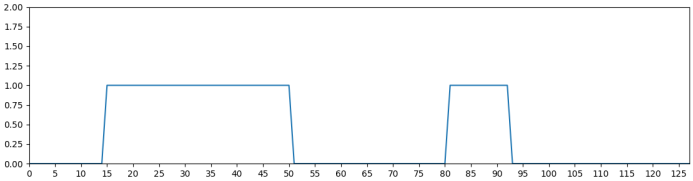


Figure 6 : Python Plot Left Side with Fixable Noise

Figure 5 shows what the processed camera data looks like when approaching the left side of the track. The pulse became more compact while approaching the left side of the

array data. In this case the error checking would be skipped and the indices of the edges would be taken to determine the midpoint. For the case of Figure 6 which indicates the camera seeing and additional pulse of white the error correction was implemented. This was done by comparing the width and indices of each pulse to the width and indices of a previously successful pulse. The pulse with the smaller percent difference from the previous recording was picked. If however both width and indices percent differences aren't discernible neither were picked and the previous indices were used for finding the midpoint.

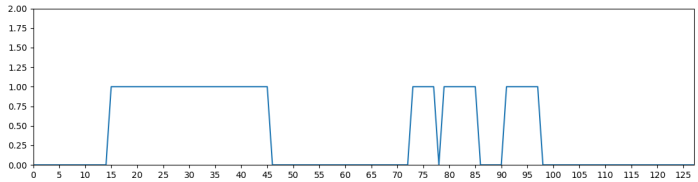


Figure 7 : Python Plot Left Side with Non-Fixable Noise

The case of Figure 7 showed the camera seeing more than 2 pulses which was more difficult to discern which pulse would be closer to the previous successful indices recorded. The causing of this case was probably due to noise getting through the filtering or a glare maybe coming off the side of the track. So these pulses would be ignored and the previous indices were then used for finding of the midpoint of that run.

B. Quadratic Servo Turning

The first thing determined with the found midpoint by the algorithm was the servo PWM to appropriately turn the cars front wheels. When first implemented the difference from the desired midpoint index of 65 was used to proportionally offset the centering servo PWM of 7.75 to turn left or right. This 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. Usually making it bounce between the left and rights sides of the track. This was ultimately due to the linear relationship of how the servo offset was calculated and the solution chosen to remedy this was to make that relationship more of a quadratic. This helped the car by making it turn its wheels at smaller amounts when the midpoint difference isn't too large but made it turn much harder when coming close to the edges. Spreadsheet plotting made it easier to determine a quadratic function for the offset PWM for the servo. A linear range from 0 to the absolute offset of 1.75 was plotted against a pseudo quadratic range with the same min and max values.

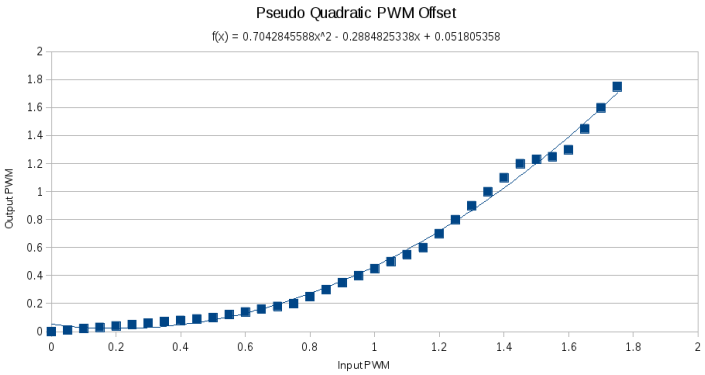


Figure 8 : Pseudo Quadratic Plot

$$f(x) = 0.7042845588x^2 - 0.2884825338x + 0.05180535 \quad (2)$$

The trend-line of the psuedo quadratic can be seen above in Figure 8 and to be of a more desirable form. The resultant 2nd degree quadratic can be seen in equation 2. The values of the function were adjusted through trial and error to from a more desirable equation.

$$f(x) = 0.6522225753x^2 - 0.1072593887x \quad (3)$$

The c portion of the quadratic was removed so it could result in 0 when given 0. This however made the lower end results become negative. To mitigate this the a and b portions of the quadratic were lowered resulting in equation 3.

Table 1 : Quadratic Table Values

PWM Offset	Pseudo Quadratic PWM	Quadratic PWM
0	0	0
0.05	0.01	-0.003732413
0.1	0.02	-0.0042037131
0.15	0.03	-0.0014139004
0.2	0.04	0.0046370253
0.25	0.05	0.0139490638
0.3	0.06	0.0265222152
0.35	0.07	0.0423564794
0.4	0.08	0.0614518566
0.45	0.09	0.0838083466
0.5	0.1	0.1094259495
0.55	0.12	0.1383046652
0.6	0.14	0.1704444939
0.65	0.16	0.2058454354
0.7	0.18	0.2445074898
0.75	0.2	0.2864306571
0.8	0.25	0.3316149372
0.85	0.3	0.3800603303
0.9	0.35	0.4317668362
0.95	0.4	0.4867344549
1	0.45	0.5449631866
1.05	0.5	0.6064530311
1.1	0.55	0.6712039885
1.15	0.6	0.7392160588
1.2	0.7	0.810489242
1.25	0.8	0.885023538
1.3	0.9	0.9628189469
1.35	1	1.0438754687
1.4	1.1	1.1281931034
1.45	1.2	1.215771851
1.5	1.23	1.3066117114
1.55	1.25	1.4007126847
1.6	1.3	1.4980747708
1.65	1.45	1.5986979699
1.7	1.6	1.7025822818
1.75	1.75	1.8097277066

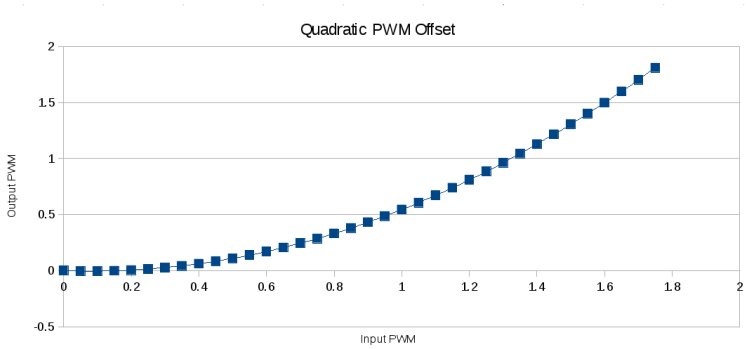


Figure 9 : Quadratic Plot

Table 1 and Figure 9 display what equation 3 was able to process when fed the linear PWM range determined by the midpoint difference. To handle the small range of negative values the quadratic could produce the function using the quadratic was written to set any value less than 0 to a PWM offset of .001 to keep turning small.

C. Car State Transition

Aside from determining the turning of the cars wheels the midpoint difference was used in part to determine state of the cars movement. The car would change between various states in order to determine what PWMs to use and how they were applied to the car motors for optimal speed for going straight or turning. These states were defined as STRAIGHT, TURN, and HARDTURN. STRAIGHT simply meant that the PWM for the motors would be set to 100 when it determined that the midpoint difference was less than 50% of the midpoint max offset value of 15. The max offset was found when placing the car to the far left and right of the track. Either the TURN or HARDTURN would engage when the midpoint difference was equal to or higher than the 50% value.

The TURN State simply engaged differential motor speeds with the use of a Proportional-Integral-Derivative (PID) control to gradually change them and also permitted turning at high speeds. The HARDTURN state similarly engaged differential breaking on the motors by applying reverse PWMs on the motors with the use of PID. This was done in the case of the car going straight at full PWM for a long period of time and then entering a turn. This was determined by using dynamic memory allocation via a linked list to keep track of the last 12 states the car was in. The first 2 states acted as a buffer while the 10 tail states were checked to see if 80% were straight or not. The 80% check was in case the car entered the turning state briefly in a straight away. When entered a counting variable incremented by the car's PIT timer would be set to 0 and the HARDTURN PWM settings would be engaged. The HARDTURN state would remain until the PIT timer incremented to the count limit of 25 and would then transition to the TURN state. If the midpoint difference was less than 50% of the midpoint max offset before the count finished then it transitioned back to the STRAIGHT state.

D. PID Use for DC Motor PWM

PID control was used to drive the PWM on the DC motor to a maximum PWM. The maximum PWM was determined by the car state algorithm. Hence, changing the PID to the DC motors according to the current state of the car, made it accelerate faster without derailing from the track. Moreover, the PID was tuned incrementally, i.e. first the P control was tuned by keeping I and D zero and set to the maximum value for which the car was stable, next the I and D were tuned in a similar fashion.

IV. Results

A. Improved Speed

In the first iteration, the car did not have any notion of states and was simply using PID control to reach a static PWM for the DC motor. Although it was stable on the track, the speed was limited. By incorporating states, PID became a function of the track and hence, higher PWM values were introduced for STRAIGHT states. As a result, the car could accelerate faster and efficiently make turns at high speeds.

Moreover, in the first iteration, the servo didn't use any quadratic tuning and linearly changed to go left or right. This caused the car to constantly sway in the curvy tracks. As a result, during high speeds, the car was prone to derailing. Hence, by introducing a quadratic function to tune the servo, the car didn't move as much when it saw the curvy tracks, and hence was able to complete it with higher speeds without derailing.

B. Improved Line Processing

The line processing algorithm was tuned iteratively by testing runs on several types of tracks. For instance, when the car reached a 4 way intersection, the signal from the camera was similar to Figure 7. Hence, through trial and error and encountering such edge cases, the algorithm accounted for such situations by ignoring such spikes in the signal.

C. Performance Tweaks

To get the best possible run, several tweaks were discovered. Some rules of thumb used throughout every runs were:

- If the camera was too inclined towards the track, the car became unstable. Hence, the camera angle was set very slightly and almost horizontal as seen in Figure 1.
- Dust on the tires is a major factor in causing the car to derail. Hence, before every run, the tires were cleaned.
- For every runs and tests, a fully charged battery was used whenever possible. This ensured that any of the tunings such as PID, line processing etc. didn't take the battery charge into account.

V. Conclusion

In this work, a fully functioning autonomous car was built from the ground up. The car was made using commercially available electronics components. Concepts from digital electronics, control theory and embedded programming were applied. Through a competitive racing, the car was able to achieve a lap time which secured second place in the IDE Cup Car Competition. Although the current solution

demonstrated speed and robustness, several improvements can be made to the car. For instance, the car state transition can be made more robust to avoid false positives. Moreover, machine learning techniques such as creating a simulated track and using reinforcement learning for control can be introduced.

ACKNOWLEDGMENT

The authors would like to thank the course's teaching assistants and the instructors - William Gowell, Andrew Lints, Long Pham, Dr Ray Ptucha and Prof. Louis Beato - for their help with the project.

REFERENCES

[1] Lydia Hays and William Gowell. *Line Following Vehicle for the Freescale Competition*, Rochester, NY: Rochester Institute of Technology.

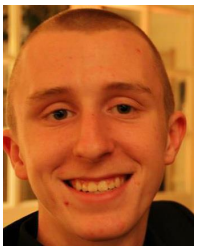
[2] *FRDM-K64F—Freedom Development Platform—Kinetis Mcus—NXP*. Nxp.com. N.p., 2017. Web. 15 May 2017.

[3] *Freescale Cup Shield For The Freedom KL25Z — NXP Community*. Community.nxp.com. N.p., 2017. Web. 15 May 2017.

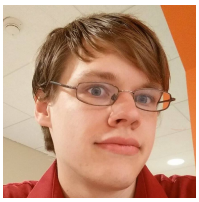
[4] *Line Scan Camera Use — NXP Community*. Community.nxp.com. N.p., 2017. Web. 15 May 2017.

[5] *How To Build A Robot Tutorials - Society Of Robots*. Society-ofrobots.com. N.p., 2017. Web. 15 May 2017.

Syed Tousif Ahmed is an undergraduate student in Computer Engineering at RIT. He is working as a Research Assistant in the Future Everyday Technology Lab and at NTID. His interests lie in Computer Vision, Machine Learning, Embedded Systems and Cryptography.



Jeff Barker is a fourth year Computer Engineering student at the Rochester Institute of Technology. He is currently pursuing the Advanced Course in Engineering (ACE) cybersecurity internship program at the Air Force Research Laboratory. His interests lie in offensive and defensive cyber strategies, embedded and systems level programming.



Christopher Ranc is an undergraduate student in the Computer Engineering BS program at RIT and Minoring in Physics. His interest stem into personal projects, Linux Containers, Functional, Embedded, and systems level programming.