

CMPE.460.01 – Final Project

Freescale Car

Name: Donald MacIntyre, Andrew Cope, Jacob Lauzon, Ryan P McLaughlin

Performed: 12/5/2014

Submitted: 12/10/2014

Lab Section: L1

Teaching Assistant: Vu Dinh

Teaching Assistant: Lisa Trova

Lecture Section: 01

Professor: Adriana Becker-Gomez

Abstract

The purpose of this project was to design and implement an autonomous, line-following, battery powered car. The car was designed to meet the specifications of the Freescale Cup competition. These specifications provided requirements for the microcontroller, batteries, motor driver, car size, and track size that would be used. The car was built from a provided Freescale Cup car kit, which included everything necessary to construct and run the car. The Freescale K25 microcontroller was used to control all aspects of the car, and was programmed using Matlab and Simulink. A Matlab package was provided specifically for the Freescale Cup that added Simulink libraries for each part of the car (the steering servo, the motors, etc.). To allow for line following, a 128 by 1 pixel camera was mounted to the front, center of the car. Simulink was used to program the car to receive the data from the camera and produce an output to send to the steering servo. To produce the output, a Proportional-Integral (PI) controller was used. The PI controller provided proportional gain to respond to turns and integral gain to reduce the steady state error to zero when the car was going straight. To help with turning, a differential wheel drive was implemented in Simulink that would allow for one wheel to slow down, or even stop, in the event of a sharp turn. This allowed for the car to slow down while turning, ensuring that it did not lose the line. An iterative process was used to tune the parameters of the PI controller and the differential drive. Many iterations of testing and tuning were performed to reach the final result of the car completing the track unassisted at a moderate speed. The car was able to complete a track with various size turns, two straight sections, and an intersection. In the future, many enhancements can be made to the car to allow for better steering control, better speed control, and faster overall speed. Upgrades to consider include a faster processor, another camera, wheel encoders, and a better controller model.

Procedure

The provided regulation Freescale Cup car components were first assembled to create two cars. Each kit included a car body (including back wheel drive motors, 4 wheels, and a suspension), a stepper motor, a Freescale FRDM-KL25Z microcontroller, a FRDM-TFC motor controller shield, a bumper, steering control components, 2 batteries, a battery charger, and miscellaneous hardware (including screws). All assembly and programming steps were executed on both cars/kits, so all future references will refer to a single car to reduce redundancy.

Referring to the Freescale Car Assembly Manual, the steering control unit of the Freescale car was assembled. To do this, an Arduino was used to turn the steering servo to its' 90° position, in order to center the servo such that it would be able to turn 90° in either the positive or negative direction before reaching the servo's stop. Centering the servo guaranteed that once the tie rods (connecting the servo to the wheels) were installed, the servo's previously unknown position would not prevent the wheel's from turning in a full range of motion, (the servo could have been positioned starting at its 0° position, at which point it would be unable to move in the negative direction, preventing the car from turning in one direction). After centering the servo, the servo

horn was assembled and attached in the center position of the servo, as shown in the Freescale Car Assembly Manual, in order to provide a point to attach the left and right wheels to the steering servo via a tie rod. Specific care was taken in assembling the servo horn, as the servo horn assembly came with a number of components, specific to the manufacturer of the servo. The servo was determined to be a Futaba model, so the servo horn components marked with “FU” on them were selected and assembled to create the horn. The cover over the servo gear was then removed, and the servo horn was pushed down tightly onto the gear and screwed on to secure it. The different servo makers have different gear sizes and number of teeth, so the various components allow the assembler to create a correctly fitting servo horn, as opposed to one that does not fit tightly, or is too small to fit onto the gear. After connecting the servo horn, the servo was slid into the slot between the front two wheels with its horn centered and pointing straight up, and secured with two screws.

The tie rods, which connect the servo horn to the wheel, allowing the servo to control the direction of the wheels, were then assembled and installed. Because the servo gear is not centered in the middle of the servo body, the gear (and therefore the horn) was not equidistant from the two wheels. To remedy this, two different lengths were utilized for the tie rods, with the shorter tie rod connected to the wheel closest to the servo gear. To create each tie rod, three pieces were used. The middle bar, one of short length, and one of long length, was threaded on both ends, in order to attach to the mounting mechanism on either end. On the wheel end of the rod, a joint piece was attached, allowing it to receive (and hold) a ball-shaped pin. At the servo horn end, a joint with a circular hole through it was attached. A ball bearing with a hole through it was then pressed into the hole in the connector using pliers, in order to allow the rod to pivot as the servo was turned. The long and short tie rods were then connected to the servo horn by securing the ball bearings to the horn with screws on the side farthest and closest to the servo gear respectively. With the servo side attached, the rods were adjusted by further twisting them into the connectors, shortening the length of the tie rod, until the ends of the rods could be moved inside the wheels, close to the wheel hitches. Fine adjustments were then made, until it appeared that both wheels would be oriented straight when the hitches were attached to the tie rod. The boots/caps at the end of the tie rod were then attached to the wheel hitches, completing the steering unit. As necessary, the wheel orientations received fine adjustments by turning the tie rods with pliers. Afterwards, the bumper was attached to the front of the car using two tapered screws.

After assembling and connecting the steering control unit and bumper, the battery was mounted to the car body. The kit came with two 7.2V batteries, which needed to be readily swappable in case one battery became low, which could negatively affect performance. The body came with an in-place mount to rest the battery on, located beneath the suspension. To secure the battery, a battery was laid across the mount beneath the suspension, and an adjustable zip-tie was passed beneath the mount in two locations to hold the battery in place. This allowed the battery to be attached securely, while also allowing it to be quickly removed and replaced. The camera was then assembled by placing the camera lens over the line scan camera circuit board, and attaching it with 2 small metal screws. After testing the camera in a variety of lighting conditions, it was discovered that bright light bled through the back of the camera disrupting stable

reading of the line. To remedy this, black electrical tape was wrapped around the back of the camera, to prevent light from bleeding into the camera and destroying the clarity of the captured signal.

The camera was then mounted on the front of the car, referring to the Freescale Car Assembly Manual. The manual includes a 3D model of a mount that is printable using a 3D printer. The three components of the 3D printable mount are shown in Figure 1.

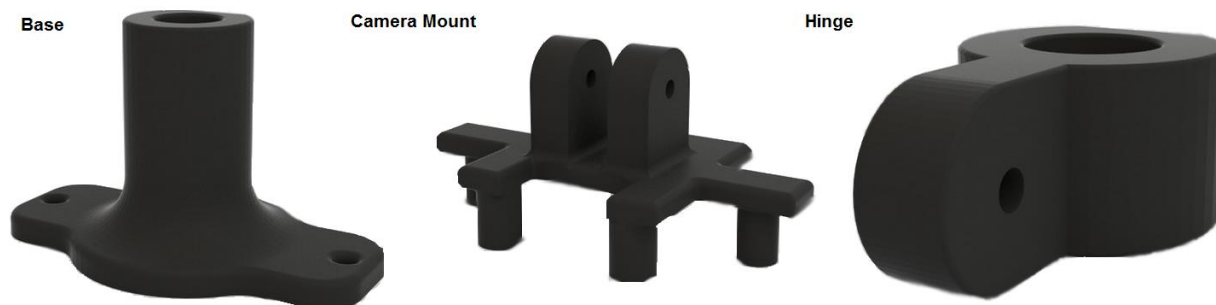


Figure 1. Three Pieces of the Camera Mount

The base component was made as the support of the device, attaching to the car, and supporting a wooden dowel, the other two printed components, and the camera itself. Once the base was printed, the resin supports created in the printing process were removed. The base was then mounted directly on top of the servo by removing the plastic plate the servo was fit under, and replacing it with the base. The base was then tightly screwed down using the same screws that once held the plate. This placement kept the camera high off the ground, towards the front, and protected from immediately bumping into something (as it might have on the bumper). Multiple wooden dowels were then filed down to fit snugly, without too much resistance, into the base, in order to provide a variety of heights for the camera. Separately, the residue from the other two components was removed as they were printed. The hinge was then slid into the slot in the camera mount component. The longest of the available screws was then screwed directly through the three layers, attaching the hinge to the camera mount. The camera was then attached via two screws, to the two screw holes closest to the camera mount component. A wooden dowel was placed into the base, and the large hole in the hinge was slid down around the dowel. A single thumb tack was used in the base to keep the dowel from moving, while another tack was used in the hinge to prevent it from turning or sliding down the dowel.

The FRDM-KL25Z microcontroller was then mounted to the Freescale car body by screwing the two yellow plastic extensions shown in the Freescale Car Assembly Manual to the bottom of the microcontroller. The reverse ends were then screwed down to the screw holes on the top rear of the car, near the rear-wheel drive motors. Once the FRDM-KL25Z microcontroller was securely connected to the Freescale car, it was connected to the camera, battery connector, servo, and motors via the FRDM-TFC Motor Control Shield. Figure 2 shows the connections (and their color codes) made to the camera, standing upon the mount, with the black side of the cable connecting to ground, and the opposite side connected to AOUT.

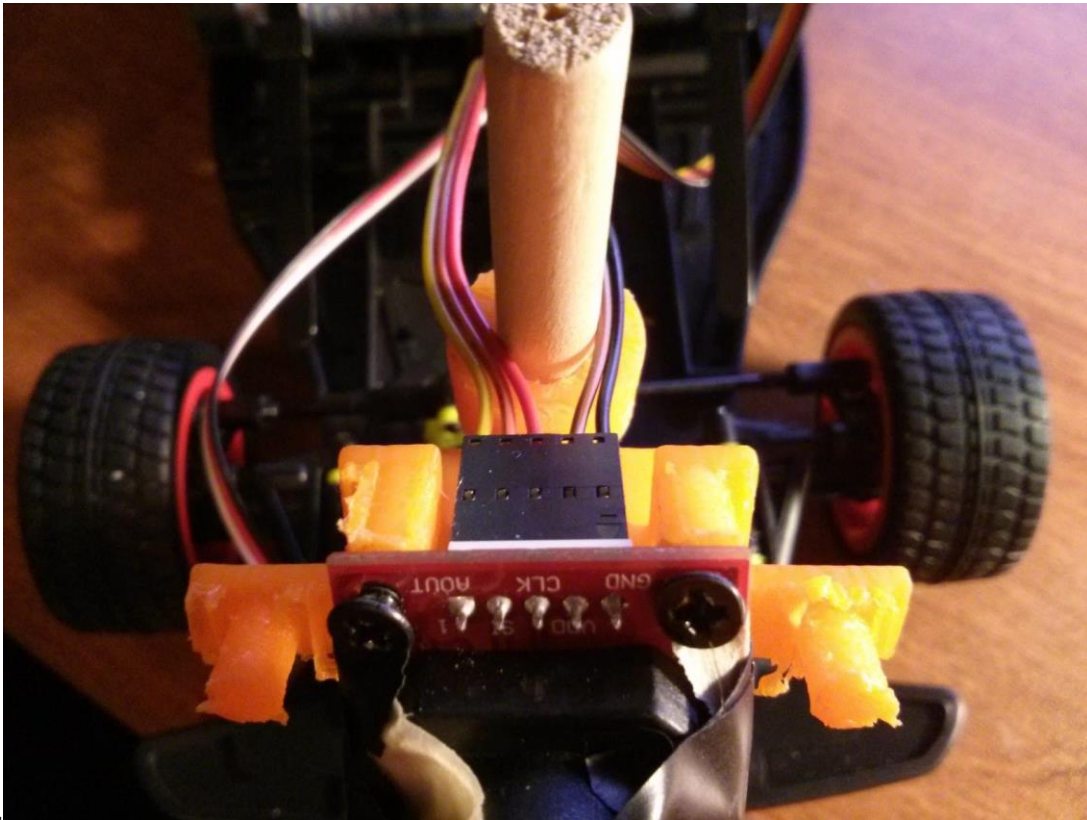


Figure 2. Pinout from Line-Scan Camera

The same cable in Figure 2 was then connected to the K25's motor shield's camera connector, as shown in Figure 3. The orientation of the multicolored cable remained the same as Figure 2, meaning that the black wire was connected to ground, and the yellow cable was connected to the AOUT of the camera.

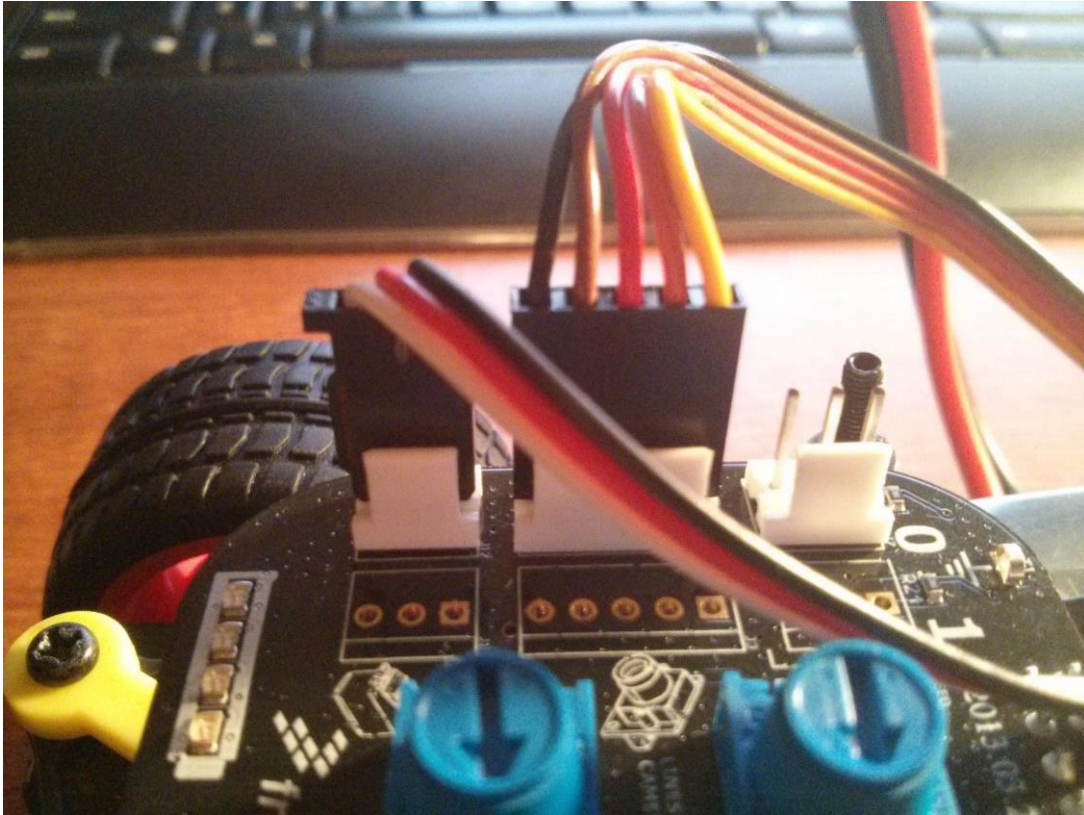


Figure 3. Camera and Servo Connectors on Motor Shield

Figure 3 also shows the smaller servo connector, as it was connected to the motor controller shield. It was connected with the ground pin (black) towards the interior of the board. After connecting the servo and camera, the motors and battery connector were connected to the motor controller shield. The other connections were made using screw terminals on the right side of the car. The battery connector was two stranded core wires, loose at one end, and part of a male connector (matching the battery's female connector) on the other end. The red wire of the battery connector was connected to the V_{BAT} terminal, and the black wire was connected to the respective ground terminal, denoted GND (both at the center screw terminal). The left motor's positive and negative terminal (red and black wires respectively) were connected to the B screw terminal, with the ground terminal connected to the terminal denoted '2', and the positive/live wire connected to terminal '1'. Similarly, the right motor was connected to the A screw terminal, with the red wire connected to the terminal denoted '1', and the ground wire connected to the terminal denoted '2'. Figure 4 shows the connections to the battery connector and motors, with the right motor's connections on the left, the left motor's connections on the right, and the battery connector connected in the center.

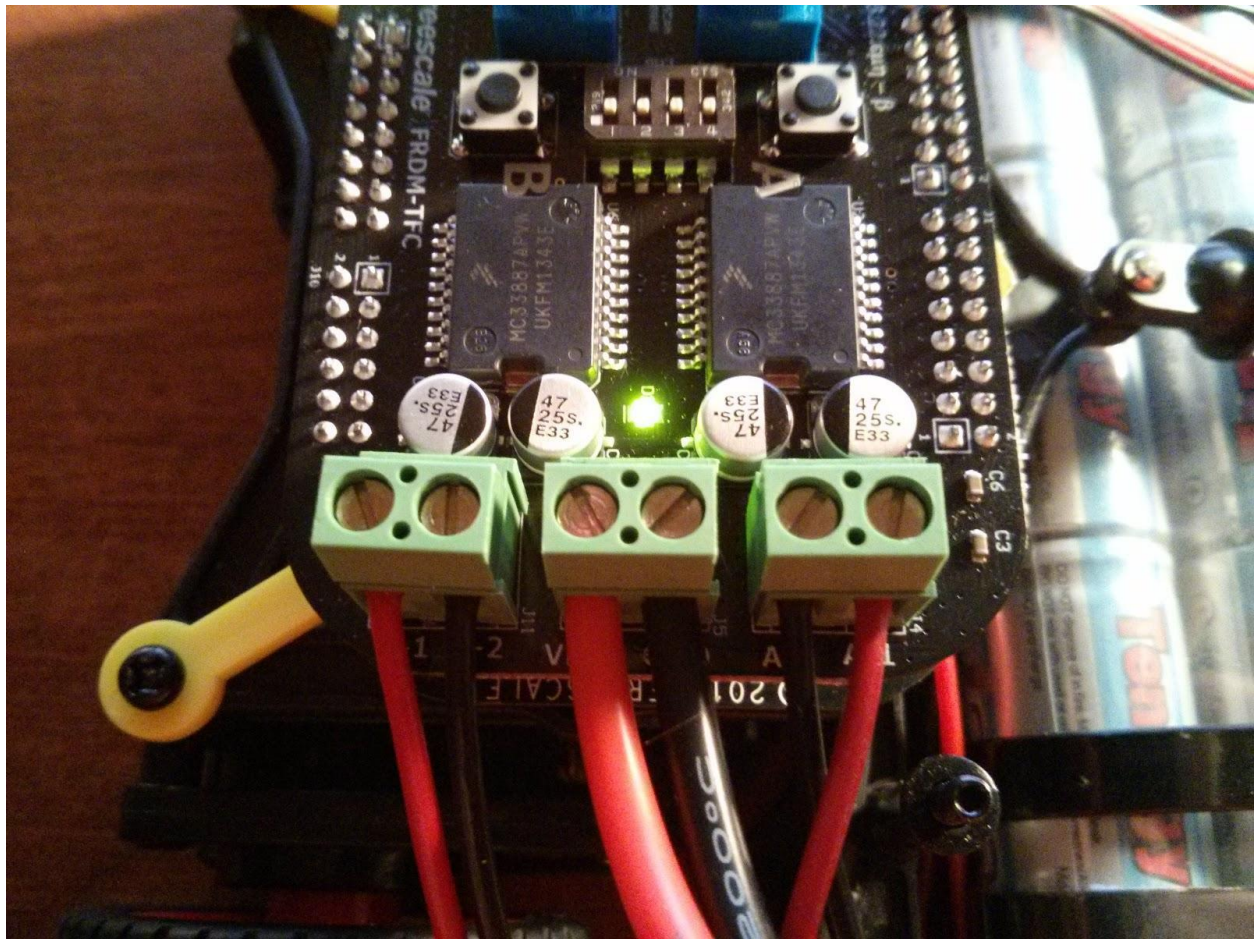


Figure 4. Motor and Battery Connections on Motor Shield

The battery could then be connected to the K25 by connecting the battery's female connector to the newly attached battery connector on the K25. With the battery, motors, servo, and camera all mounted and connected, the Freescale car had finally been assembled, and was ready to program. While an assembly or C language implementation of a controller may have been more lightweight and produced higher performance, the rapid prototyping/development offered by Simulink through Matlab proved to be the more practical option given tight time constraints.

To accomplish the necessary Matlab programming, Matlab had to be requested from the Mathworks Freescale Cup page. The link to this download is provided in reference document 3. The option to "request software" was selected and an account was created. Upon successful completion of the necessary forms, a MathWorks license was created and provided via email. To download the software, the previously created Mathworks account was associated with the provided license and activation key. This was done following the steps provided by reference document 4. Once associated with a license the software was then downloaded and installed to the computer. Download instructions are provided in reference document 5. Upon successful installation of the Matlab software activation was required. This activation occurred automatically as MathWorks prompted for account credentials. While problems did not occur, MathWorks

acknowledges that problems can occur with automatic activation due to firewall settings. Reference document 6 provides details for performing a manual activation of Matlab software. A manual activation may also be necessary for a computer that does not have an internet connection. At this point Matlab was successfully installed and was fully operational. In order to interface with the Freescale FRDM-KL25Z board, the embedded coder support package was downloaded and installed. The support package can be found on the MathWorks website, and is shown in reference document 8. Double clicking on the file opens Matlab and launches the Support Package Installer which provides step by step instructions for installing the support package.

The Embedded Coder Support Package for the Freescale FRDM-KL25Z allowed for code to be automatically built and generated. The embedded coder supported the following peripherals: 14 digital I/Os, 6 analog inputs, 1 analog output, 3 serial Tx/Rx lines from on-board UARTS, and an RGB LED. The support package also provided library blocks to interface the TFC-Shield which included the following peripherals: a line scan camera, dipswitch, two potentiometers, two push buttons, two servos, and two DC motors. Before generating code on the board it was important to download the proper drivers for the Freescale FRDM-KL25Z, and to verify that the proper bootloader was operational on the Freescale FRDM-KL25Z board.

In order to program the car, a Simulink model was created using the embedded support package for the FRDM-KL25Z board. Simulink was used to build, load and run the created model directly on the FRDM-KL25Z platform. Simulink did this by creating C code which was compatible with the FRDM-KL25Z. A binary file could then be loaded onto the FRDM-KL25Z board thus programming the car.

The version of Matlab used was R2014b. The following provides the necessary steps for allowing the Freescale car to be programmed from the Matlab environment. It is important to note that to perform a firmware update, a computer running Windows 7, or XP is necessary. A PC running Windows 8 will not work to upgrade the FRDM-KL25Z firmware.

1. Verify that the Freescale car contains the proper firmware. This is done by connecting the car to the FRDM-KL25Z board via a USB mini type B cable. Using File Explorer, navigate to the FRDM-KL25Z device. Open the sda_info.htm and verify that the bootloader is 1.11 and the application version is 1.14. If both these parameters are correct then as of December 7, 2014 the FRDM-KL25Z board will be compatible with Matlab R2014b. Figure 5 shows the proper settings.
2. If the application version is not correct follow the steps of reference document 8 to perform the update to the latest version.

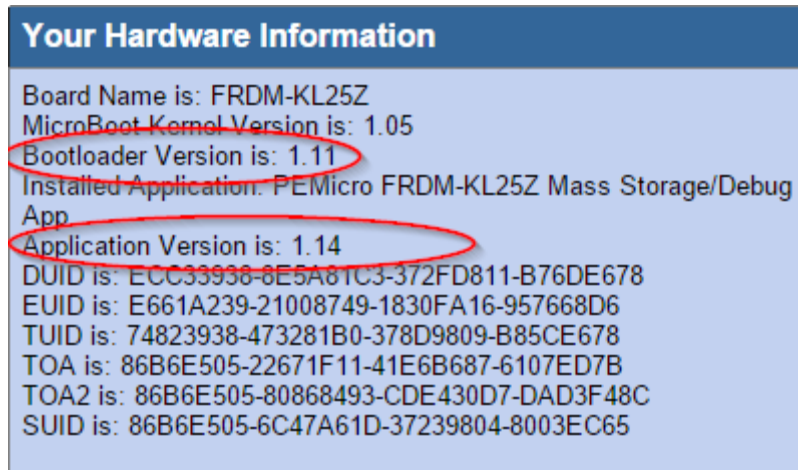


Figure 5 - FRDM-KL25Z Hardware Configuration

3. The Freescale FRDM-KL25Z serial drivers then need to be installed to allow for serial programming of the microcontroller. First download and install the P&E OpenSDA USB drivers from the link provided in reference document 9. Download the PEDrivers_install.exe for Windows to perform a manual install. At this point you will be prompted to create an OpenSDA account.
4. Connect the USB cable from the computer to the FDRM-KL25Z board. The board should appear in File Explorer as a storage drive labeled FRDM-KL25Z. Verify that the board appears by going to the device manager and finding the FRDM-KL25Z entry under portable devices as seen in Figure 6.

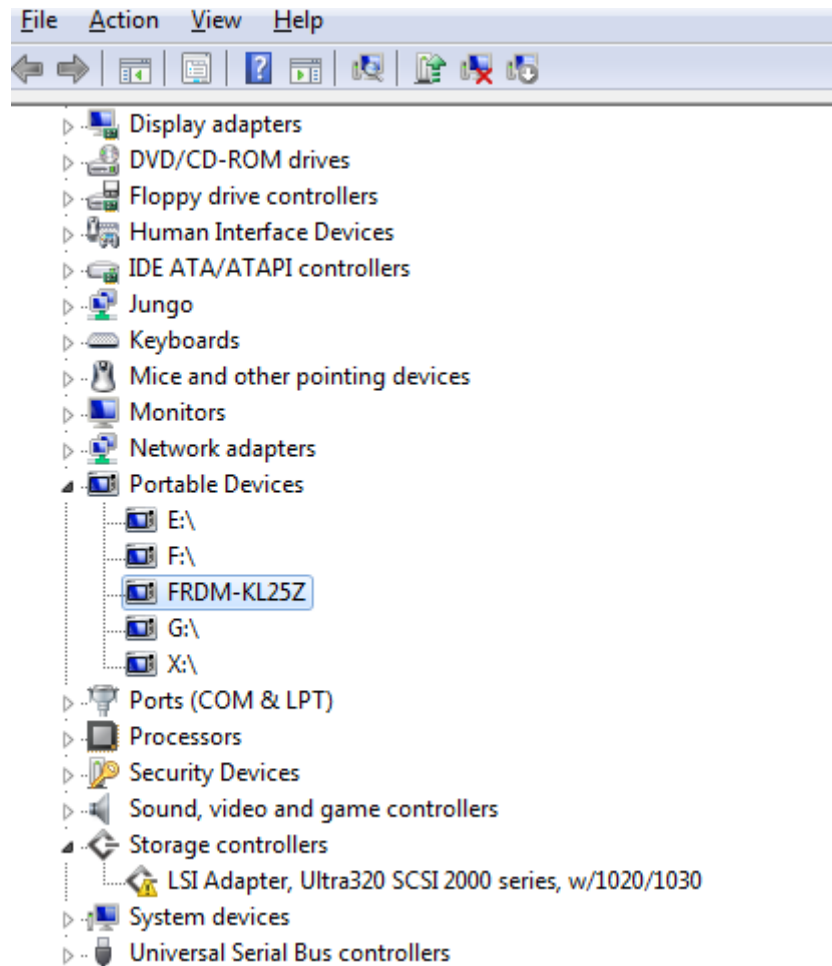


Figure 6 - FRDM-KL25Z Hardware Configuration before Downloading Serial Drivers

5. From the Device Manager go to Ports and check if "OpenSDA - CDC Serial Port" is available. If so then the driver installation is complete. "PEMicro/Freescale - CDC Serial Port" should be visible as shown in Figure 7.

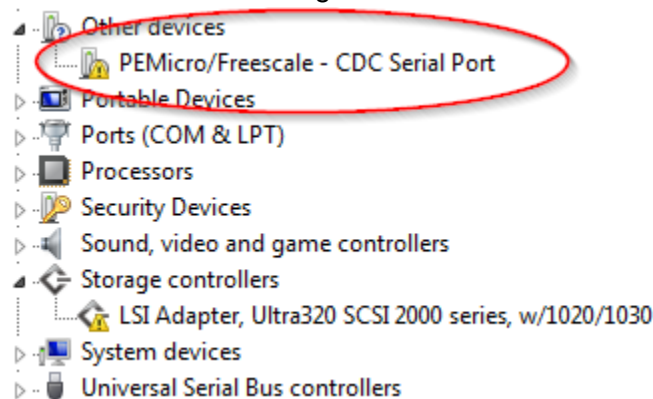


Figure 7 - FRDM-KL25Z Hardware Configuration before Installing Drivers

6. To update the drivers right click on “PEMicro/Freescale - CDC Serial Port” and select update driver software. Browse to the location which the previously downloaded driver files were saved and point to this location thus allowing for location of the updated serial drivers.
7. This completes the driver installation and the FRDM-KL25Z should appear as a serial port under Ports with no yellow exclamation point as shown in Figure 8.

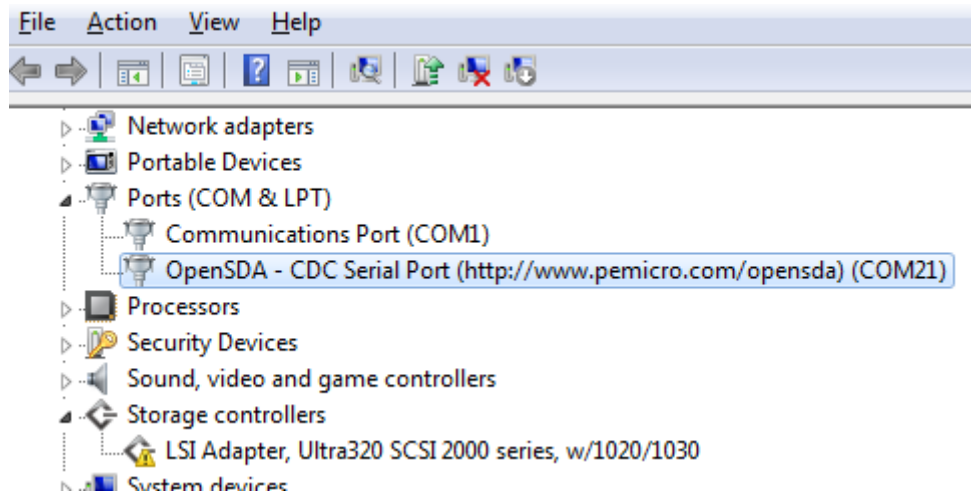


Figure 8 - FRDM-KL25Z showing proper install of Serial Drivers

At this point a simple provided example project can be run on the hardware. To accomplish this open the provided Simulink example project. This is done by selecting the APPS tab from Matlab and then selecting the Freescale Cup Companion application. This will provide a pop-up, at this point select “Show Examples”. Select the first example which is titled, “Getting Started with Freescale FRDM-KL25Z Support Package”. This document can also be found as reference document 10. This reference document provides a preconfigured model which can be loaded onto the car to light the LED. If problems occur, make sure drivers are up to date and simulation settings are exactly as specified in reference document 10. At this point, the LED located on the FRDM-KL25Z should be illuminated with a red color.

The developed algorithm was based heavily on classic control theory. A closed feedback loop was used, as the measured position relative to the line was compared to the desired position relative to the line. This allowed for the generation of an error signal which was then fed into the controller. The controller compensated and amplified the error signal, in order to produce a position command. Figure 9 shows a high level block diagram of the controller which was used. The Simulink model contained two major sections: steering controls, and speed control.

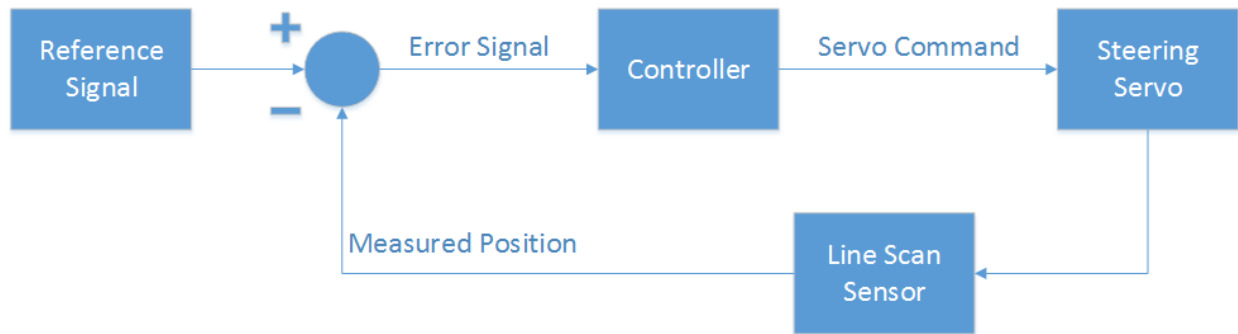


Figure 9 - High Level Control System

Steering Control

The provided line scan data was accessible in Simulink by instantiating a line scan data block and choosing the sampling rate. This data was provided as a 128 by 1 array of pixels. Values represented the intensity of sampled data where 65,536 indicate bright area (such as the white line) and a value of 0 represented a very dark area. The index of the array corresponded to a particular pixel. For example at index 1 the provided data was for pixel 1. For more information on the line scan camera see reference document 2. A sample of the digital output provided by the Simulink line scan camera is shown in Figure 10.

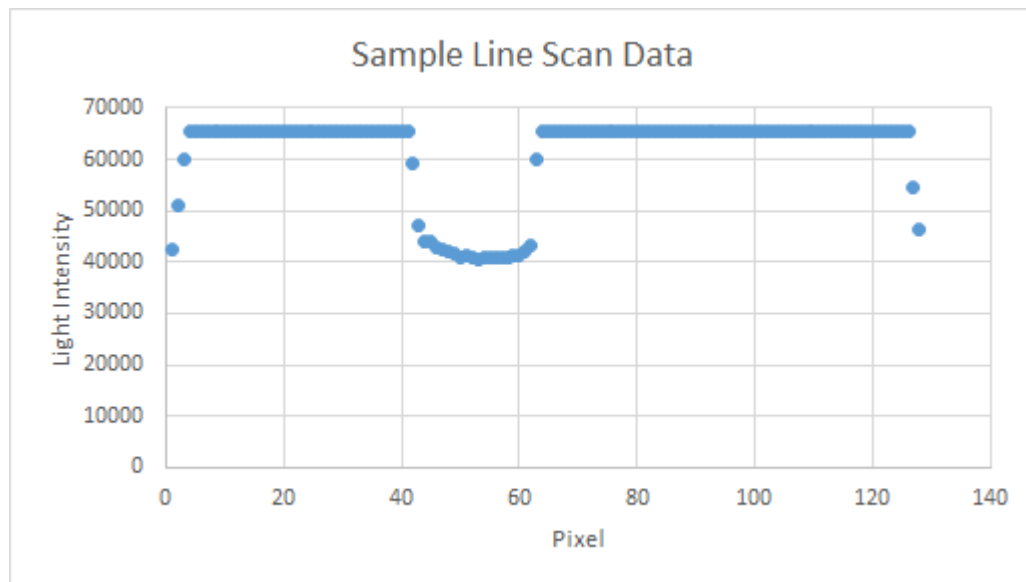


Figure 10 - Sample Raw Line Scan Data

To process the line scan data the following algorithm was generated to process the camera data and determine at which pixel the center of the line was currently located at. The algorithm steps are as follows.

1. Truncate 10 bits of each side of the line scan camera data to account for the lens which covered the outside pixels.
2. Take the derivative of the truncated data using the definition of the derivative given by Equation 1. This allowed for the generation of a new vector which contained the rate of change around each pixel. With this information the edges of the black line could be more readily determined. A sample derivative generated vector is shown in Figure 11.

$$f(x) = (f(x + 1) - f(x - 1))/2 \quad (1)$$

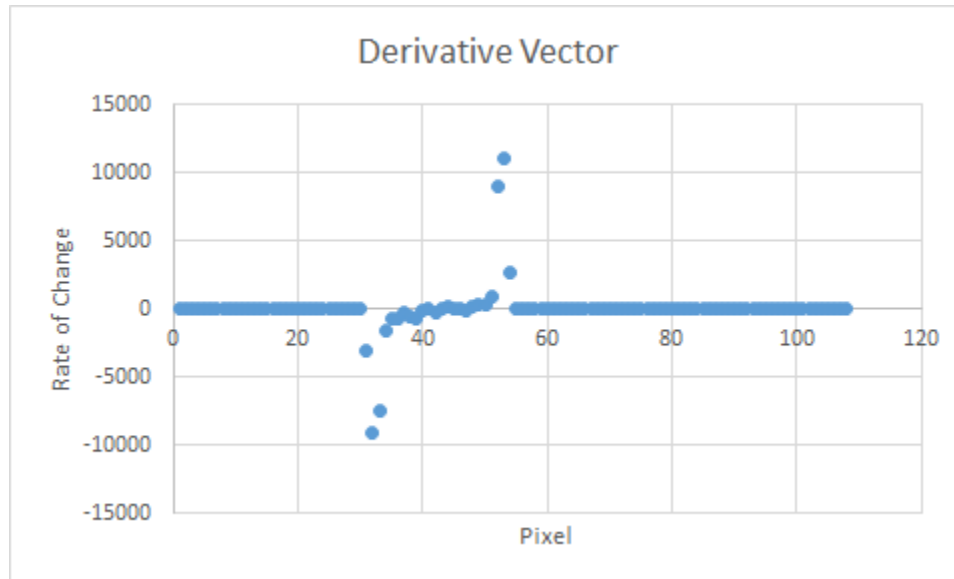


Figure 11 - Derivative vector generated from the raw data of Figure 10

3. The minimum and maximum values of the derivative data were then determined to determine the location of the two edges of the line. The derivative data was at a maximum at the edge location, due to the contrast that occurred between light and dark at these locations.
4. The center point of the line was then determined by taking the average of the two edges. Thus this algorithm assumes that the center of the line is in the center of the two edges.

This algorithm produced the pixel value for which the line was currently centered on, i.e. the cars' current position. This information was used to generate an error signal by taking the ideal center pixel (128/2 or 64) and subtracting the actual position found using the derivative method. This error signal was then fed into a PI controller.

The purpose of the PI controller was to take in an error signal and use this information to make a decision on how to command the steering servo. To implement the controller the DC servo parameters were first experimented with in order to understand the available range of motion. From this experimentation, all the way right, and all the way left values were found. These physical limitations were imposed on the controller thus preventing the command of an impossible position on the wheels. The PI controller was generated using a built in Matlab PID controller

block. The controller was carefully tuned to obtain optimal results. The manual tuning method was used until performance was occurring as desired. A value of 0 was chosen for the derivative control, shutting off the derivative term due to the jerky effect on the wheels with a derivative control term. The proportional gain was set to 0.6. A proportional gain higher than this value created a stability problem as the system oscillated around the value. A proportional gain lower than this reduced the ability of the car to respond to changes in the system, which could mean losing the line at a drastic change. An integral gain of 0.1 was used. The integral gain allowed the proportional gain to be lowered to dampen the oscillations around the line experienced by the car. The integral gain also helped to correct for long term error which was specifically important when the car was navigating a loop. It was also important to enable anti-windup for the integrator. Integrator windup refers to a situation in which a large error is generated which saturates the controller output and leads to overshooting when correcting thus leading to a system which becomes quickly unstable. This problem was mitigated by initializing the integral to a desired value and also only allow for the operation of the integral part of the controller while the car was in the controllable region.

Speed Control

The implemented speed control was also implemented through the use of a controller fed by the same error signal as the steering control. When zero error signal was encountered, both back wheels were powered with the same command. When an error signal occurred the controller determined in which direction the car needed to turn and powered the back wheels accordingly thus reducing the error signal. For example if the error signal indicated a turn to the left was necessary, the right wheel was powered with maximum power, while the left-wheel was underpowered. This allowed for a sharper turning radius to occur and also provided basic variable speed control. While a more complex algorithm could have been used, the differential drive linearly decreased the power provided to the motor that needed slowed. A constant, considered a minimum, was used in the Simulink model to specify the lowest percentage of the 'full' (20 % of full power in the demo) drive power that any wheel could be slowed down. In the demo, a maximum drive power of 80% was specified, meaning that the slowed wheel would only be driven at 80% of the local maximum of 20% of full power in a worst case turn. To implement this, the power provided to the hindered wheel was calculated as a linear function of the difference between the local maximum (20% of full power), and the minimum differential drive power (80% of 20% equals 16% of full power), by reducing the maximum drive by 1/64th of difference between the maximum and minimum for every 1 unit in the error function (i.e. $\text{max} - 20 * [\text{max} - \text{min}] / 64$, for an error of 20). A linear function was used in order to provide smooth compensation, where the car should only be compensating for a small difference in the error, whereas exponential functions may appear non-smooth, and allowing error to build until it makes a large correction. Thus the linear function could help prevent oscillations by reacting to early/small errors and compensating for them.

Simulink Model

Figure 10 shows the complete Simulink system diagram. The steering controller is shown on the top line of Figure 10. This has the line scan data being generated using an exposure time of 10 mSec with a sample rate of 50 mSec. The output data from the line scan camera block is fed to a data type conversion block to provide the correct data type to the following blocks. The calcPos block calculates the current position using the derivative method. This position is fed into the controller block which calculates the steering servo command to generate. The second major controller is the speed controller shown in the middle of Figure 10. This has the error signal fed into the Differential Drive Controller block and drives the two DC Motors (left and right wheels). At the bottom of the model, a battery read and battery indicator is shown for convenience such that an indicator is provided regarding the life of the battery.

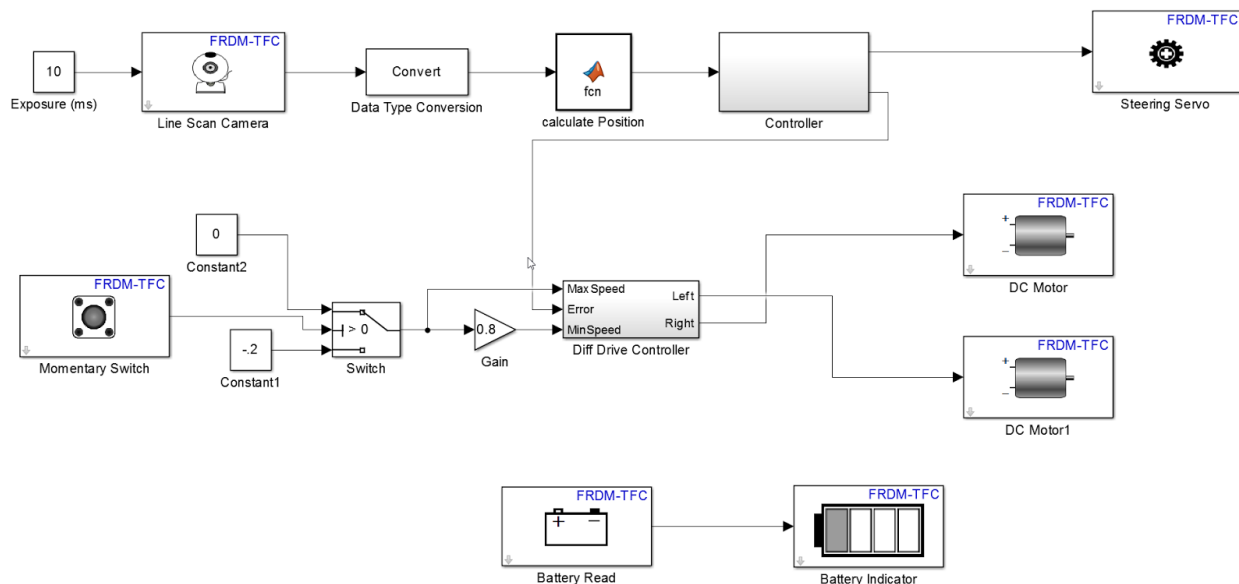


Figure 10 - Simulink Model

The calcPos block shown in Figure 10 performed the derivative method on the provided line scan camera data. In this block a derivative vector was created and was populated with the rate of change around each pixel according to Equation 1. From the derivative vector, the maximum and minimums of the vector were determined. These values were understood to be the left and right edges of the black line. To determine the center point of the line the two edges were then averaged together. The code implementing the calcPos block is shown in Figure 11. The input to this block is camData, which is the 128 by 1 array from the line scan camera. The output of this block is a single pixel value at which the line is currently centered around. The range of the current position value output is from 0 to 128, with 64 being the steady state centered on the line position.

```

function calcPos = calcPos(camData)

% Take the derivative
deriv = zeros(1,128);

% Create the derivative vector
for i = 11:118
    deriv(i) = (camData(i+1)-camData(i-1))/2;
end

[~, pixel_max] = max(deriv);
[~, pixel_min] = min(deriv);

calcPos = (pixel_max + pixel_min)/2;

```

Figure 11- calcPos Implementation

The calculated position produced by the calcPos block was then fed into the steering controller. The implementation of the steering controller is shown in Figure 12.

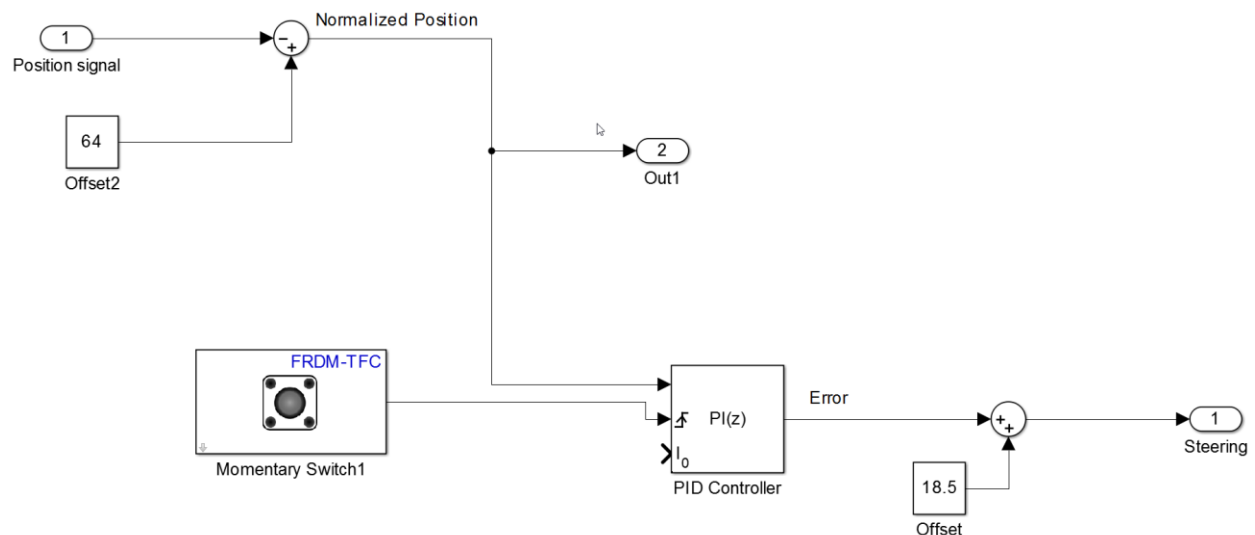


Figure 12- Implementation of Steering Controller

The input to the steering controller is the calculated position of the line. The position signal was normalized by subtracting an offset of 64 from the actual car position. The value of 64 was chosen as it corresponds to the centered position and the offset subtraction normalizes the position signal to zero when centered. If the line was perfectly aligned, then the center of the line should be in the exact middle of the 128 pixels of the line scan camera. The normalized position signal was then fed into a PID controller block. It is important to note that the PID controller must be operating on a discrete time domain to allow for code which can be compiled. The PI controller also took as input a reset which allowed for an external reset of the controller. This allowed the car to ignore any random noise which may have occurred before the car is on the track ready to

begin the course. By resetting the car right before the car began the course, the controller was able to begin in a known state allowing for deterministic behavior. The output of the PI controller was a signal, centered on 0, which could be used to command the servo. Positive values from the PI controller requested a right turn while negative values corresponded to a left turn. A value of 0 corresponded to the car continuing straight. Due to the mechanical installation of the servo, it was determined that an offset of 18.5 was required. Therefore, to account for the offset, 18.5 was added to the controller output to allow for the PI controller commands to translate to the necessary servo command. The output of this controller was then sent to the steering servo. It is important to note that the steering servo was programmed with hard-coded maximum and minimum limits. These values prevented the servo from turning thru an angle that was not physically possible. If the controller was to command a value outside of the acceptable range, the servo would only move to its hard-coded limits.

The DC motors A and B were powered by the Diff Drive Controller block shown in Figure 10. The differential drive controller had three inputs: maxspeed which indicated the maximum wheel speed, error which indicated the error generated by the steering controller (shown in Figure 12), and minspeed which indicated the minimum wheel speed. The outputs of this controller are the motor speed values for both the left and the right DC motors. It is important to note that the momentary switch shown in Figure 10 was used to switch between accepting an actual value. This allowed for the car to be placed into a hold state at which no power was sent to the wheels. This proved ideal as it was used to prevent the cars' wheels from turning when not on the track. The differential drive controller consisted solely of the Matlab function shown in Figure 13. The Matlab function produced a slowed wheel speed based on the error signal. If an error signal was non-existent then the speed would be equal to the maximum speed thus allowing both wheels to turn at the same speed. After computing the slowed speed signal, it was determined which wheel would get the max signal and which wheel would get the slowed signal. This decision was based on the sign of the error function. If the error function was positive, the line must be to the right of the car, and the right wheel is slowed. This will allow for the left wheel to drive harder than the right wheel thus allowing for the car to turn sharper to the right. If the error signal is negative, then the line must be off to the left. Therefore, the slowed wheel is set to the left wheel thus allowing the right wheel to be driven harder than the left wheel allowing for the car to turn sharper to the right.

```

function [left,right]= fcn(error, max, min)
%#codegen

slowed=max - abs(error)*((max-min)/45);

if (error <=0)
    % Line is on right

    % Maximum to left drive
    left = max;
    right = slowed;

else
    % Line is on left

    % Maximum to right drive
    right = max;
    left=slowed;

end

```

Figure 13- Differential Drive Controller Implementation

Results

The car was tested on a single track that included turns of various sizes, two straight portions, and an intersection. A section of the track can be seen in Figure 14.



Figure 14. A Portion of the Test Track

An iterative process was used to tune the parameters of the controller to allow for the car to make it around all parts of the track. The PI controller, for steering, had two gain parameters, the

proportional and the integral, that needed to be dialed in to the correct value. In addition the differential drive had a parameter that controlled how much a wheel was allowed to slow down, which also needed to be tuned. The process involved a lot of trial and error, in that, a parameter value was changed, the car was tested on the track, and, if it was unable to make it around for some reason, the process was repeated. Some errors that were seen included the car losing the line while going around a turn because it was either not turning sharp enough or was turning too sharp. The PI controller causing the car to begin to turn back and forth (oscillate) until it eventually lost the line, or the error in the PI controller becoming too high (due to integral windup), causing the controller to tell the car to turn a certain direction indefinitely.

Eventually the controllers were tuned well enough for the car to be able to make it around the track. To complete the track a constant, normalized speed of 0.2 was used. This meant that each DC motor was running at a maximum of 20% of the maximum speed. On the straight portions of the track the car maintained a constant speed and traveled in a simple straight line. Since there was no turn, the speed was simply the maximum speed of 0.2. As the car approached a turn, it would turn back and forth slightly at first and then begin to make the turn. This back and forth motion was caused by the PI controller overcorrecting the turn until it eventually evened out. As the turn progressed the differential drive controller would slow one wheel slightly which caused the car to slow down slightly during the turn. As the car left the turn, there was another small section of oscillation, in some cases, and then the car would straighten out and resume the constant speed. Overall, the car was able to make it around the entire track in around 44 seconds. While this was fairly quick, it could have been much faster because the maximum speed was only 20 %.

Analysis

During the design, development and testing of the Freescale Car several issues were discovered. These issues are mentioned here in order to mitigate the risk of those attempting to duplicate the obtained results.

The 7.2V car battery should not be applied to the car during USB programming. During the programming process, Simulink was seen to set various processor outputs during the programming process non-deterministically. This created situations in which the processor outputs controlling the H-Bridge of the DC motors were active. Therefore if the battery was enabled, the DC motors would spin at a maximum speed. The same behavior was seen with the steering servo as it moved non-deterministically during programming.

Various issues were also found with the line scan camera which greatly affected the performance of the data received from the car. It was determined that the line scan camera was very sensitive to light being exposed to the backside of the camera. This was noted in reference document 2 and was also found in the testing of the data. This issue was mitigated through the use of electrical tape being placed over the back of the sensor. The line scan camera also required a well-illuminated area such that it could provide accurate information regarding the

contrast of what it was seeing. While an LED headlamp was not used, it is recommended to mitigate this as an issue. The parameters of position of the camera relative to the line, exposure and focus of the camera were also found to be important. To tune these parameters to provide for best results an oscilloscope was used. Each parameter was then tuned until the best possible results were obtained. The characteristics of a good waveform included: a sharp contrast between light and dark areas, low levels of noise, and a reasonable line resolution. The exposure time affected how long the camera took to complete a sample. Due to the relatively low levels of light available it was important to use a relatively high exposure time. By guaranteeing a quality light source the exposure time could be lowered. This provides an advantage as the algorithm could then be run at a faster rate thus allowing for the car to navigate the track at higher speeds.

The current algorithm has several limitations which restricted the operation of the car. Due to the speed of the processor the control algorithms could only be run every 50 mSec or 20 times a second. This limited the speed at which the car could be run as the car was able to cover too much distance between sampling its position and the system would quickly become unstable, thus leading to the loss of the line. Also currently, for deterministic behavior to occur the line scan camera must be able to see both edges of the line. If the car loses either of the two edges, the second edge will be determined by random noise thus leading the car to operate randomly. This behavior could be improved by adding logic to handle the case at which an edge has been lost. By adding edge loss detection, the controller can take this into account and when making control decisions. Also a rate limiter, could be implemented to prevent the commanded value from changing any faster than a specified limit. This would be useful in terms of when the servo is commanded by the controller to move from control limit to control limit instantly. This behavior is not mechanically possible as it is not possible for the car to move from one edge of the line to the other in 50 mSec, and therefore the steering servo should ignore these commands.

Being new to Simulink, much time was spent handling issues in understanding the tool. It is important to note the variable types of blocks. If types do not match, depending on the block Simulink will not produce an error message but will instead interpret an input to a function as a type that it is not. This leads to non-deterministic behavior so it is very important to make sure the output type of one block matches the input type of the block that it goes too. The use of convert blocks can also be used to convert between data types. Another Simulink problem that occurred was attempting to run the update algorithm too fast for the current processor. While Simulink did not generate any errors, when run in hardware the code was unresponsive. While never verified with 100% certainty, it is assumed that the processor was not given enough time to handle its interrupts before it was again being interrupted. When the update speed of the algorithm was reduced normal behavior was once again observed.

The implemented controller consisted of a PI controller only. A derivative part was deemed not feasible due to the nature of the derivative controller as it would react to noise leading to a jittery behavior. In order to make derivative control more feasible, the error signal must be filtered in way to eliminate frequency components that are not possible. By implementing a filter, derivative control can be used, which will thus allow for a large increase in car performance. Another possible improvement to the controller would be to add in feed-forward control. This

would allow for the controller to anticipate changes and create a turn command ahead of time. For example if a second camera were to be acquired it could provide with a further look ahead to see upcoming turns.

During the implementation and testing of the car, the need for various improvements was realized. These improvements would be needed to allow for the car to travel faster around the track. The first improvement would be the addition of wheel encoders for each wheel. Currently, the speed of the car was affected by battery voltage because it was being controlled with a PWM signal with a constant duty cycle. The battery voltage varied from around 8.4 V to 6.4 V based on the level of charge. This meant that a duty cycle of 20% would drive the motor faster if the voltage was 8.4 V versus if the voltage was lower than 8.4 V. The addition of wheel encoders would allow the speed of each wheel to be accurately measured and controlled independent of battery voltage. Another improvement would be the addition of another camera. Another camera would allow for the car to look further ahead to know if a turn is going to occur in the near future. This would allow for some predictive speed control, meaning the car could travel faster on the straight portions and slow down for turns. To go along with this upgrade, a better speed controller would need to be implemented that could interpret the data from the second camera and alter the speed appropriately. The new speed controller would also have to include some electrical braking, meaning turning the motors in reverse in order to slow down faster. This would be necessary if the car was traveling very quickly into a turn and needed to slow down before the turn. It would also be helpful to allow some of the parameters to be configured with various switches on the car rather than reprogramming the car just to change a couple controller parameters.

Another improvement that would help to drastically improve performance would be to create a more complete and tuned model and controller in Simulink. If an accurate model was constructed, which allowed for accurate simulations within Matlab the current steering controller could be converted to a more precise PID controller which would allow for more fine tuning in the steering control. The current motor controller would also need to be upgraded to handle the input from the wheel encoders and use PID loops to determine an output for each wheel. In addition, if the complete system was modeled in Simulink, it could be used to precisely tune each of the controllers. All of these improvements would probably lead to the need for a faster microcontroller as well. The K25 only has a clock speed of 48 MHz and is easily overwhelmed if it is trying to do too many things at once. A faster processor would eliminate any bottleneck caused by the processor. In addition, a faster processor would allow for the sample time of the camera(s) to be set much lower meaning that the steering servo and the motors could be updated much more often. This would allow for the car to run faster because it could react to changes in the track faster.

Conclusion

The purpose of this project was to build, program, and demonstrate an autonomous, line-following car. Due to the limited time of three weeks to complete this project, the minimum requirement was that the car had to complete the course unassisted. With the minimum requirement achieved, the remaining time was directed towards making the car move as fast as possible, without any significant changes. The result was a car that moved at a maximum of 20% of its maximum possible speed, and completed the track in 44 seconds. In order to attain the fastest possible time, the car must be able to move at the highest percentage of its maximum speed. The use of a faster processor, better steering control, and more efficient algorithms are key to achieving a faster track completion time. Overall this project provided great experience in hardware, software, and the interfacing and the interactions between the two.

Appendix

The reader is referred to other design documents and resources for additional details regarding the Freescale car.

final_demo.mp4 - A video demonstrating the successful completion of a test track performed by the autonomous car.

freescale_pid.slx - The Simulink model which was used to program the car.

Reference Documents

The reader is referred to other documents for additional details from which the Freescale Car design and development is heavily leveraged. These documents may be specifically referenced or just generally useful.

Car Assembly Documents

1. Freescale Car Assembly Manual
<https://community.freescale.com/docs/DOC-1014>
2. Freescale Community - Line Scan Camera Use
<https://community.freescale.com/docs/DOC-1030>

Matlab Documents

3. Freescale Cup - Matlab Examples, tutorials and software request
<http://www.mathworks.com/academia/student-competitions/freescale-cup/>
4. Matlab Central Associating a license in the License Center
<http://www.mathworks.com/matlabcentral/answers/102871-how-do-i-associate-myself-to-a-license-in-the-license-center>
5. Matlab Central Download MATLAB products that I am licensed for from MathWorks website
<http://www.mathworks.com/matlabcentral/answers/98268-how-do-i-download-matlab-products-that-i-am-licensed-for-from-mathworks-website>
6. Installing MATLAB on a Computer that Doesn't have Internet Access
<http://www.mathworks.com/matlabcentral/answers/105854-how-can-i-install-and-activate-matlab-on-a-computer-that-doesn-t-have-internet-access>
7. Freescale FRDM-KL25Z Microcontroller Support from Embedded Coder
<http://www.mathworks.com/hardware-support/frdm-kl25z.html>
8. OpenSDA Firmware Update Instructions
http://developer.mbed.org/media/uploads/chris/updating_the_opensda_firmware.pdf
9. OpendSDA Support
<http://www.pemicro.com/opensda/>

10. MathWorks - Getting Started with Freescale FRDM-KL25Z Support Package

<http://www.mathworks.com/help/supportpkg/freedomboard/examples/getting-started-with-freescale-frdm-kl25z-board-support-package.html>