DEVELOPER GUIDE

Lab Topics and Additional Information

RC Camera Car
P14226

Tim Southerton
Brian Grosso
Alex Reid
Kevin Meehan
Lalit Tanwar
Matthew Morris

I. Chassis Construction and Information

A. Assembly Instructions

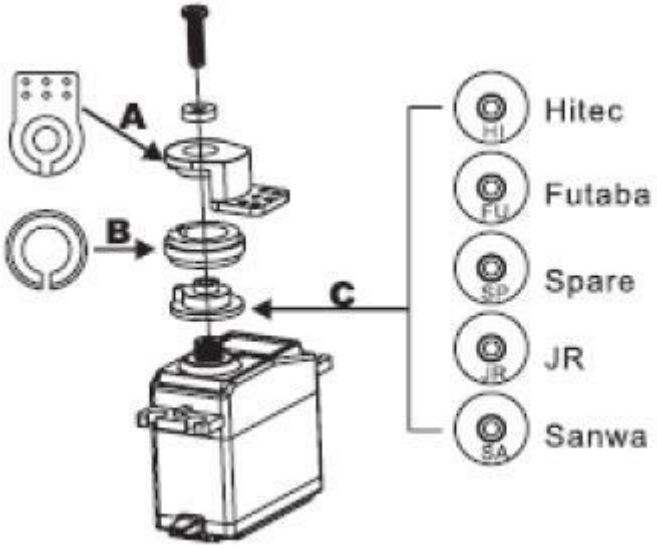
For assembling the Freescale Car Cup chassis, the directions can be found on the site below, which is part of the Freescale community site where all sorts of helpful information can be found. Unfortunately, not all of the directions are that helpful, so some are copied here and modified to give extra clarity.

General Freescale Community Site for Freescale Cup Car Assembly

Assembly of the car focuses on three main aspects, namely mounting the steering servo, mounting the freedom board (KL25Z) to the car, and mounting the line following camera. These items will be addressed individually.

1. Servo and Steering

Inside the kit are various mounting options for different servo manufacturers. Look for the Futaba bushing of the servo as this is the most common.



Before doing anything, you want to center the servo bushing for the car on the servo. This means connecting the tab on the plastic piece to be so that it fits on the gearing of the servo pointing vertically upward WHEN THE SERVO IS AT ITS MIDDLE POSITION. The only way to move the servo to the center position is by feeding it power, ground, and a PWM signal.

The easiest way to do this is using an Arduino, but since this is not a Freescale item I will not go into much detail. Anyway, this Youtube video gives you the code you need to center a servo with an Arduino Uno, and additionally you can just modify the sample code for the Sweep Arduino Example to do this. The example is helpful because it will show you the full range of motion of the servo, giving you an idea of what can go wrong. These servos only spin 180 degrees, so if you don't mount the connecting pieces so that the wheels are straight when the servo is at the center, you might limit your turning capabilities in one direction of burn out the servo. Remove any existing servo horns and screws when you understand the servo's range of motion. When everything is centered in looks right, you should see something like the images below. MAKE

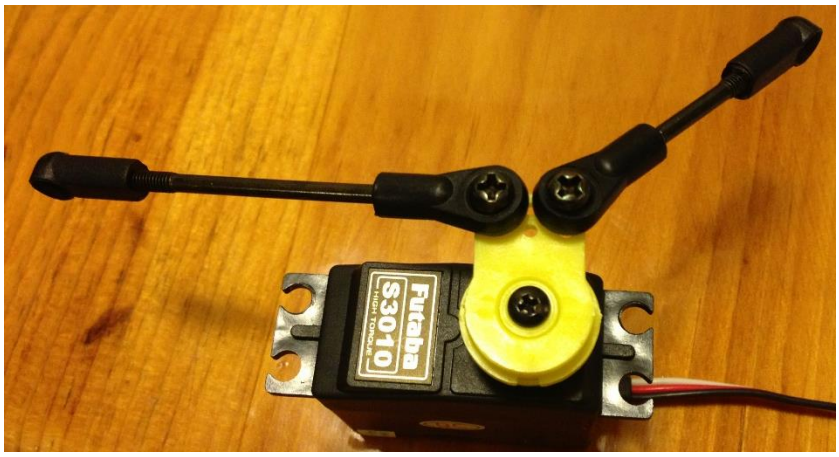
SURE THE TAB POINTS TO THE SAME SIDE OF THE SERVO AS BELOW, OTHERWISE YOU WILL HAVE TO MOUNT THE OTHER COMPONENTS IN THE OPPOSITE ORIENTATION.

You will need the pieces shown below. You will screw this into the servo. Pieces are notched, so assembly is straight forward. Mount the yellow servo plate assembly to the servo as shown below. Make sure to add the small yellow washer (pictured below) in between the servo plate and the screw. Tighten well, a servo produces a good amount of torque and will slip if not tight, but remember this is metal-on-plastic, so you can strip the connection.

Assemble the short arm and long arm. The plastic pieces are identical in this case, but the rods are of different length. This process is most easily accomplished with two pairs of pliers or a small adjustable wrench, one to grab the metal rods and the other to tighten each end individually. These will take some adjusting, so just get the threading started and leave for later.

Once finished, the metal balls need to be installed in the open plastic ends. The easiest way to do this seems to be to put the balls down on a table using the hole to sit flat, then push the plastic end down over the ball until you hear a snap. This will take some force, so alternatively do the same process with a pair of pliers.

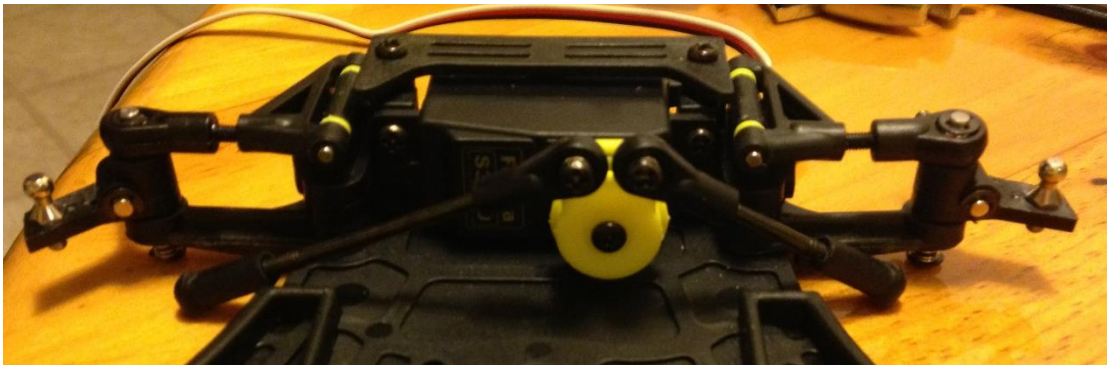
While doing this, locate the black, round-headed screws pictured. These are used for mounting the arms to the servo horn. Screw the plastic screws into the servo horn as seen below. This also takes some force, as the plastic is not threaded. If this is not going well, mark the position of the servo horn, remove the yellow pieces, and clamp that to assemble this separately before reattaching in the same orientation.



Now the servo has to be mounted to the chassis. Locate the small plastic blocks seen below, along with two more of the same screws you used to mount the servo horn arms. Set the servo flat against the table and brace (or clamp) the plastic blocks behind the set of tabs one either side. Again, with great force screw the screws into the top holes until the blocks are tight against the servo plastic mounts from the back.

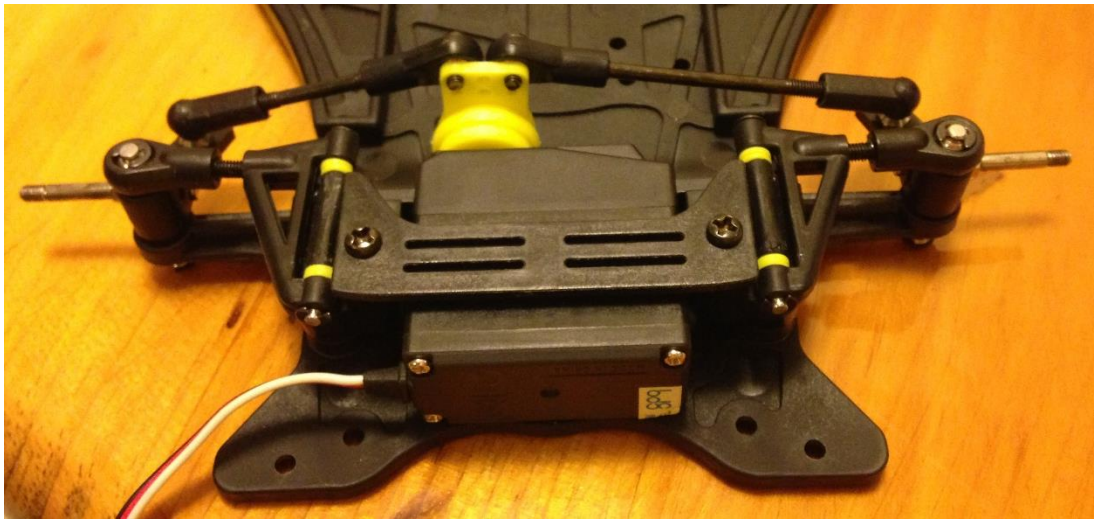
Now slide the servo assembly in from the center of the car outward and screw the attachment blocks in from the bottom of the car, as seen below. The screws used for this are the tapered head screws (all of which are the same length), which should leave a flat finish and not catch the floor.

Almost there. Now all you need to do is adjust the length of the servo arms and attach the plastic end cap pieces to the metal hitch pieces on each wheel. Here we removed the wheels using the provided metal tool. Be careful not to lose the washers that are behind each wheel nut.



The difficult part is making the wheels straight when the servo is centered. The servo arms and plastic ends have to be adjusted repeatedly to give the correct length for connecting to the hitch points, and when the plastic caps snap into place, the wheel rods will not be perpendicular to the car centerline, which is where you want them. A helpful hint can be seen in the image below, which is what you do to get the plastic caps off of the hitch points when it turns out you need to adjust the arm length. Simple put a flat head screw driver in as seen and pry CCW until the plastic cap pops off. This may mar up the plastic slightly, so don't do this many times.

If you can't get the rods perfectly straight, it is better to have the wheels slightly angled forward than backward, as this will steer more preferably. The image below shows how well we did.



Now just put the wheels back on. At this point you will probably want to mount the bumper. This is as simple as that seen below. The indent that corresponds to the plastic on the front of the chassis press together (translate the bumper upward in the picture) and the bumper is secured with screws and nuts. Depending on how well Freescale did with your set, the taped flat-headed screws that look like every other screw on the bottom of the chassis may work, but if they do not just use the same screws that you used for the servo horn arm mounts. They do not protrude much below the chassis anyway so it works and is significantly easier to mount a nut on.

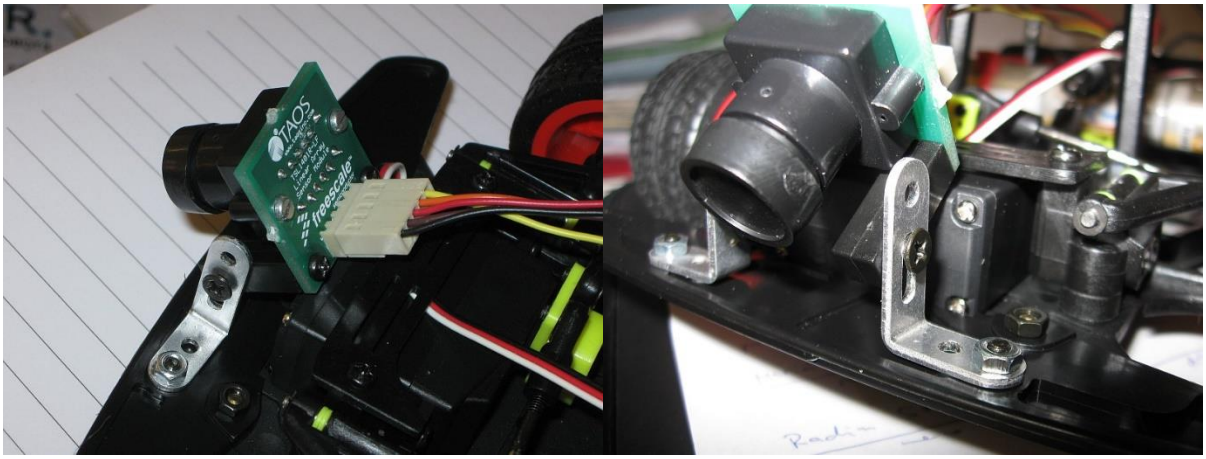
Helpful Tip: DON'T MOVE THE SERVO BY HAND OR PAST THEIR STOPS. THIS WILL BREAK THE PLASTIC GEARS INSIDE. Just saying.

2. Freedom Board

For our project, significant modification was made to accommodate significant amounts of extra electronics that are not necessary for the car. This included the addition of a Lexan mounting plate over the entire chassis, which can be seen in the Modifications Section. As for basic mount of the Freedom Board to the Freescale Cup Car, see the video below, as it proposes the easiest solution with parts provided.

3. Line Scan Camera

Similar, for our project, the line following camera was not needed for anything, so little was done in the mounting of this component. The method seen below from the link above is the easiest way to mount the components with some of the parts provided, which seems to just use the servo attachment posts and some L-brackets. These brackets can be purchased from Parallax for cheap here or from Home Depot here. For reference, previous teams have needed to mount the camera significantly higher for viewing area, but this can be done by adding a post between the L-brackets and camera. The rest is up to you.

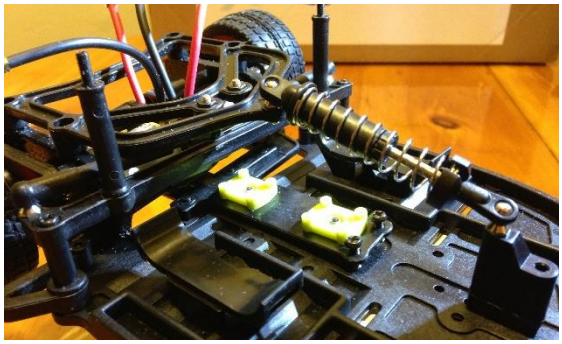


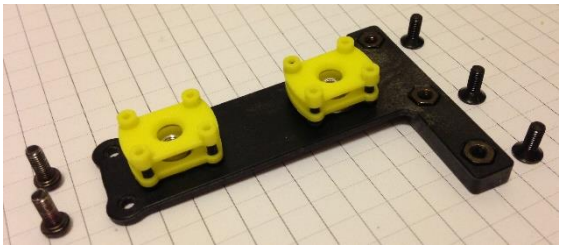
To attach the camera we found useful to prepare two metal L-shaped pieces made from aluminum. With the help of black plastic distance posts (already available in the kit) and these metal stands, you may freely change the position of the camera over the surface. See link above in header for CAD files.

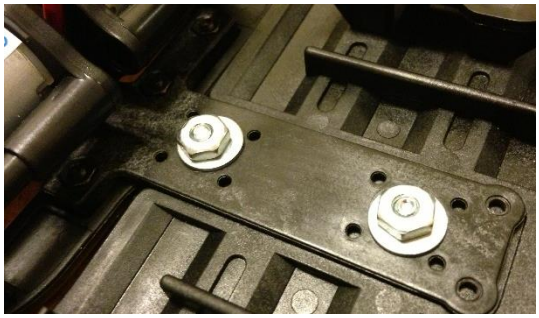
B. Modifications

1. Suspension Removal

The first design feature that was deemed useless for our project goals was the rotary joint suspension system implemented on the back of the car. This allows for three wheels to be on the ground when going over uneven terrain, but due to the ground clearance of the car this effect cannot be realized. For our purposes it makes more sense to have a rigid chassis to which the electronics are mounted, so these components were removed.







As can be seen, many parts were removed and the result gave considerably more open space on the chassis. The plastic T-bracket was secured using two #8-32 flat heat screws with nuts and washers to prevent movement between the two sections. This design is further reinforced by the adapter plate noted later, which connects the front and rear chassis sections through standoffs.

2. Servo Position Adjustment

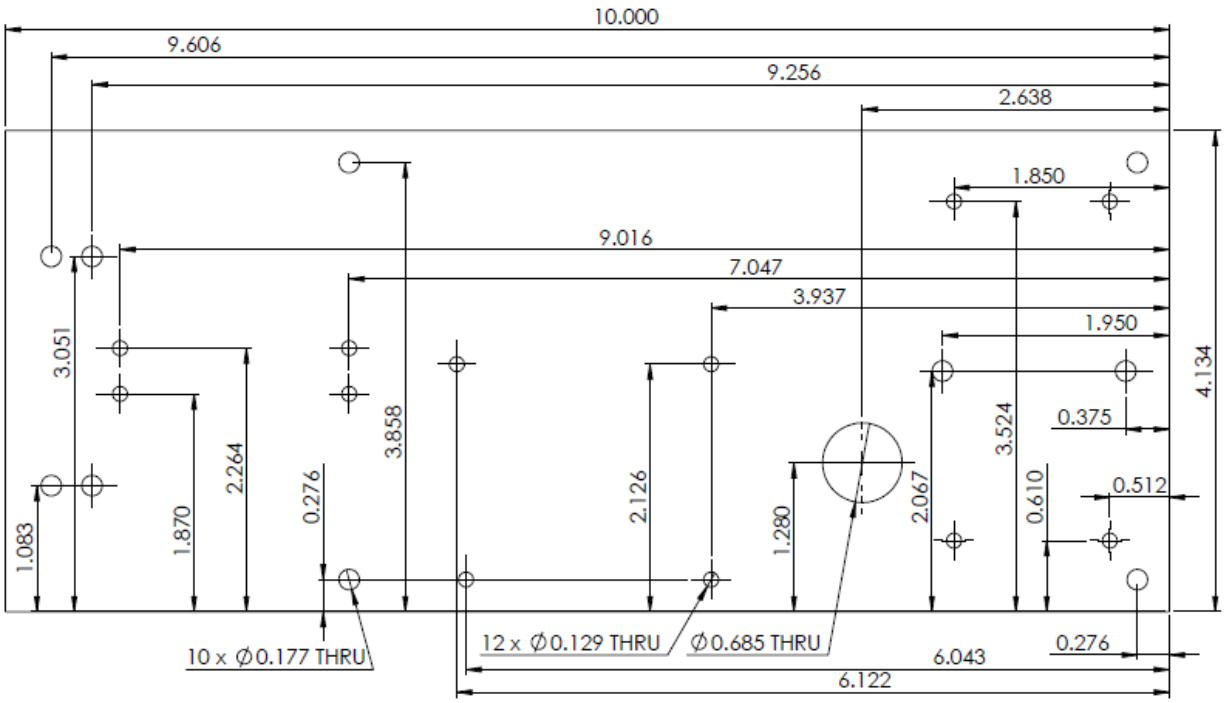
After some preliminary testing it was noted that more room would be available in the front of the car for bumper mounting and standoffs if the servo was reversed in orientation on the chassis. The provided plastic servo mounting blocks also became loose frequently and required excessive work to fix. Aluminum L-brackets were used along with left over screws and nuts from the kit

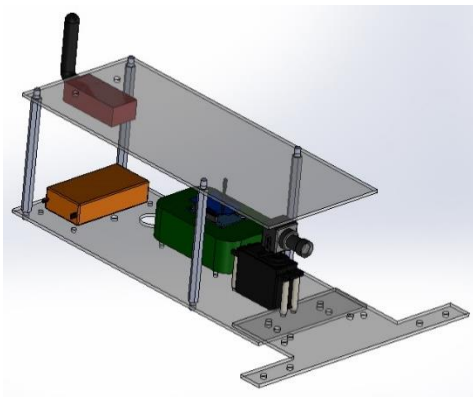
to mount the servo in a reverse orientation using two of the holes already available in the chassis. Two small plastic tabs had to me snipped off of the back of the servo screw mounts, and two more holes were drilled directly in front of the mounting holes for increased support. The new design allows for easier component removal and service and provides the same turning capabilities as the original mounting configuration.

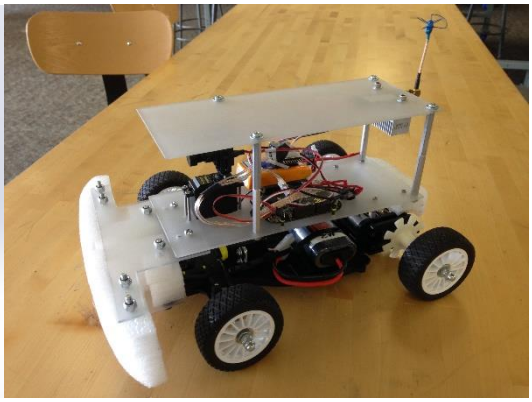
3. **Prototyping Plate**

The hole layout of the original chassis was not suitable for either the Freescale microcontroller or any of the other electronics we needed to add for this project. For this reason, we decided to make a prototyping plate out of Lexan. This plate attaches to the car at several points to increase rigidity, is elevated on stand-offs, and contain the hole pattern of all of the parts we are attaching to the car. In the diagram below, all dimensions are in inches for machining purposes. For the CAD model check out Fixed Adapter Plate. To download any CAD file, right click the "Display" link in the CAD directory and choose "Save link as…" This will save a .SLDPRT or .SLDASM file to the desired location.

Additional CAD work and component testing had to be done to optimize the hole layout for all of the components. This yielded the following CAD rendering, which shows where all of the components mount on the frame. Of course, this differs slightly from the final product, but it gives a good idea of the final objective of the design. For the CAD check out Fixed Car Components. To download any CAD file, right click the "Display" link in the CAD directory and choose "Save link as…" This will save a .SLDPRT or .SLDASM file to the desired location.

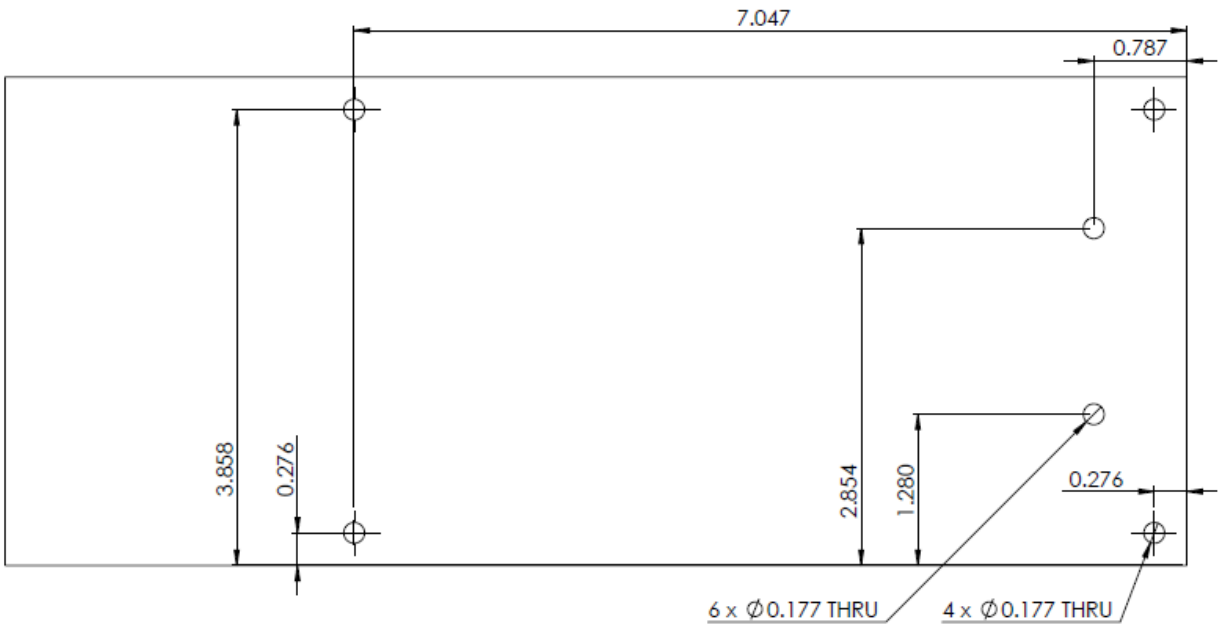






4. **Protective Plate**

To protect the electronics added to the car, an additional Lexan plate was added above the adapter plate and separated by standoffs. This design allows for some protection from outside forces while still allowing access to the electronics for modification. This standoffs are simple 3.5" long #8-32 aluminum hexes. A schematic of the plate to be fabricated is below, with dimensions again in inches for fabrication purposes. For the CAD model check out Protective Plate. To download any CAD file, right click the "Display" link in the CAD directory and choose "Save link as…" This will save a .SLDPRT or .SLDASM file to the desired location.



5. **Wheels**

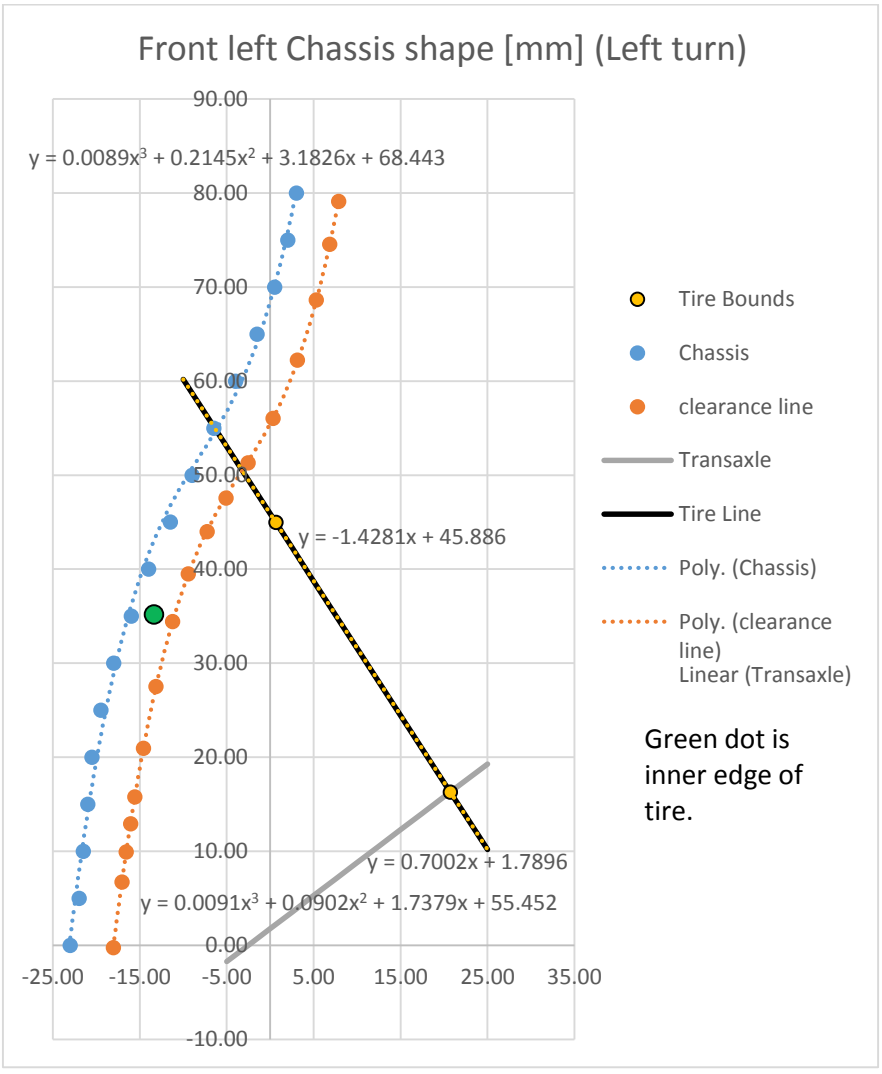
The original Freescale cup chassis wheels were 2 inches in diameter. This left only a few millimeters in terms of ground clearance. In addition, the locknuts attaching the front wheels stuck out such that the future goal of driving on the two side wheels would not work. In order to better accommodate this goal for future iterations of the project, softer, larger tires were deemed more useful. Fitting the car with monster truck tires would be out of the question due to the limited power of the small motors. For this reason, it was decided that the tires should be in the vicinity of 3 inches in diameter.

The new tires obtained are 2.8 inches (70mm) in diameter and much better suited (softer, bigger, and no pieces sticking out) to the task of providing a platform that could potentially be used for driving on two wheels in the future while not negatively affecting the usability of the car for normal driving purposes.

6. Front Transaxles

a) *Length Calculation*

Since the wheels were larger, the front wheels would now contact the chassis while turning if the original transaxle was kept. Some geometry was done in mapping out the shape of the chassis and determining the length of the new transaxle. A goal of 5mm clearance at a turning angle of 35 degrees for the inner wheel on a turn was assumed in the geometry as this was deemed to be the limiting case since the front bumper would not get in the way because it would need to be redesigned anyway. An excel file was set-up to iterate transaxle length given the parameters above and the radial distance inward from where the wheel rim met the transaxle and the inner edge of the tire which was 17 mm.

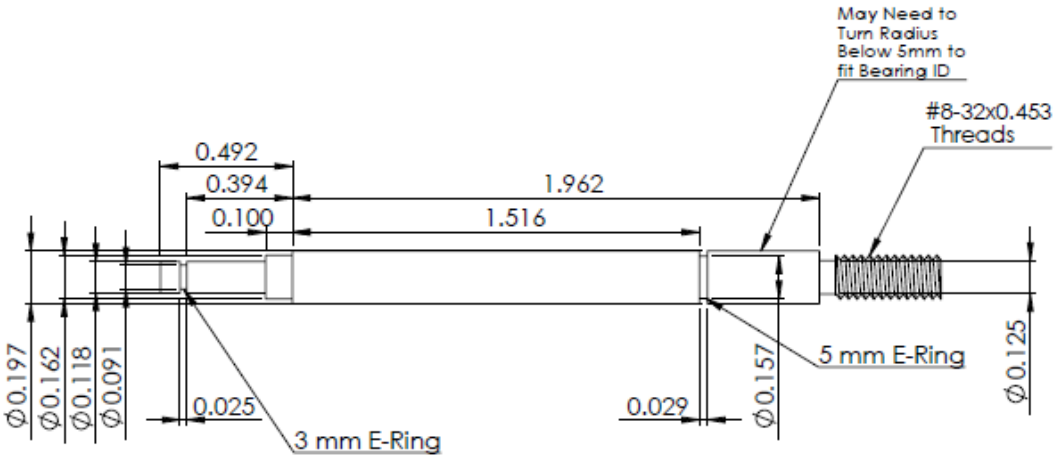


From iterating this map with new transaxle lengths, it was decided that the new transaxle length should be anywhere for 40 to 60mm. The old transaxle length was about 15mm.

b) *Practical Design of Front Transaxles*

The real transaxle would need to have additional length to stick out on either side. The inner side needed to clear where it was mounted on the kingpin so that a snap ring could be fitted to

hold it in. The outer side needed room for threads and a lock nut to hold the wheel in place from the outside.5mm centerless ground rod made from tool steel was used in the construction of the transaxles. Below is the technical drawing. As can be seen when comparing the drawing to the picture, a step-up was added to the drawing after spacers had to be made with the given dimensions to reduce movement in the connection point. This should be done in the initial machining for future parts. For the CAD model check out Front Axle Extension. To download any CAD file, right click the "Display" link in the CAD directory and choose "Save link as…" This will save a .SLDPRT or .SLDASM file to the desired location.



Dimensions in Inches

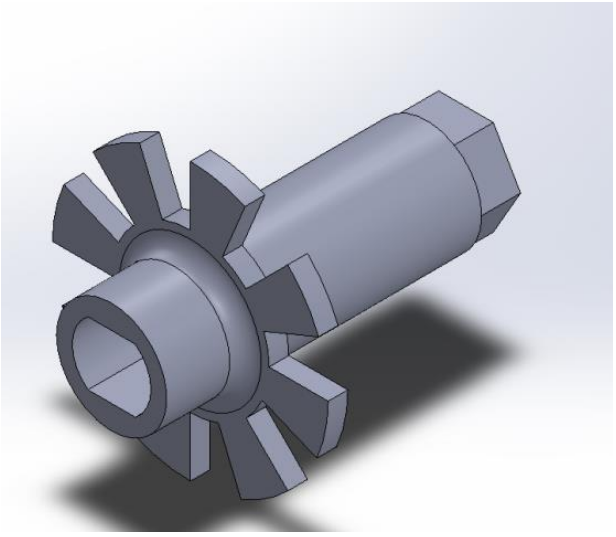
All dimensions are given in inches because that is how the lathes in the machine shop are set-up. All length dimensions are based off of the outside edge of the kingpin mount. The 3mm E-ring groove holds the transaxle from sliding outward. The outer opening of the kingpin mount is a 3.5mm hex and the inner opening is a 3mm hole, this hex in the previous transaxle prevented the transaxle from sliding inward. In the new transaxle, that hex is replaced by a step-up, and the step from 0.118" (3mm) to 0.162" and finally to 0.197" (5mm) prevents the transaxle from sliding inward. The 5mm E-ring slot was positioned to leave room for the washers, two bearings, and wheel rim, but leave just enough overhang (over the 0.125" diameter) so that the locknut could tighten up and eliminate slop in the system. The overall length of the transaxle is about 74mm; it is about 50mm in the length used in the calculation which falls right in the middle of the range.

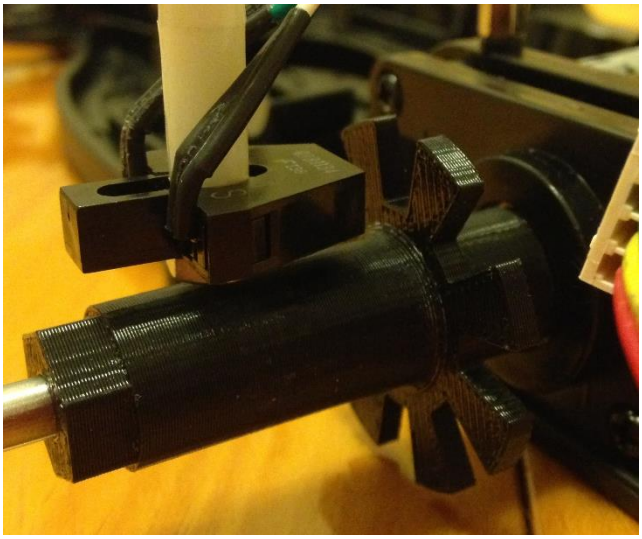
Due to the rod being about 5.03mm in diameter and the bearings being about 4.98mm in diameter, the sections of the transaxles outside of the 5mm E-ring groove had to be turned down slightly to allow the bearings to fit.

7. Rear Encoder / Wheel Adapters and Optical Gates

a) *First Iteration Encoder*

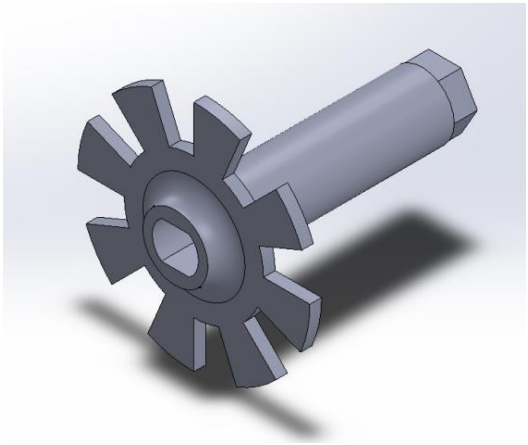
The new wheels came with new requirements to be mounted on the car. The old adapter for the wheels was a cylinder with two flats and a central hole for the alignment axle. The new wheels required a 12mm hex to transmit torque. So an adapter to go between these two shapes had to be designed. Since there was currently no good way to measure wheel speed on the original chassis, it was decided that these new adapters for the rear wheels would also act as encoder disks. The concept for out encoder design stems from a commercially available speed sensing encoder set that can be found here. The optical gates used for this project were donated and datasheets can be found here. These are reflective optical switches that are to be mounted to the chassis facing the encoder. A prototype of the encoder idea was 3D-printed before new front wheels were considered for testing purposes and design feedback. This design was later updated when the new wheels were added. For the CAD model check out Encoder Wheel Mount. To download any CAD file, right click the "Display" link in the CAD directory and choose "Save link as…" This will save a .SLDPRT or .SLDASM file to the desired location.



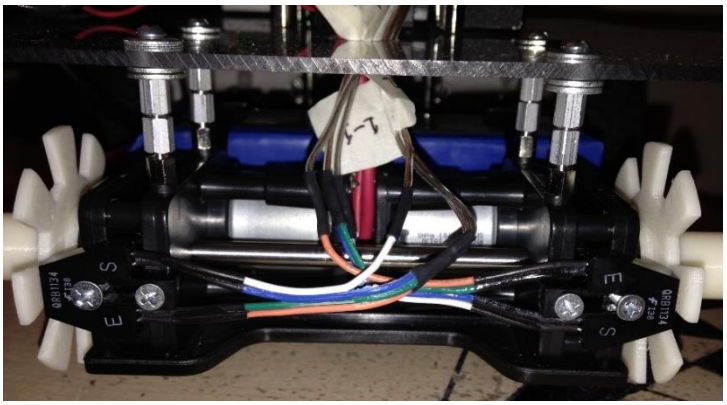


b) *Second Iteration*

In order to provide a platform for balancing on two wheels, the track width of the front and back wheels had to be the same. The new front transaxles and wheels put the new track width target at 8.4". The new encoders had to be designed to meet the same value for the rear track width. The position to mount the optical sensor also changed. So between iterations, the encoders got longer, larger in diameter, and the encoder teeth were moved closer to the wheel hubs to provide optimum spacing with the optical sensors mounted on the back. The color was also changed from black to white to obtain better values from the photo sensor. In addition the hex was widened by about 0.5mm to eliminate slop. The inner diameter of the encoders were drilled out with a #7 drill to 0.201" (5.1mm) to allow them to spin freely on the alignment rod. Depending on the material used in 3D printing, the interface between the encoder and hub may be tight and require minimal amounts of material to be removed to slide on easily. For the CAD model check out Encoder Wheel Mount rev2. To download any CAD file, right click the "Display" link in the CAD directory and choose "Save link as…" This will save a .SLDPRT or .SLDASM file to the desired location.

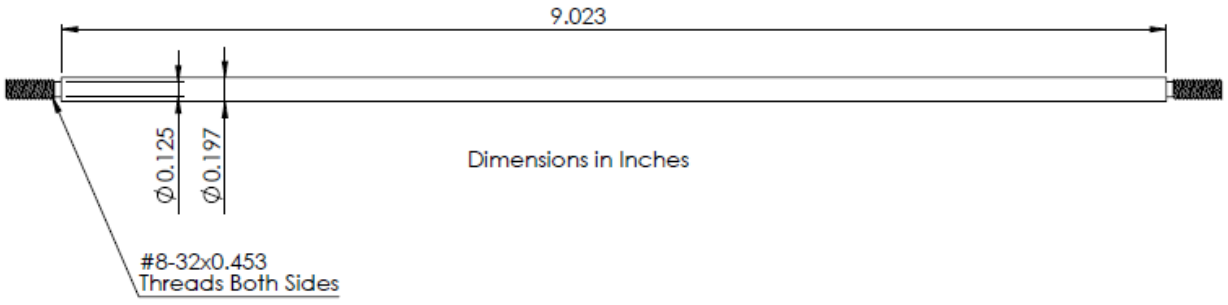


The new encoders were made larger so that the optical switches could read the rotational data while being mounted rigidly to the chassis with screws. The design also allowed for adjustability in the disk to switch distance, which was set at the optical 3mm gap. The connecting wires had to be replaced with more flexible connections to accommodate wiring constraints on the chassis. Be careful when drilling the screw holes into the side of the plastic where seen below. There is limited extra material in this location so size the mounting screws and tighten appropriately.



8. **Rear Axle / Alignment Rod**

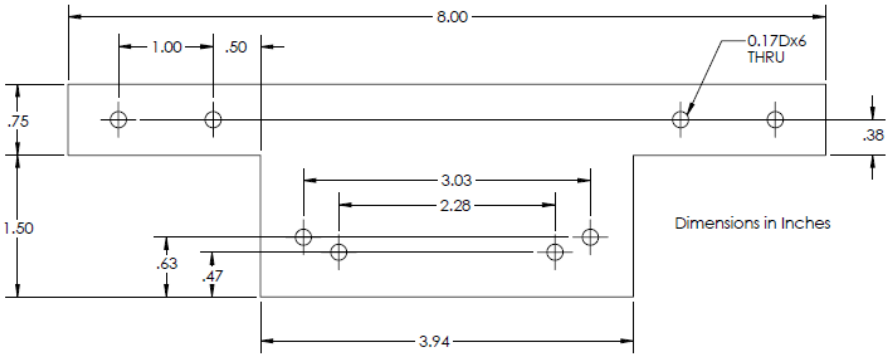
A new rear axle (which only serves to align the rear wheels and keep them from falling off) had to be designed to accommodate the new length of the encoders and to allow the locknuts on the threads to tighten up against the wheel to eliminate slop. For the CAD model check out Back Drive Rod. To download any CAD file, right click the "Display" link in the CAD directory and choose "Save link as…" This will save a .SLDPRT or .SLDASM file to the desired location.

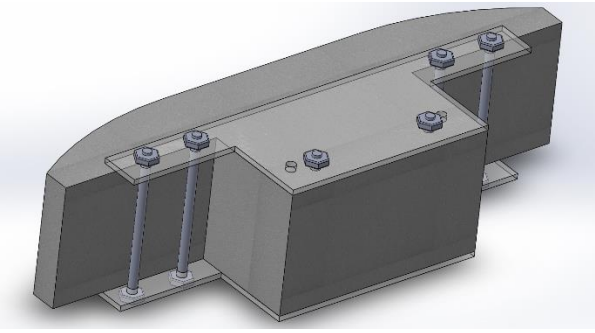


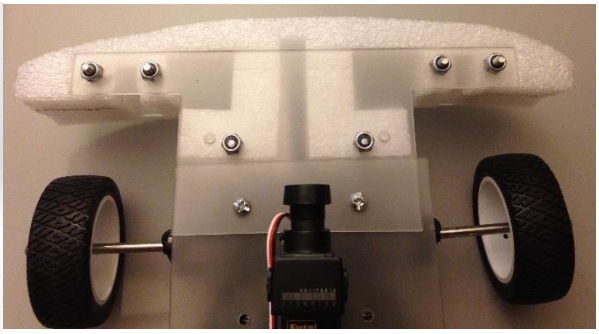
The rod is from the same piece of 5mm center ground rod made of tool steel. The inner diameter of the wheels were also drilled out with a #7 drill to 0.201" (5.1mm) to allow them to spin freely on the alignment rod.

9. **Bumper Modification**

The bumper design that would allow for the significant chassis weight increase and adequately protect the electronics went through many iterations, as quick analysis techniques do not really exist for impact testing. Instead, we went through some material and design testing before coming to an acceptable final product. The design consists of two Lexan plates that sandwich a piece of polyethylene foam and are held together with screws and nuts. Two holes for the original bumper had to be enlarged to mount the new bumper with #8-32 screws. The foam as free from senior design (it is suggested to find packing material rather than buying foam, which can get expensive) and was cut using a hot wire foam cutter to get the desired geometry. However, reasonable results can be achieved with a sharp knife. After the shape was cut, reliefs were cut into the back of the foam to allow for more compression. A back bumper was added by simply cutting an additional piece of foam and using the existing standoffs for the frame as securing points. The foam simply slides over the standoffs. For the CAD model check out Bumper Frame Bottom and Top. To download any CAD file, right click the "Display" link in the CAD directory and choose "Save link as…" This will save a .SLDPRT or .SLDASM file to the desired location.







C. **Car Steering Servo and Drive Motor Specs**

As measured on the console and confirmed online, the range of the steering wheel is +/- 120 degrees. The steering angles of the car wheels are targeted at +/- 35 degrees to simulate normal car handling. These values yield a steering ratio of 3.4.

D. **Battery and Power Management Specs**

The Freescale Cup Car is powered by a 3000 mAh NiMH battery that is charged with the supplied Tenergy brick charger. The status LED will be red while charging and green when charged. The current level selector switch can be in either setting: 2A will charge more quickly, 1A will prolong the life of the battery but will take longer to charge. This is a very robust battery that outputs 7.2 V nominal and 8.2 V at max charge, and they can be fully discharged without issue with damaging the battery. These batteries do not have a quick charge time (2 - 4 hours), so two batteries are supplied with this car so that one can be charged while the other is being used.

The camera and video transmitter on the car are powered by a separate 3S 35C 11.1V LiPO battery pack. This battery pack is estimated to have a run time of two hours, and we have an additional battery pack for continuous use of the system. The batteries are charged with the silver Tenergy balance charger (simply plug the white connector into the four pin connector port). Make sure the switch is set to “LiPO,” and the 1-4cells Li-Po/LiFe LED will be red while charging and green when fully charged.

II. Physics Theory
Torque Vectoring for Cornering Performance

A. Goal

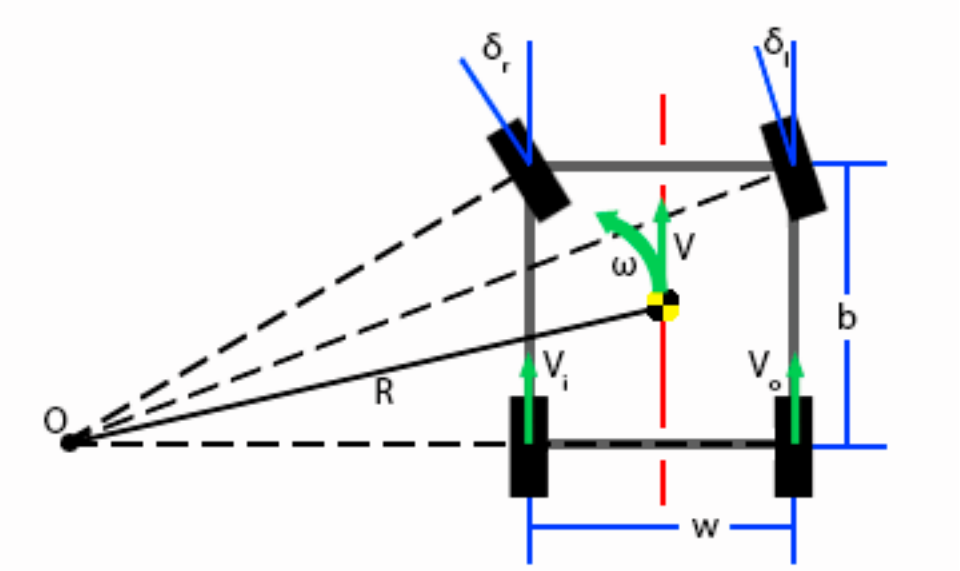
Define a desired wheel speed input signal to serve as a reference signal in the tangential speed control loop for the inner and outer wheels on a turn. Use these reference speeds to change the left and right wheel speeds to alleviate wheel slip through turns.

B. Assumptions

- No slip
- Car moves as rigid body through turn

C. Inputs

- Tangential (forward) speed of car (generated from throttle input).
- Steering angle (angle of front wheels)



D. Notation

1. Variables

- *R*, radius [m]
- *V*, tangential velocity [m/s]
- *w*, track width of vehicle [m]
- *b*, wheel base [m]
- δ, steering angle of front wheels [deg]
- ω, angular speed through turn [rad/s]

2. Subscripts

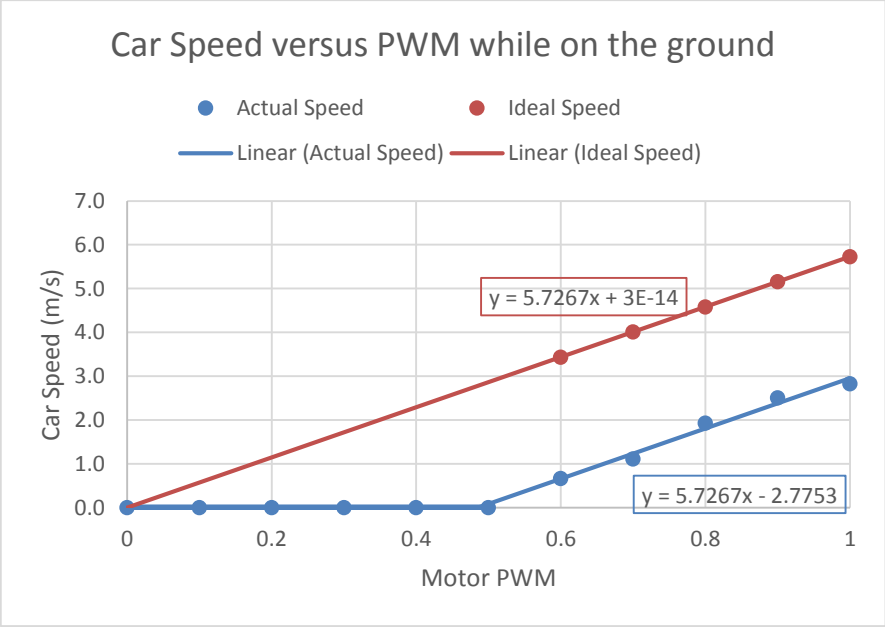
- *r*, right side of car
- *l*, left side of car
- *i*, inner side of car on a turn
- *o*, outer side of car on a turn
- No subscript means the variable applies to the center of the car

E. Vehicle Parameters

Parameter	Symbol	Value	Units
Wheel Base	b	0.1981	m
Track Width	w	0.1397	m
Tire Radius	r	0.0356	m
Time Constant	τ	1.4	s
Total Weight	Fg	1670	g
Steering Angle	δ	-35 < δ < +35	deg
Tangential Velocity	V	0 < V < 6 (ideal)	m/s
Dead Zone	DZ	+/- 2.78 (0.5 PWM)	m/s
Velocity Saturation - DZ	V_{sat}	+/- 1.52 (0.75 PWM)	m/s
Torque Saturation - DZ	T_{sat}	+/- 0.7 (0.75 PWM)	m/s^2

1. Saturation and Dead Zone values

Note that the Dead Zone value, *DZ*, is reported by what speed the 0.5 Pulse Width Modulation (PWM) corresponds to ideally as that is the PWM required to get the car moving while it is on the ground. The Saturation values, V_{sat} and T_{sat}, reported in terms of Saturation minus the Dead Zone for convenient use in Simulink since the control loops in the Simulink model are in terms of m/s, the placement of the saturation blocks immediately after the compensators requires that dead zone already be taken into account. The value of T_{sat} was obtained from the graph in the System parameters of the Controls Application Section. The method used to determine T_{sat} can be found in the Non-linearities section. Also note that the motor saturation is arbitrarily capped at 0.75 PWM instead of 1.0 PWM. This is done as 1.0 PWM is just too noisy to get useful encoder data. The arbitrary PWM cap affects the value of the V_{sat} and T_{sat}, and if that value changes from 0.75 PWM, then V_{sat} and T_{sat} must be re-calculated. All values in this table appear as they are used in the Simulink model, no addition conversions are needed. Use the chart below to convert from PWM to speed if necessary:



The data was obtained by passing through a timing gate 3 times at each PWM averaging the results and drawing a trend line. The ideal speed line is drawn with the same slope as the Actual Speed line, but with no Dead Zone, use this to determine what the input speed should be in Simulink.

F. **Method**

The average steering angle δ corresponds theoretically to the average of the left and right Ackerman steering angles:

$$\delta = \frac{\delta_r + \delta_l}{2} \tag{1}$$

However due to slop in the steering linkages, δ is assumed to be a linear function of the maximum turning angle of the wheels and the console steering wheel itself. The angle of the console steering wheel θ is known from the encoder in the steering wheel. Both δ and θ are in degrees. The value of δ varies from -35 degrees (left turn) to +35 degrees (right turn), while θ varies from -120 degrees to +120 degrees.

$$\delta = \theta(\frac{70}{240}) \tag{2}$$

Use steering angle, δ, to determine turn the radius, *R*. The no slip assumption implies that this is a low speed turn which assumes that lateral forces are negligible, and the turn radius is independent of the car's speed.

$$R\ [m] = \frac{(b\ [m])}{\tan(\delta\ [\mathrm{deg}])} \tag{3}$$

*Note that if the steering angle becomes zero, *R* will tend toward infinity. In order to avoid this the code simply contains an "if" statement that says if $|\delta| < 0.001°$, then $\delta = 0.001°$. This will result in a turning radius so large, that the wheels will spin at the same speed, but the code will believe the car is turning right.

Use the known track width, w. in meters to determine the radius of the turn as seen by the inner, R_i, and outer rear, R_o, wheels in meters:

$$R_o[m] = R + \frac{w}{2} \tag{4}$$

$$R_i[m] = R - \frac{w}{2} \tag{5}$$

Use forward speed input and central turn radius to get the angular speed of the car through the turn:

$$\omega\ [\frac{rad}{s}] = \frac{V\ [\frac{m}{s}]}{R\ [m]} \tag{6}$$

Use the rigid body assumption (all parts of car have the same angular speed, ω, through the turn at a given time) to map the inner, R_i, and outer, R_o, radii to an inner and outer tangential speed:

$$V_o[\frac{m}{s}] = (R_o[m])\left(\omega[\frac{rad}{s}]\right) \quad (7)$$

$$V_i[\frac{m}{s}] = (R_i[m])\left(\omega[\frac{rad}{s}]\right) \quad (8)$$

These tangential wheel speeds are what the PID loops use as the set-points. Equations (3)-(8) are useful for debugging purposes as the intermediate values have easy to grasp physical meanings, however equations (3)-(8) can be reduced to:

$$V_o\left[\frac{m}{s}\right] = V[\frac{m}{s}](1 + \left(\frac{w}{2b}\right)tan(|\delta|)) \quad (9)$$

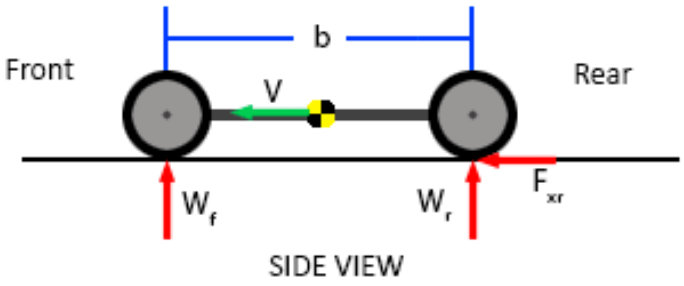
$$V_i\left[\frac{m}{s}\right] = V[\frac{m}{s}](1 - \left(\frac{w}{2b}\right)tan(|\delta|)) \quad (10)$$

Where V [m/s] is an input to the system, and represents the central tangential speed of the vehicle, in the microcontroller code, V comes from the throttle input. When implementing the control system on the KL25Z, equations (9) and (10) were used to reduce calculation time and complexity instead of equations (3)-(8).

G. **Notes**

Equation (3) assumes an average steering angle, δ, for the front wheels. This angle would be the average of the inner and outer wheel steering angles because they differ slightly due to the Ackerman principle, however for us, δ comes from Eq. (2). In addition, using a bicycle model with a high speed turn would vastly complicate Eq. (3) to:

$$R = 57.3\left(\frac{b}{\delta} + \left[\left(\frac{W_f}{C_{\alpha f}} - \frac{W_r}{C_{\alpha r}}\right) + \left(\frac{W_r}{C_{\alpha r}}\frac{F_{xr}}{C_{\alpha r}}\right)\right]\frac{V^2}{g\delta}\right) \quad (11)$$



Where δ [deg] is the average steering angle, b [m] is the distance between the front and rear axles, W_f [N] is the weight felt by the front wheels, W_r [N] is the weight felt by the rear wheels, V [m/s] is the tangential velocity, $C_{\alpha f}$ [N/deg] is the cornering stiffness for the front wheels, $C_{\alpha r}$ [N/deg] is the cornering stiffness for the rear wheels, and F_{xr} [N]is the tractive (driving) force coming from the rear wheels, g [m/s^2] is the acceleration due to gravity, and 57.3 is the conversion factor used if δ is in degrees.

Radial slip is currently unaccounted for due to the no slip assumption. The differential drive feedback will attempt to eliminate tangential slip, but will not affect radial slip. For this reason, the radial slip may be estimated possibly by obtaining the centripetal force the car is experiencing, and comparing it to the largest amount of centripetal force the tires can provide on the given surface. (This would require estimating the coefficient of friction as well as measuring radial speed or acceleration).

III. Controls Application

A. Torque Vectoring Model

Theory of Operation

The goal of this torque vectoring control system is to improve the turning characteristics of the car by accounting for the difference in speed required from the inner and outer drive wheels during a turn. This system will use closed loop feedback to determine the speeds at which each drive wheel should spin during a given turn.

B. System Inputs and Outputs

The required system input signals are:

- The wheel speeds of the rear wheels from the encoders (pulse widths of time measured over the width of an encoder tooth converted later to [m/s]).
- The steering angle of the steering wheel θ [deg] (later converted to δ by Eq. (2).

The required system parameters are:

- The track width of the wheels (left/right)
- The wheel base (forward/back)
- Wheel diameter
- First order approximation time constant

The desired outputs of the system are:

- Left and right wheel speed

C. Signal Processing

The only remaining calculation before the PID loop can work is to convert the left and right encoder data into a tangential speed so that the signal makes sense for debugging purposes. This is done by measuring the distance from the center of the wheel at which the optical switch is located and combing this with the knowledge that there are going to be 8 cycles of tooth/no tooth to calculate the arc length subtended during a measured pulse width, T_s [ms]. When converted to yield [m/s] the results are:

$$V_{l,feedback\,[\frac{m}{s}]} = \frac{16.83}{T_{s,l}[ms]} \tag{12}$$

$$V_{r,feedback\,[\frac{m}{s}]} = \frac{16.83}{T_{s,r}[ms]} \tag{13}$$

D. PI Control

Closed loop feedback will be used to determine the output inner and outer wheel speeds. These loops require a reference signal which is based on the average velocity, *V* and instantaneous turn radius, *R* which is based on the steering angle (see Physics Theory Section). The closed loop feedback portions of the control system are to be tuned using PI control directly on the KL25Z (freedom board) as shown below:

$$CO = CO_{bias} + K_c\left(e(t) + \frac{1}{T_i}\int e(t)dt + T_d\frac{dPV}{dt}\right) \tag{14}$$

Where *CO* is the controller output (*V*), *e(t)* is the error at time t (setpoint *V*- encoder *V*), *PV* is the process variable (encoder *V*), and *Kc*, *Ti*, and *Td* are the PID proportional, integral, and

derivative coefficients. The PID coefficients can be modified by changing *Kc*, *Ti*, and *Td* in the code.

E. Discretization of PID in MBED

The continuous Eq. (14) used by MBED must be discretized for use on the freedom board. First, in MBED, the parameters given are *Kc*, *Ti*, *Td*, and *Rate*. *Rate* is the variable which describes the discrete time-step to be used in seconds. The discretization method is as follows:

$$Integral = e(t) + e(t-1) \quad (15)$$

$$Derivative = \frac{Z-1}{(Z)(T_S)} \quad (16)$$

Where *e(t)* is the error between the set point and the process variable at sample time (*t*). The integral contains an "if" statement which only integrates if the input is not pegged at a limit. This is to prevent reset windup. Note that the derivative control is unfiltered.

F. Important Note

Wheel Speed Output

The PI control loops operate on the reference velocity for left and right wheel speeds, not inner and outer wheel speeds. The reference inner and outer velocities are generated from throttle input and steering input. The inner and outer reference velocities are then assigned to left and right prior to entering the control loops. This is based on steering angle, for instance if the steering angle is 20 degrees to the left, the right wheel reference will be assigned the outer wheel speed. This will change when the steering angle crosses zero.

G. Simulink Model

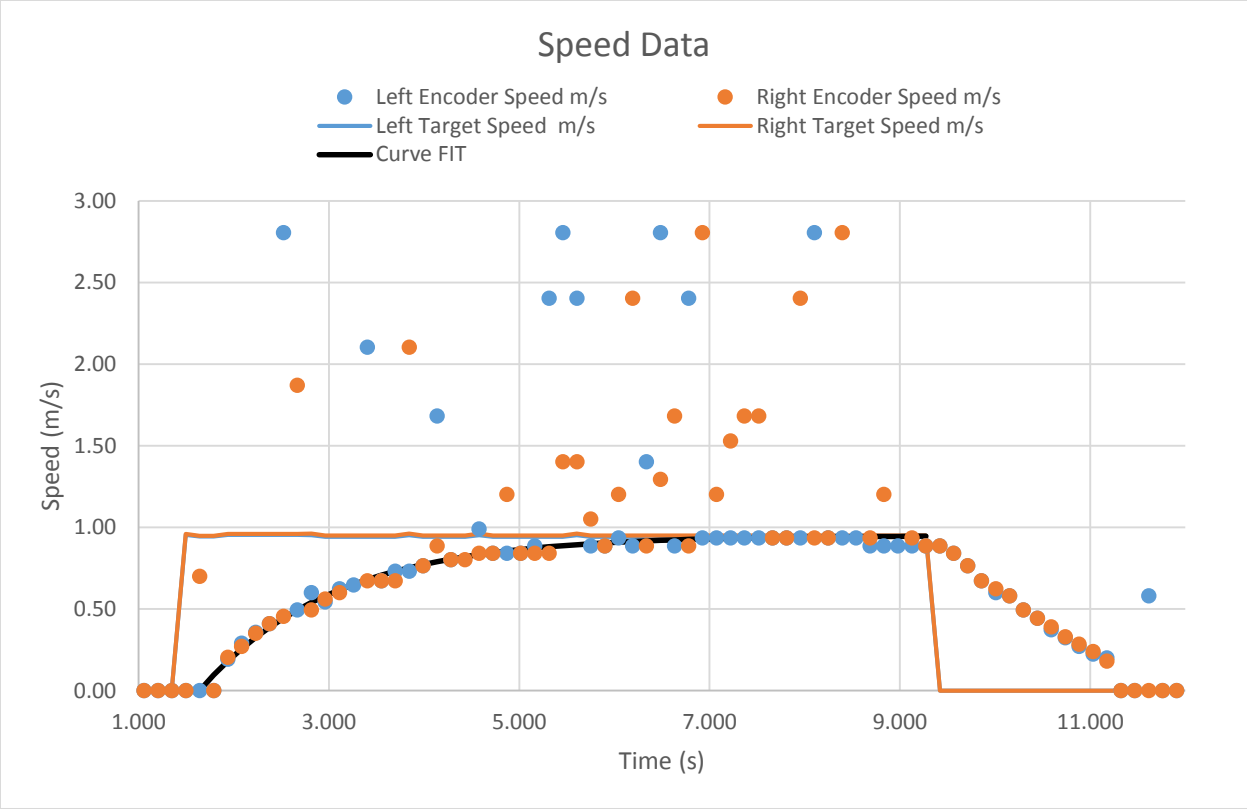
1. System parameters

See Section E of the Physics Theory Section for a table of parameters used.

The motors are modeled using a first order approximation with a single time constant. Using velocity versus time data for an acceleration from rest to maximum speed, an exponential curve was fit to the data. The curve fit equation is:

$$Max\ Speed(1 - exp(-\frac{t-t_0}{\tau}) \quad (17)$$

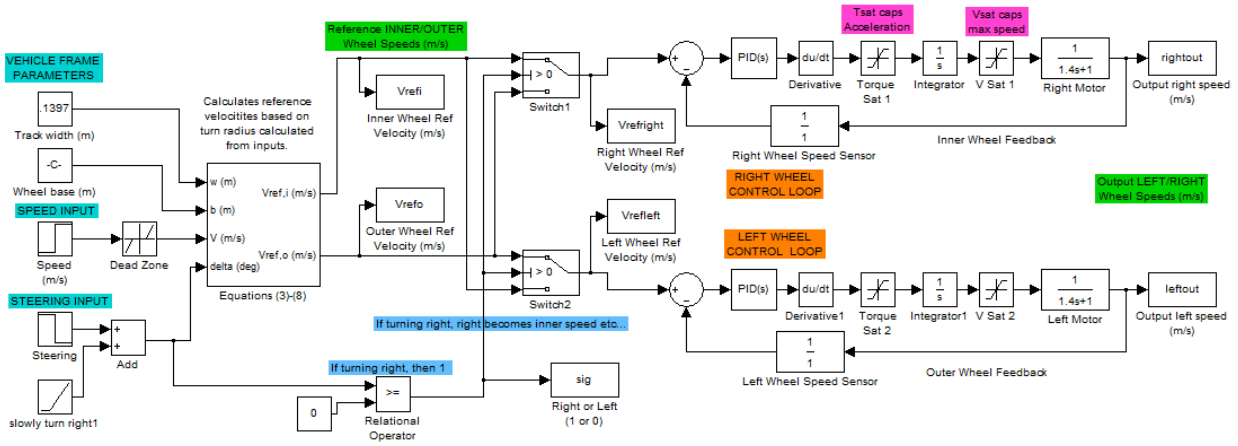
For a total system weight of 1670 grams (only affects time constant) and a max speed of 0.95 m/s, the time constant was fit to be 1.4s.



The above data is unfiltered and PI control was off. The curve fitting was done by manually iterating the time constant as Excel's least squares approach was biased toward the earlier parts of the curve due to the higher slope and greater data point density as the car was moving slower. In addition, the noise would also throw off the least squares fit. Only the left wheel was fit, since it was not exact and the data from each wheel was very close.

2. Control Loops

The Simulink model is continuous in time and relies on signal generators for steering angle, δ in degrees, and left and right input wheel speeds in m/s. The user can make these whatever he/she wishes as long as $0 < V < 6$, and $-35 < \delta < 35$. The system parameters are already in the Simulink model, and once the input is given, it goes through Eq. (3)-(8). The feedback loops then operate on the reference speeds for the left, V_l, and right, V_r, wheels. The control loops are unity feedback with a PID compensator, a torque saturation T_{sat} (for acceleration), a max speed saturation V_{sat}, and a 1st order approximation of the motors as seen below.



The PID coefficients in Simulink can be changed by double clicking on the PID blocks and adjusting *P*, *I*, and *D*. The PID model implemented by Simulink is as follows:

$$CO = e(t)\left(P + I\int\frac{1}{s} + D\frac{N}{1+N\frac{1}{s}}\right) \tag{18}$$

Where *CO* is controller output. Note that these values differ from Eq. (14) so any tuning done in Simulink will need to be converted to the format in Eq. (14) for use on the car. Also note that *N* is a coefficient used to filter the derivative control.

3. Non-linearities

Non-linearities are taken into account in the Simulink model as well. The speed input is subject to the dead zone block before it enters the sub-system containing equations (3)-(8). In addition, the outputs of the compensators are subject to the saturation blocks. Note that the control loops work in [m/s] this corresponds to a certain rotational speed of the motors, which corresponds to a certain voltage, but the *DZ* and V_{sat} values must be given in [m/s] which means some experimenting must be done with motor pulse width modulation (PWM) values if each group is to determine values for itself. Our version of this relationship is located in the vehicle parameters section. The dead zone block in Simulink simply subtracts the *DZ* value for any input higher than *DZ* (for positive inputs) or relays a zero for signal less than *DZ* (for positive inputs). The velocity saturation blocks clip the velocity signal at the maximum/minimum values given by V_{sat}, and the torque saturation blocks clip the acceleration signal at the maximum/ minimum values given by T_{sat}.

The values for V_{sat} [m/s] were obtained from the encoders as they simply correspond to the car's maximum speed (at maximum PWM). The values for T_{sat} [m/s^2] were obtained from recording the velocity versus time data using the current arbitrary PWM cap in the car code. The first few points were fit with a line to get obtain the maximum acceleration value T_{sat} [m/s^2]. The values for *DZ* were obtained while the car's wheels were on the ground by feeding increasing PWM values to the motors until the car began to move. The value of *DZ* in m/s is obtained from multiplying the motor's fractional PWM value at the edge of the dead zone by the maximum speed of the car in m/s.

4. **Logic**

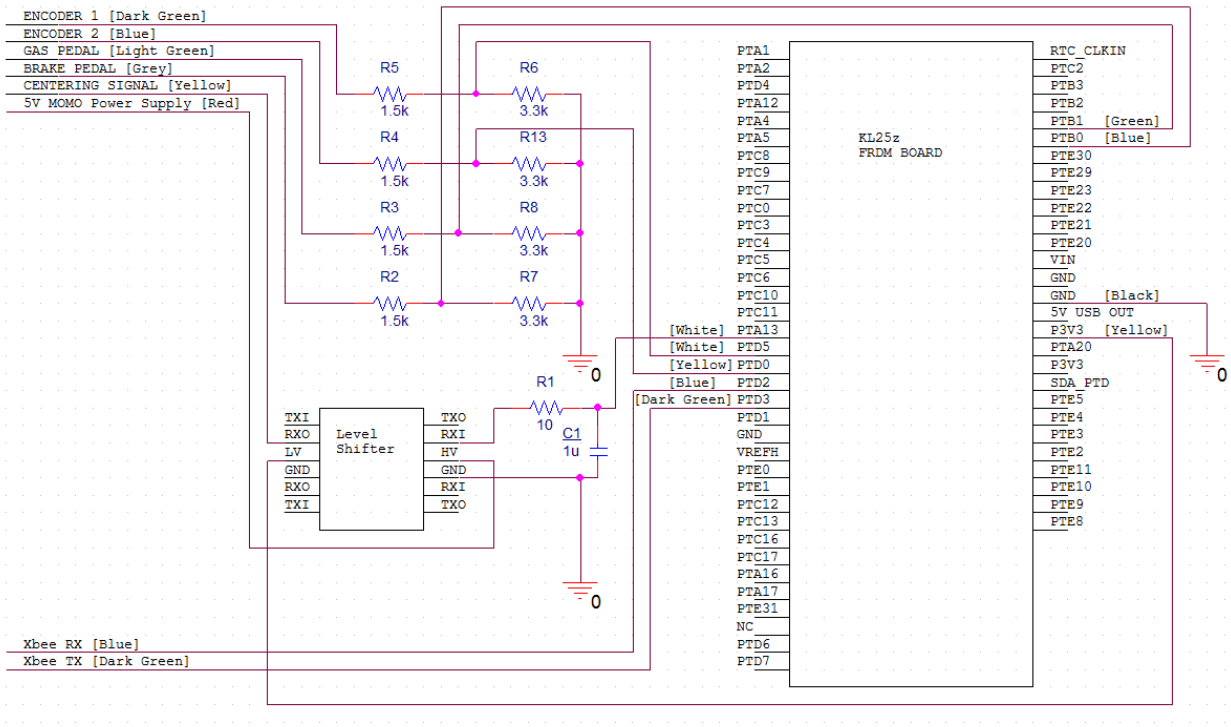
Lastly, the Simulink model must decide which reference velocity to send to the left/right wheels. This is accomplished by using a variable called "*sig*" which equals 1 when $\delta > 0$, and equals 0 when $\delta < 0$. So *sig* is 1 when the vehicle is turning right. *Sig* is defaulted to read to the right due when $\delta = 0$ to avoid the turn radius going to infinity. *Sig* is then used as the switch criterion for the two switches required before the control loops to determine whether left or right is inner or outer. On a right turn, right will be the inner wheel.

IV. Console Electrical Information

The driving console (Logitech MOMO steering wheel) features a KL25Z microcontroller. More information about this microcontroller can be found at the MBED handbook page. The wheel is powered with the stock USB connector, and the signals from the different internal components are conditioned and fed into the microcontroller.

A. Electronics Modifications

1. Schematic



Above is a schematic that represents the connections made in the MOMO console. These connections are made to enable the sensors in the MOMO console to communicate safely with the FRDM Board. The FRDM operates at 3.3 V for most of its input signals, while the MOMO console's sensors operate at 5V. To protect the FRDM board, the above circuitry has been designed. For the Encoder 1 & 2, Gas Pedal and Brake Pedal, a resistor network was created. The Wheel Centering Signal requires a level shifter. A level shifter is a series of MOSFETS and resistors designed to step up or down a voltage to the desired range. In this case, the level shifter is only being used in one direction, and it's to step down the centering signal voltage to 3.3 V.

The level shifter requires two different voltage supplies to operate. The HV (High Voltage) pin is connected to the power supply from the MOMO console. This was previously attached to FRDM board's power supply, but it was found that the FRDM did not provide enough current to make the output signal from the level shifter reach 3.3 V. The LV (Low Voltage) pin on the level shifter is attached to the FRDM board's 3.3 V output. The output pin, labeled RXI, has a low pass filter attached between it and the input pin on the FRDM Board. This is because the signal was noisy when first measured.

2. **Wiring Harness Pinout**

Inputs	Wire Color	Outputs	Wire Color
5V	Red	Xbee TX	Dark Green
Steering Encoder 1	Dark Green	Xbee RX	Blue
Steering Encoder 2	Blue	Steering Encoder 1	White
Gas	Light Green	Steering Encoder 2	Yellow
Brake	Grey	Gas	Green
3.3 V	Yellow	Brake	Blue
GND	Black	Centering Out	White

3. **Signal Conditioning**

There are many different kinds of signal conditioning, and, depending upon what the project requires, only a few will be needed for any project. The most common is filtering, which is useful for isolating desired signals and reducing noise. The MOMO console does this often throughout its system, so the only filtering needed was for the centering signal, which was just to remove a little noise. In this case, a low-pass filter with a time constant of 0.1 was used.

The other kind of signal conditioning used for the MOMO console was a voltage level shifter. Level shifters are most commonly used when trying to communicate between two different systems that operate at two different voltages. The most common example of this is communicating with almost any microcontroller and serial RS-232 logic on a computer. RS-232 operates at 3.3 VDC, while most micro controllers operate at 5 VDC, similarly to USB.

The level shifter used in this project was the BOB - 11978, from sparkfun.com. This level shifter shifts between the two voltages 5 VDC and 3.3 VDC, because it is meant for RS-232 communications. The following pins were assigned the following values:

Pin	Value
HV	5 VDC
LV	3.3 VDC
GND	GND (0 VDC)
TXO	Centering Signal Input
TXI	Centering Signal Output

B. **Console Components**

The Logitech MOMO steering wheel, donated to the team, was analyzed for use without using the pre-existing and historic drivers for it. The original driver which existed can only be run on a

windows machine and still had numerous errors and bugs. Also, the readings from the controller could only be seen by using the GUI from Logitech.

Due to the lack of usability of the pre-existing software it was decided that the analogue signals from the internals be taken in, calibrated and then used directly. Ideas of creating a self-programmed driver were also brainstormed and it was decided that for obtaining only steering data it was too exhaustive of an effort to design and program a driver for it. Utilizing the signals straight from the internal circuitry also enabled the use of a low powered microprocessor.
It must be given note that there were two sources of power to the steering controller, the 5V from the USB and the 24V from the wall outlet. The 24V supply was only connected to the motor. When the 5V power supply was not connected all the signals read noise when the steering was moved. This was postulated to be due to the back EMF from the motor. Determining that only the 5V supply is needed was a major discovery, and only this USB is used to power the steering wheel electronics.

The concept for steering requires the following known quantities:

Turn Direction – Obtained from the quadrature encoder signal.
Turning Displacement – Obtained from the encoder PWM which will be calibrated to encompass the whole steer angle with PWM count. This will require the wheel to be centered.
Centering Location – The steering wheel center position is determined from the optical centering sensor present on the wheel axle.

1. **Quadrature Steering Encoder**

Quadrature encoders are a type of encoder used to determine angular velocity and direction. Normal encoders cannot tell direction on their own. If one were to start turning the encoder in the opposite direction, the encoder would just keep counting tics the same as if the direction hadn't changed.

The way quadrature encoders solve this problem is by using two encoders that are slightly offset by each other. The encoder plate needs to have slits in it that are big enough for both encoders to see though it at the same time, and have the space between each gap big enough to block both encoder signals when positioned over them. What the encoders generate are two PWM signals that are offset by 90 degrees from each other.

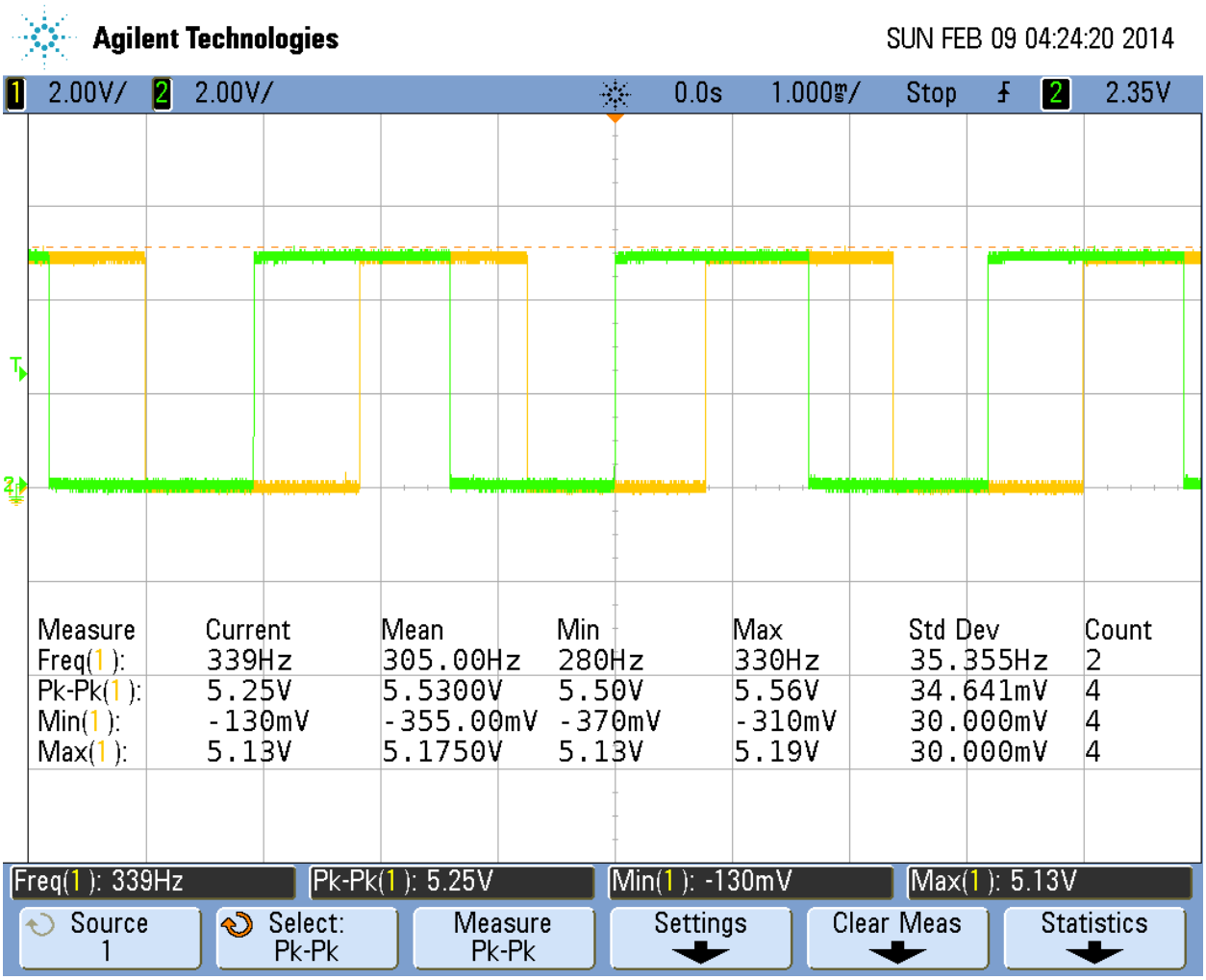
This results in a binary counter that determines the direction of movement based on the previous number that was received.

When the encoders are both seeing the gap in the encoder, their signals will be 00. When one starts to get covered up, but the other is still uncovered, the signals will read 10. When both encoders are covered, the signal will read 11. When the second encoder is covered and the first encoder is uncovered, the signal will read 01. These numbers can be converted into decimal, and, based on which number the encoders were reading previously, will determine if the encoders have changed direction.

To better understand this, the following tables show the two possible rotations and all possible encoder positions.

CW Rotation			CCW Rotation		
	Encoders			Encoders	
Phase	A	B	**Phase**	A	B
1	0	0	**1**	1	0
2	0	1	**2**	1	1
3	1	1	**3**	0	1
4	1	0	**4**	0	0

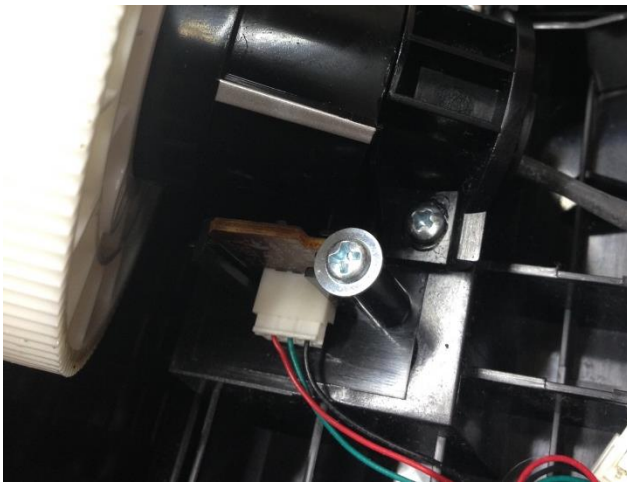
Below is an oscilloscope screen capture of the voltage levels of the two encoder signals reacting to movement can be seen.



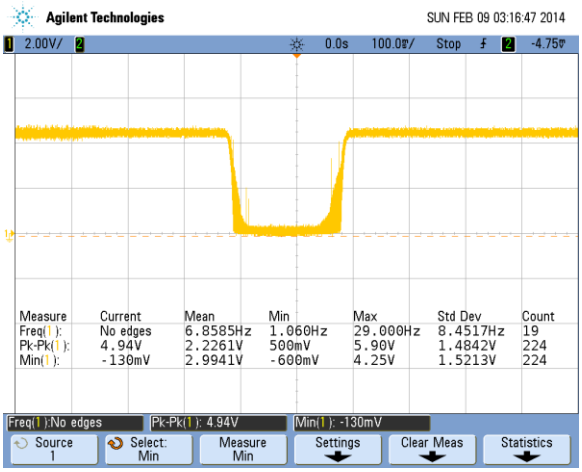
Encoder PWM

2. **Optical Gate**

The centering on the wheel is done by utilizing the signal from the optical encoder attached at the axle of the wheel. There is a white reflective strip present which triggers the optical sensor whenever it passes by the receiver-transmitter to detect that the wheel has been centered. Originally, with the windows software installed, the steering wheel would go into a centering reset routine where when the white strip would pass over the optical sensor and the motor would stop rotating and record that as the center position.



Steering Wheel Centering Reflective Strip



Steering Centering Optical Sensor Signal

Since the windows software was not being used, raw signals from the pins of the encoder were to be recorded. Whenever the white strip (as seen in Figure 3) passed over the optical sensor there was a sharp drop in voltage detected. This drop can be used to determine that the center position has been achieved and the encoder count set to zero at this point. With the center determined, the direction pulled from the diodes and the encoder PWM count reset, all the necessary signals for effective and accurate steering are now set and can be utilized.

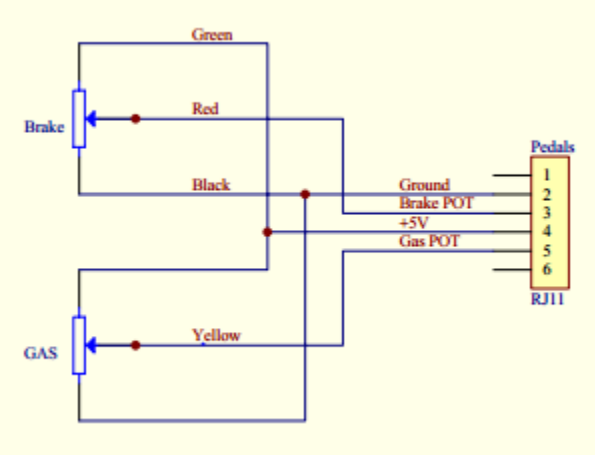
All signals have a certain chance of error. When a computer is counting the encoder tics, and an error occurs, the computer has no way to know that it is in error, and therefore, will keep it in the encoder count. This results in a long-term offset between where the computer thinks the steering wheel is, and where the steering wheel actually is.

With a fine-toothed encoder, these errors are negligible. But, as with all errors, they are undesirable and can add up to significant problems further down the road. The constant resetting of the encoder count by the centering signal removes this error as a problem. Now, the only error that can occur has to occur within the time it takes for the user to turn the wheel from the center, to one side as far as possible, and back to the center. The error becomes virtually nonexistent because of this narrow window in which it has to occur.

The centering signal, similarly to all the signals on the MOMO console, operates between 0 - 5 VDC. This is an industry standard, however, the FRDM Board requires that all signals operate between 0 - 3.3 VDC, which means this signal needed to be conditioned.

3. **Pedal Potentiometers**

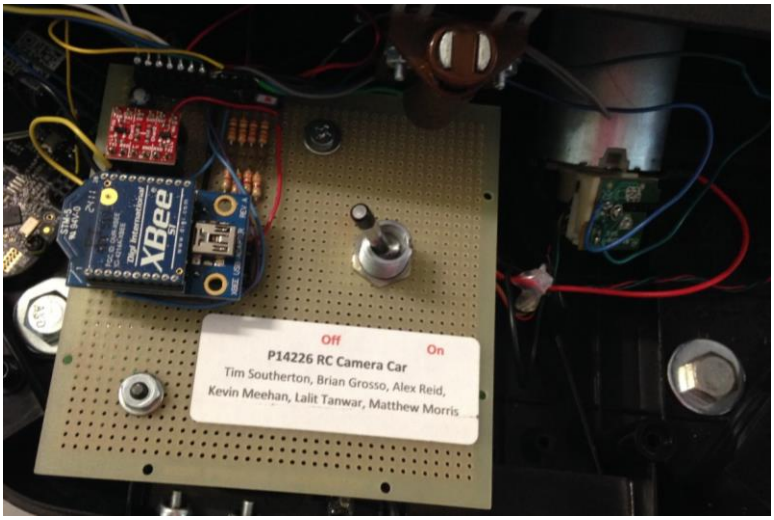
The MOMO pedals are designed to replicate the accelerator and the brake pedals of an actual car. The signals are sent as voltage values. The pedals are attached to potentiometers which vary the voltage according to the depression in the corresponding pedals. These voltage values can easily be read and calibrated to determine the depression-to-voltage value and accordingly use to set the PWM increase or decrease in the motor signal.
The pin-outs can be seen below in the figure.



http://www.tytlandsvik.no/momo/
Pedal Pin Layout

4. **Communications Switch**

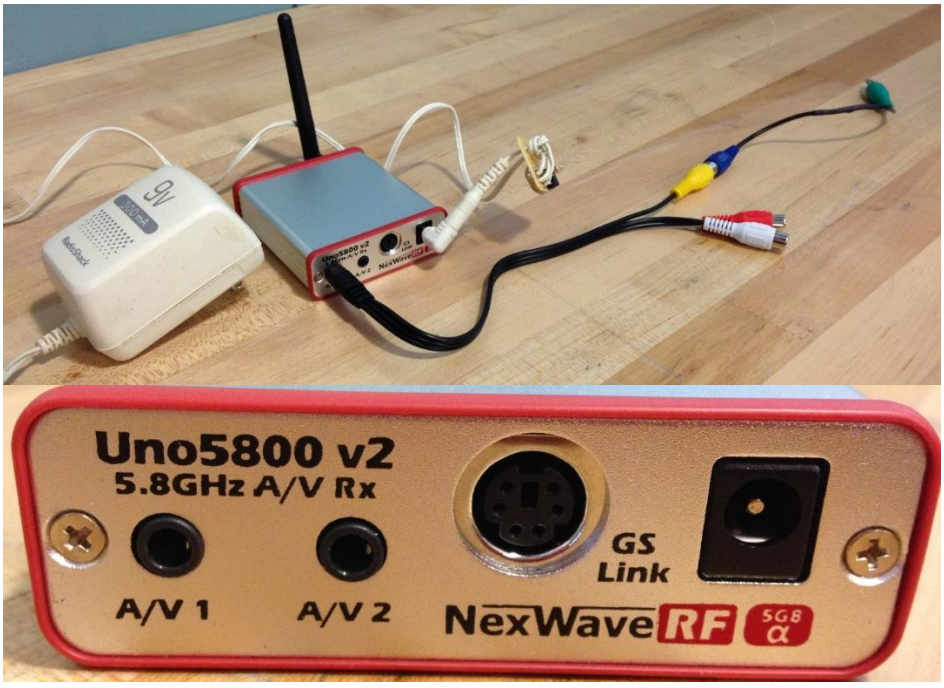
As final modifications to our system based on testing, a switch was added to the console that switches between two-way communications (scanning the received data on the console) and one way communication to the car. This switch connects PTB8 to ground on the console Kl25Z. This was added due to the fact that enabling scanning seems to cause the XBees to occasionally lose communications. Since this functionality is only needed for data logging, this switch makes it considerably easier to demo, removing the need to upload new code. We were unable to resolve this issue with the communications, but it would be an area to investigate in the future.



C. **Video RX**

The video from the RC car camera is transmitted over RF at 5.8 GHz (well outside the range of most other wireless communication protocols) to the receiver module on the console. This module is a Uno 5800 module designed specifically for RC planes that normally operates using a battery for power. Because of this, on startup the module will beep a number of times corresponding to the number of cells of the battery input voltage, and will continue to beep if the voltage drops below some specified levels. Being as in our application the module is powered by a wall DC converter, the voltage was dropped to within the specified levels for a 2S LiPo battery as specified in the user manual. This was done using a simple voltage regulator circuit. The kit comes from Ready Made RC and can be found here. Consult the user manual if the video is not being broadcast, as the button on the front of the module scrolls through the available channels if pressed and simply needs to be pressed into the correct channel is found again. As a final note, during operation the receiver will get slightly warm, which is expected and does not cause issue.

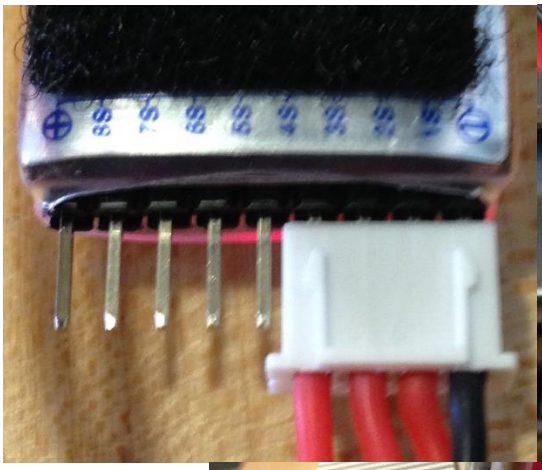
The current setup of the wireless video involves plugging the receiver through the DC barrel jack into the DC wall converter labeled “Video RX.” The wiring harness is then plugged into “A/V 1” on the video receiver and the yellow RCA on the left-hand side of the screen.

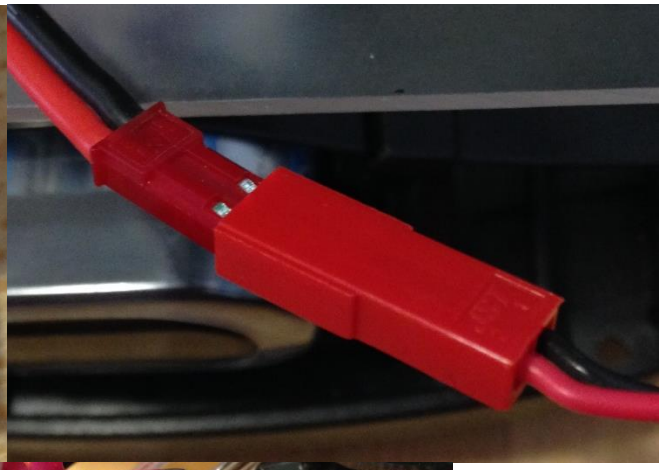


To see the image being broadcast, the TV channel has to be adjusted to “Video.” This is done using the Channel + and – keys on the right-hand side of the monitor. The selection options in order of decreasing progression can be seen below.

Channel	Function
0	N/A
PC	Used for Computer Image
HDMI2	N/A
HDMI1	N/A
Component	N/A
Video	Used for Camera Image
95	N/A

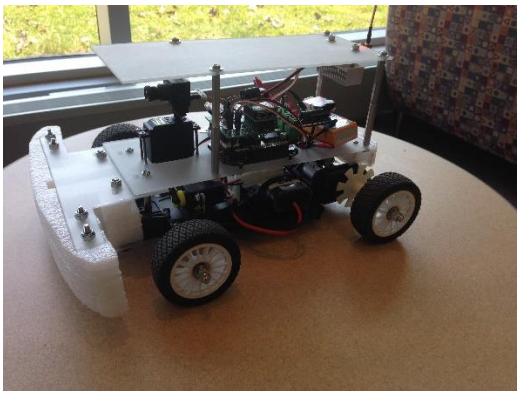
On the car side, the battery voltage indicator should be attached to the white four wire battery connector as indicated in the image below. This indicator monitors the voltage in all cells individually and will emit a warning when the voltage gets low. Be aware that the indicator will beep loudly when plugged in for the first time. The indicator can be stuck to the Velcro on the top of the battery, and the battery can be stuck to the Velcro on the car, as below. The red/black transmitter wire then needs to be plugged into the battery using the red plastic connectors.






35C
3S1P 11.1V

V. Car Electrical Information
On-Car Microcontroller & Shield Modifications



A. RC Car Onboard Control

The RC car operates utilizing the Freedom board provided by Freescale. To successfully interface the motors on the RC car with the microcontroller the Freedom board motor shield (Insert part number) was fitted on top of the microcontroller board. This however brought in significant limitations on the number of I/O pins that could be utilized for other sensors. To overcome this certain modifications had to be made on the motor shield. These modifications required the construction of a separate module which further altered the motor shield and utilized the pins in a manner that enabled easy access to various I/O pins on the microcontroller board. One important information to note is that all the external connections were done using combinations of male and female headers. This helped the connection of the different inputs and outputs easy and removed the probability of an erroneous connection.

B. Freedom Board Motor Shield

The Freedom Board Motor shield is a shield that is used to control the motors and allow for other sensors such as speed sensors, line camera, and servos to be controlled. This is mainly used in the Freescale Cup competitions but in order to be used for this projects use certain modifications were required. Those modifications are described in the next Section. This Section talks about the functionality of the shield as it arrived from the package. Only parts of the shield that are still being used in the project are described below.

1. Jumper Interface & Pin Names

The motor shield sits on top of the Freedom board and therefore occupies all the I/O pins on the micro controller. The different header names which will be referenced to in the later Sections of this guide are given below in Figures A.

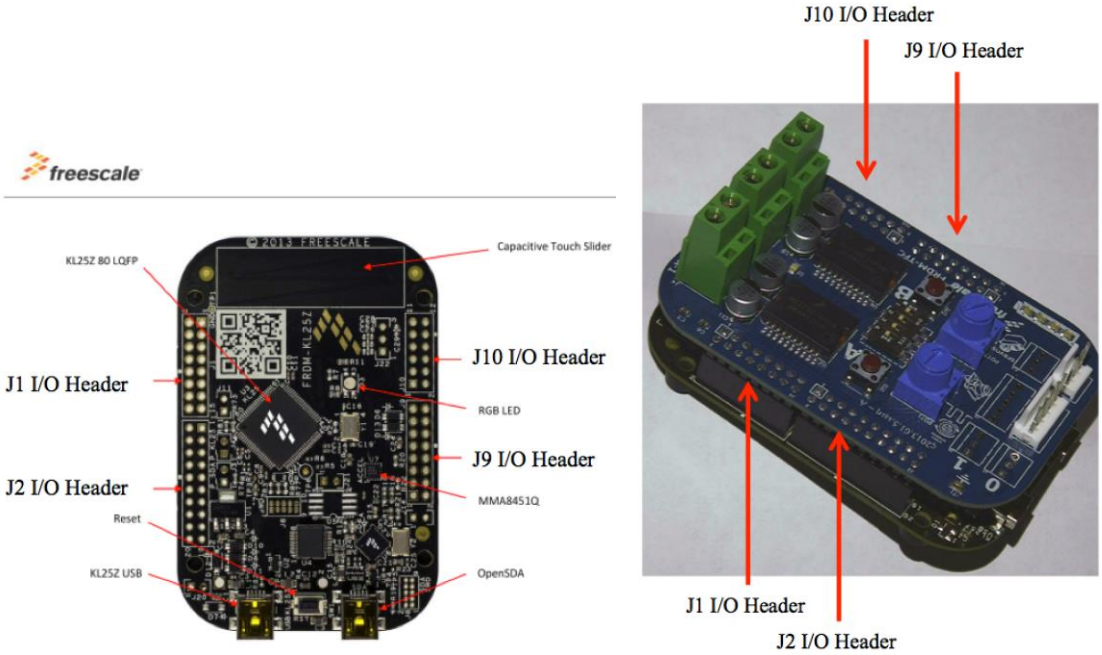


Figure A. - Freedom Board & Motor Shield Jumper I/O Headers

To describe these I/O further, Freescale has labelled the pins on the boards itself as well and each of these pins are given an alias for easy programming. Figure B shows the aliases and the pins for each female jumper. Table A shows the different pins and their mappings to the aliases for programming.

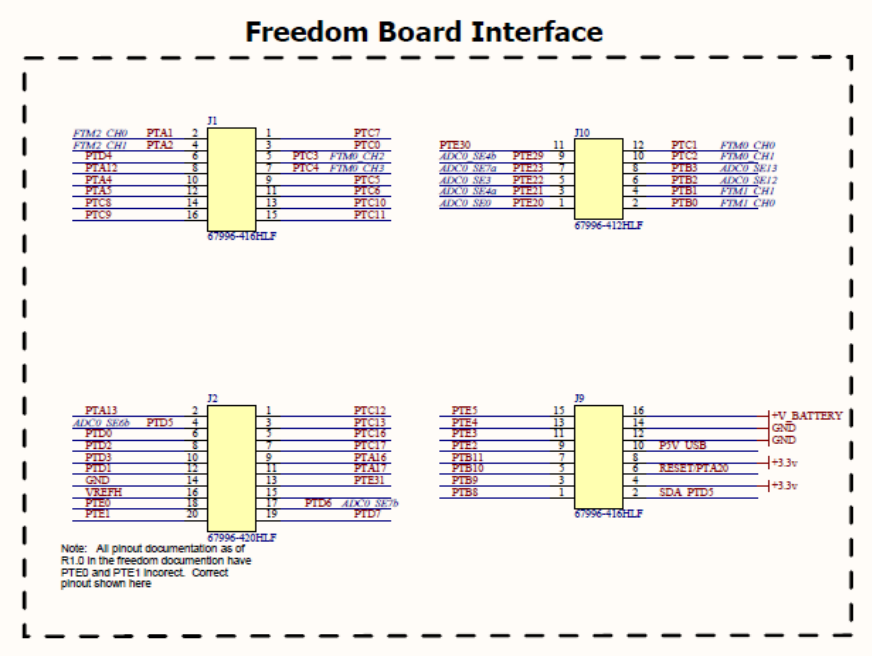


Figure B. - Freedom Board Interface

J9				J10			
Pin	Signal	Pin	Signal	Pin	Signal	Pin	Signal
J9 01	PTB8	J9 02	SDA_PTD5	J10 01	PTE20	J10 02	PTB0
J9 03	PTB9	J9 04	P3V3	J10 03	PTE21	J10 04	PTB1
J9 05	PTB10	J9 06	PTA20	J10 05	PTE22	J10 06	PTB2
J9 07	PTB11	J9 08	P3V3	J10 07	PTE23	J10 08	PTB3
J9 09	PTE2	J9 10	P5V_USB	J10 09	PTE29	J10 10	PTC2
J9 11	PTE3	J9 12	GND	J10 11	PTE30	J10 12	PTC1
J9 13	PTE4	J9 14	GND				
J9 15	PTE5	J9 16	P5-9V_VIN				

J1				J2			
Pin	Signal	Pin	Signal	Pin	Signal	Pin	Signal
J1 01	PTC7	J1 02	PTA1	J2 01	PTC12	J2 02	PTA13
J1 03	PTC0	J1 04	PTA2	J2 03	PTC13	J2 04	PTD5
J1 05	PTC3	J1 06	PTD4	J2 05	PTC16	J2 06	PTD0
J1 07	PTC4	J1 08	PTA12	J2 07	PTC17	J2 08	PTD2
J1 09	PTC5	J1 10	PTA4	J2 09	PTA16	J2 10	PTD3
J1 11	PTC6	J1 12	PTA5	J2 11	PTA17	J2 12	PTD1
J1 13	PTC10	J1 14	PTC8	J2 13	PTE31	J2 14	GND
J1 15	PTC11	J1 16	PTC9	J2 15	NC	J2 16	VREFH
				J2 17	PTD6	J2 18	PTE1
				J2 19	PTD7	J2 20	PTE0

Table A. - Pin Map for Freedom Board

2. **H-Bridge**

An H-Bridge is necessary for the fast switching of battery voltage and thereby controlling the speed of the motor. A PWM signal is sent to the H-Bridge where the optocouplers turn the motor on and off at high frequencies to reduce the overall voltage sent to the motors. The H-Bridges used for motor control are the ones fitted on the motor shield. The H-Bridge pins are already connected to the micro controller through the jumpers and these pins can be seen in Figure C below. Care must be given to not use these pins for any other use as the H-Bridges are necessary for RC car control.

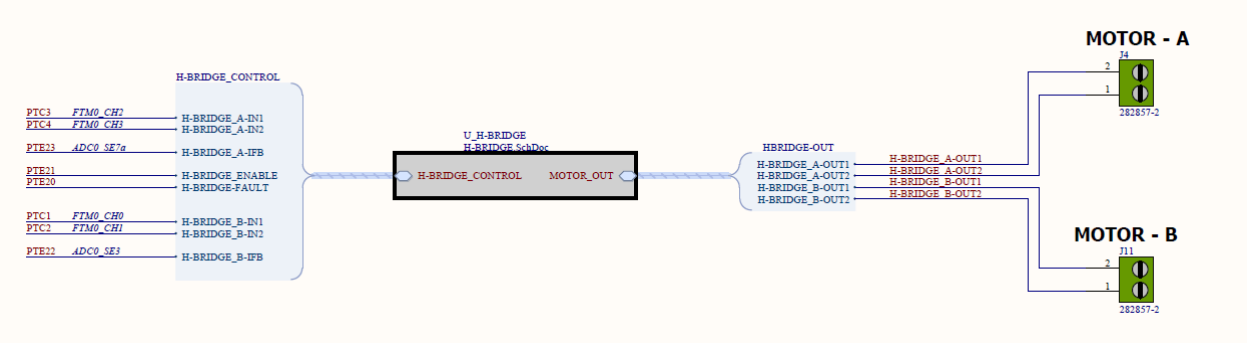


Figure C. - Motor H-Bridge Connections

3. **Potentiometers**

Potentiometers on the motor shield are necessary to trim the servos. These potentiometers are used for two servo trimming and centering. The connections to these potentiometers which are on the motor shield can be seen in Figure D below.

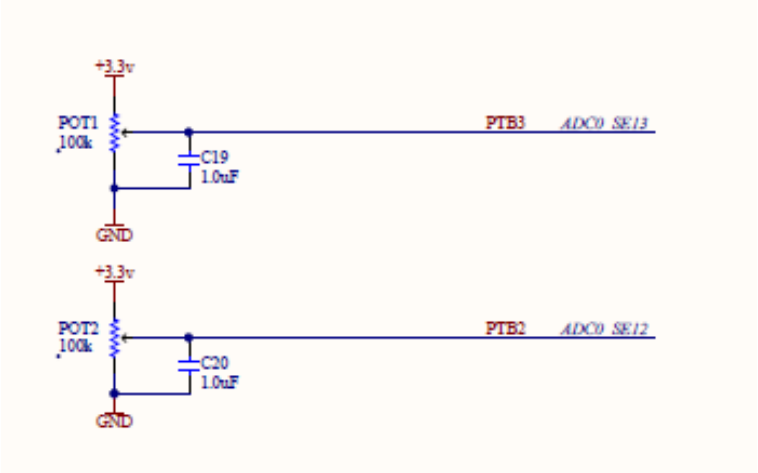


Figure D. - Potentiometer Connections

4. **Servos**

To connect to the servos the motor shield has two placements of male headers each internally connected to the appropriate I/O pins. Figure E shows the connections for the steering servo. This is similar to the additional servo connection available. To control the servo the necessary signals required are the power source, ground source, and input signal source. The camera servo had the same schematic except for the signal line being PTB1 instead of PTB0.

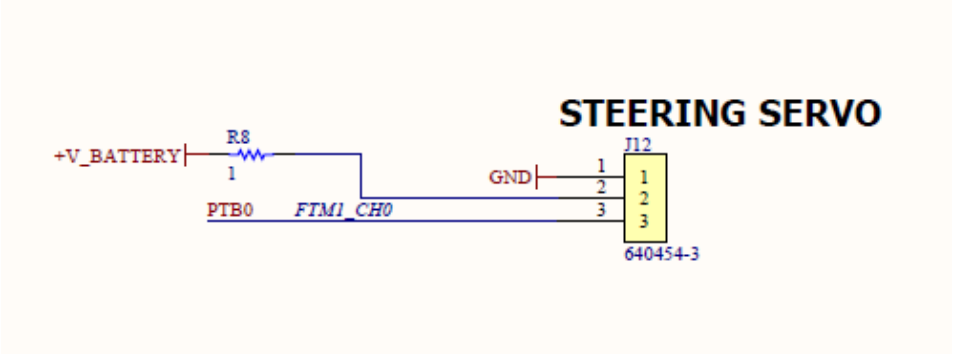


Figure E. - Servo Connections

C. **Freedom Board Modifications**

Removal of Pins & Direct Connections

Certain pins had to be removed from the motor shield and direct connections had to be made so as to enable the use of the encoder signals to accurately calculate the speed of the rear wheels as well as to accommodate for the sending and receiving of data by using an XBee. Table 1 shows which of the pins were used removed and the function implemented on those pins. The pins were chosen in such a manner that after the removal of the connection between the motor shield and the Freedom board, there was no loss in original functionality. Only the pins removed for XBee transmission and receipt of data affected the original functionality as those pins were being utilized for the Line Scan camera. This however did not affect the operability of the car since the Line Scan camera is not utilized for the purposes of this project. The removal was necessary to avoid any cross talk between the motor shield's H-Bridge and also to remove any unnecessary signals coming in from the motor shield board due to prior connections. Figure 1 shows the external wires that were attached to the Freedom board directly.

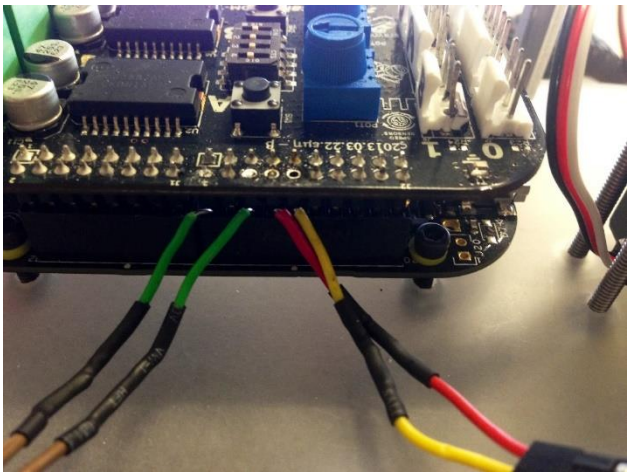


Figure 1. - Motor Shield Direct Wiring

Table 1 - Direct Connection Wiring Map			
Pin Functionality	**Pin Alias**	**Jumper Location**	**Jumper Pin Number**
Left Encoder Input	PTA13	J2	2
Right Encoder Input	PTD0	J2	6
XBee Receive	PTD3	J2	10
XBee Transmit	PTD2	J2	8

D. **Motor Shield Modifications**

Addition of Headers

Originally, the motor shield utilized the I/O pins by enabling access to them with the use of certain male headers. These headers were designed to perfectly accommodate for the sensors required to compete in the Freescale Cup. These sensors were the Line Scan cameras, speed sensors, and servos. Additional male headers were added to fill both the slots on the motor shield board as can be seen in Figure 2 below. These headers enabled the operation of two servo control signals; one for the steering, and one for the RF camera attached on the car.

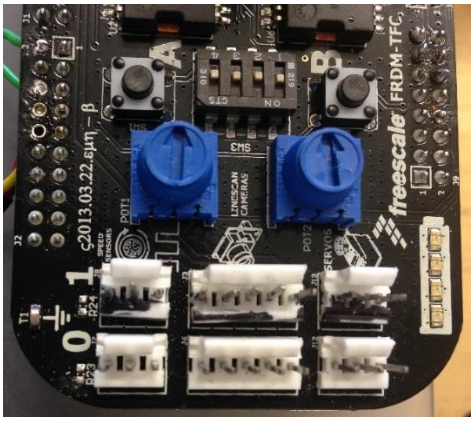


Figure 2. - Additional Headers on Motor Shield

The servo headers can be seen on the right and are labeled n the board with serial numbers 0 and 1. The 3-pin female headers are attached to the servo wires and they connect to the male adapters

on the adapter board. The motor shield jumpers were modified for access to custom sensors. The jumpers on the motor shield were changed from what they were originally. This change can be seen below in Figure 3.

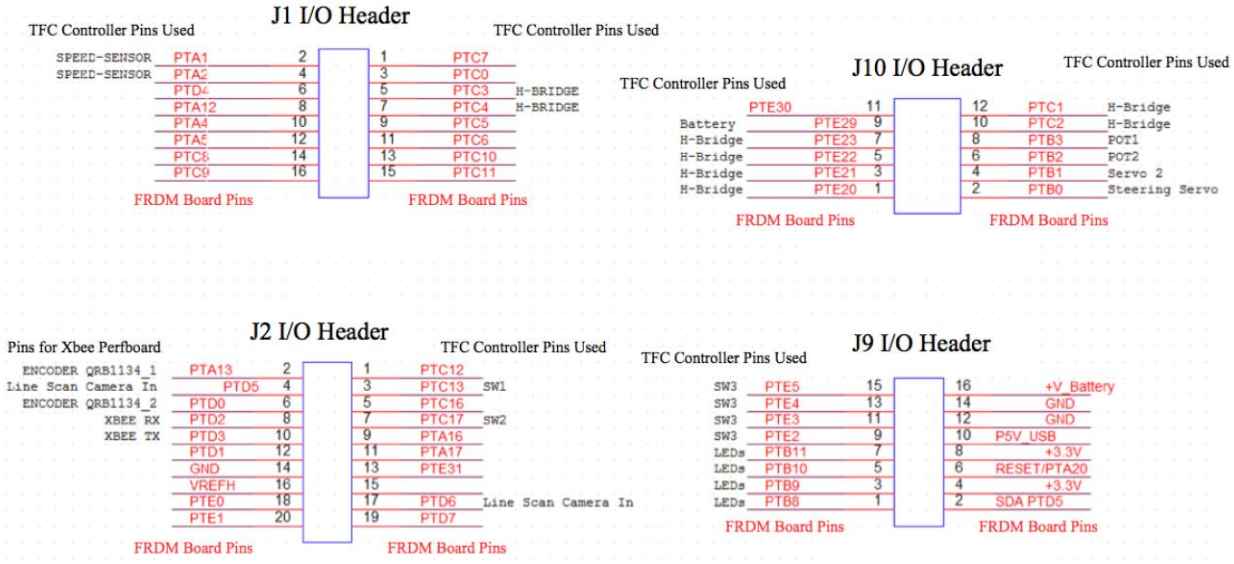


Figure 3. - Freedom Board Motor Shield Jumper Modifications

E. Adapter Board

An adapter board had to be made so as to be able to obtain access to certain pins on the micro controller board as the motor shield board restricted access to a majority of the pins. This adapter board also enabled the placement of necessary passive devices such as pull up and pull down resistors for the encoder signals. It also housed pins that provided power and ground. On top of this adapter board an XBee module was also placed. To access the trimming potentiometers placed on the motor shield two holes had to be drilled into the Adapter Board. These can be seen in the figures below as well.

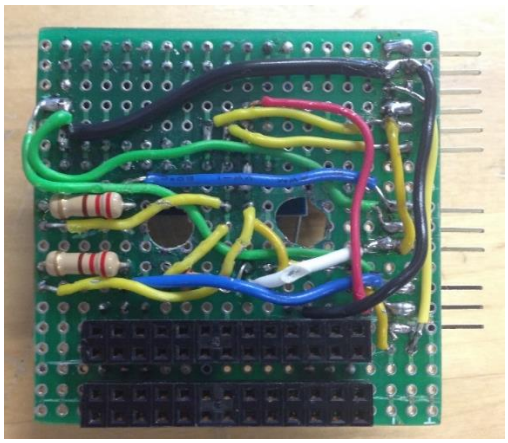


Figure 3. - Adapter Board Top View

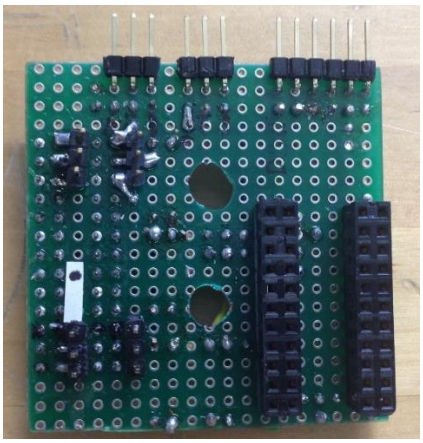


Figure 4. - Adapter Board Bottom View

The schematic for the adapter board can be seen in Figure 5 below. This schematic was made in CadSoft Eagle 6.5 software. In the center can be seen the XBee module. To the top are the JP10 and JP9 headers. Also on the extreme top is the capacitor and switch additions, described in later Sections, fitted across the battery. The battery used was a 7.2 Volt, 3000 mAh NiMH battery pack. The extreme bottom has the other side of the jumpers from the motor shield to which further input and output signals are made. Towards the right can be seen the speed encoders which are actually attached externally to the wheels.

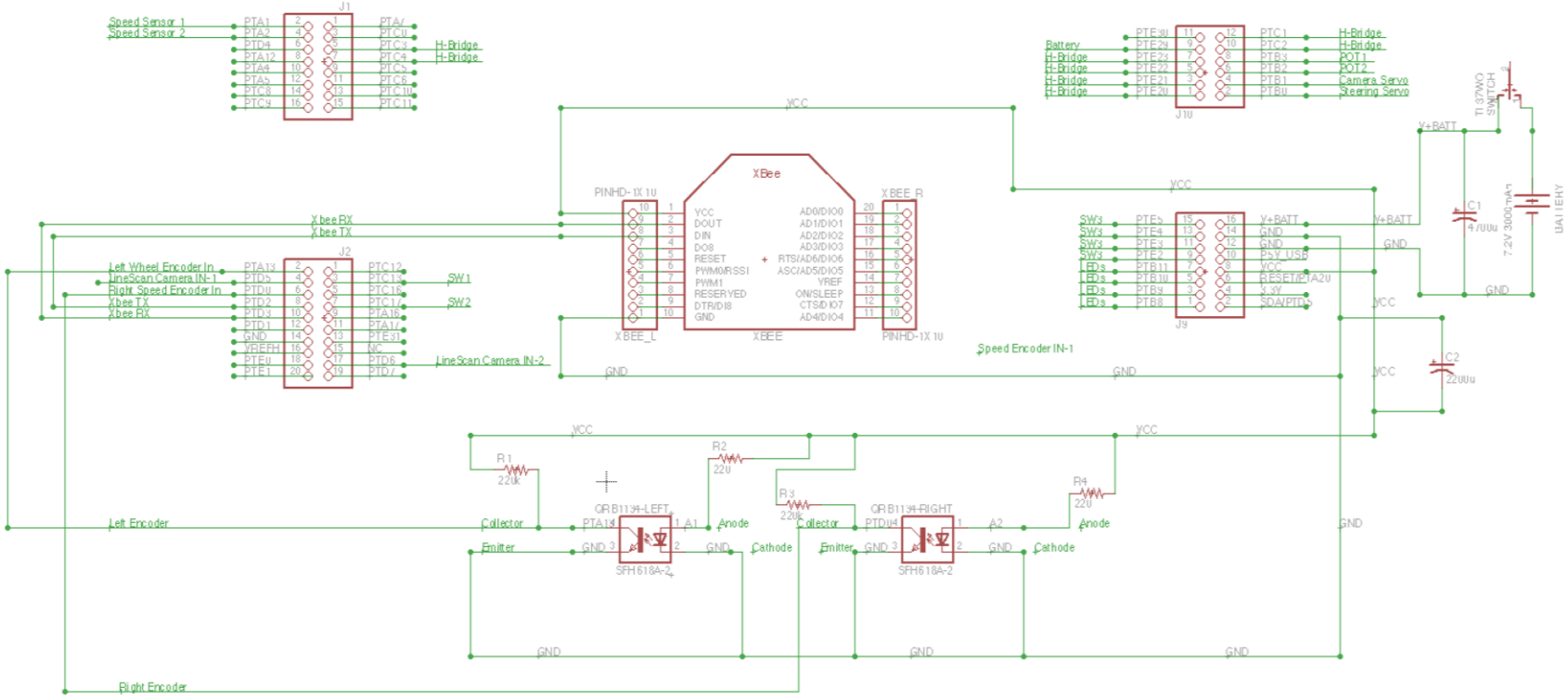


Figure 5. - Schematic of the Adapter Board

1. **Placement of Adapter Board**

The adapter board was placed on top of the motor shield. This required some modifications, which were described in the previous Section, by the addition of male headers. A harness as can be seen in Figure 6 below was attached on the Adapter Board and it sat perfecto on top of the male pins of the motor shield. This also allowed access to the male pins through the female dual line headers.

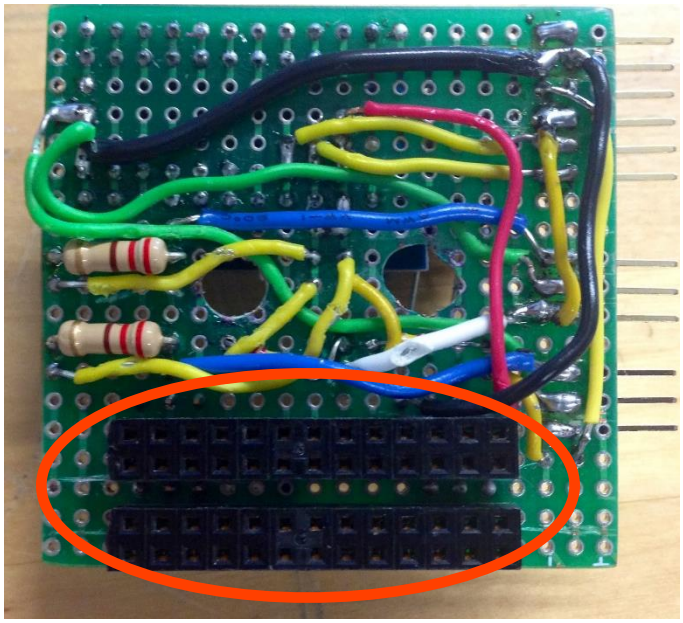


Figure 6. - Sitting Harness for Adapter Board

2. **XBee**

The XBee module is used to communicate between the MOMO steering wheel and the RC car. One of the XBee receiver-transmitter modules had to be fitter on the RC car and was harnessed on the Adapter Board. The XBee module was placed on female header pins, as shown in Figure 7 below, and was powered, and connected, by the input bus connections. The only connections required were the power, ground, transmission, and receipt. Table 2 shows the pin layout.

Table 2 - XBee Pin Connections

Pin Functionality	**XBee Pin Number**	**Jumper Number**	**Jumper Pin Number**
Power (Vcc)	1	J9	8
Ground	2	J9	14
RX	10	J2	8
TX	3	J2	10

Figure 7 below shows the female harness for the XBee. This harness requires the XBee to be placed in a certain direction so as to operate correctly. The direction is induced by a sticker on the final Adapter Board.

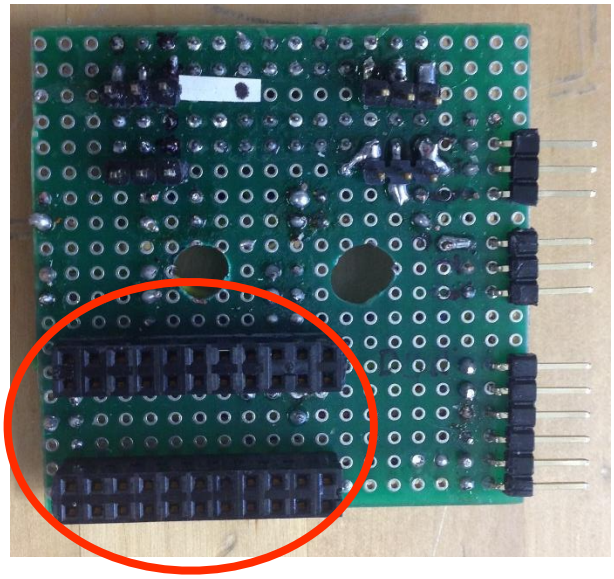


Figure 7. - XBee Harness

3. **Input Bus**

The input bus is simply a collection of wires connected to a single line female header. These wires carry the input signals from the various sensors and modules such as the encoders, Freedom Board, servos, power, etc. Figure 8 shows the different input busses that are in use.

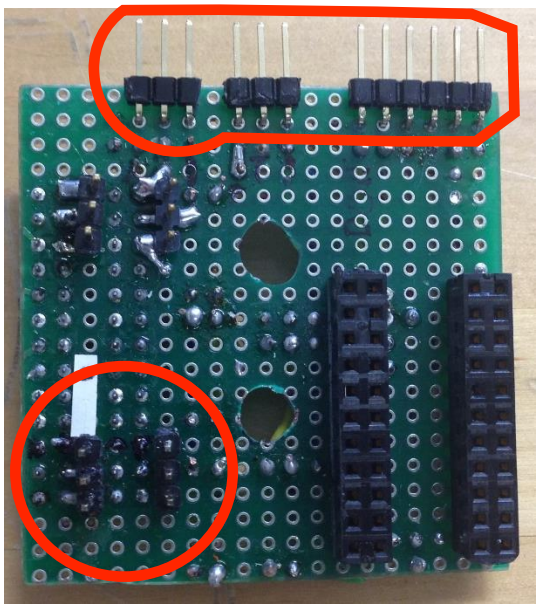


Figure 8. - Input Bus

The two 3-pin male bus connections towards the bottom left are for the steering and camera servos. It did not matter which servo pin was connected to which 3-pin header as the servos were scaled 1:1 ratio. These connections were exactly the same as that for the original Freedom Board Motor Shield as it is simply a furthering of the male headers originally on the shield. Towards

the top are the bus headers for other signals to and from the XBee module. Table 3 shows the different pins being used. Pin numbering starts from the left of Figure 8.

Table 3 - Input Pin Bus Layout			
Adapter Board Pin Number	**Pin Functionality**	**Jumper Number**	**Jumper Pin Number**
1	Left Encoder Collector	J2	2
2	GND	J9	12
3	Left Encoder Anode	-	-
4	Right Encoder Collector	J2	6
5	GND	J9	12
6	Right Encoder Anode	-	-
7	XBee TX	J2	8
8	XBee RX	J2	12
9	Right Encoder Input	J2	6
10	Left Encoder Input	J2	2
11	GND	J9	12
12	NC	-	-

4. **Servo Pin Extensions**

The servo extension pins, as described in the previous Section, are simply 3-pin male headers brought on the Adapter Board from below via the Adapter Board placement harness. This does not change functionality in any manner and only acts as male pins coming out of the Adapter Board rather than the motor shield itself. Figure 9 shows the servo pin extension.

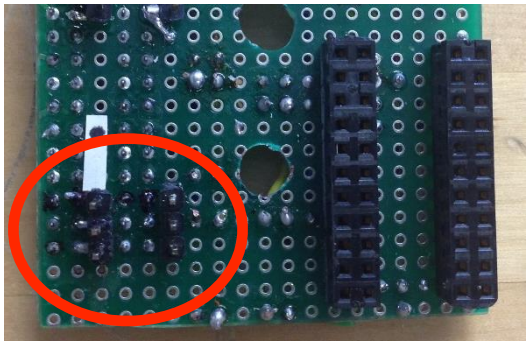


Figure 9 - Servo Pin Extension

5. **Encoder Input Bus**

Described previously, and with the pin layout from Table 3, the encoder pin bus can be seen on the left side of the Adapter Board in Figure 10 below. These pins act as connections to the encoder placed on the wheel. Figure 10 shows these pins.

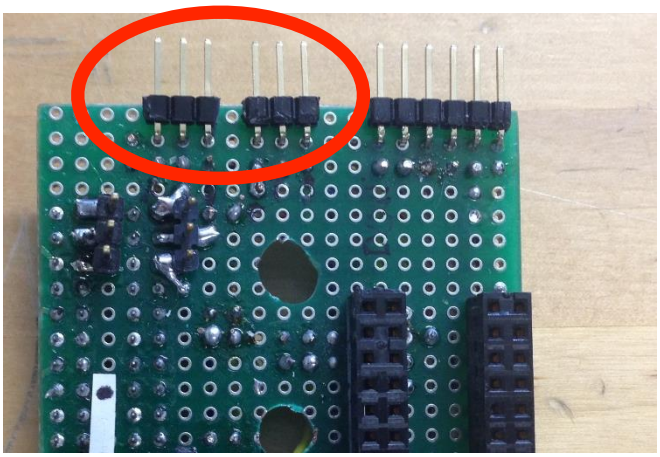


Figure 10. - Encoder Pin Bus

The pins for the two encoders are similar and are divided into 3 pins. The pin layout can be seen on the QRB1134 data sheet but on the Adapter Board the pins are as follows:

1) Leftmost pin in Figure 10 - Collector Pin
2) Middle pin in Figure 10 - Emitter and Cathode Pin
3) Rightmost pin in Figure 10 - Anode Pin

Finally, with all the connections made, the adapter board placed on top of the motor shield can be seen in Figure 11 & 12 below.

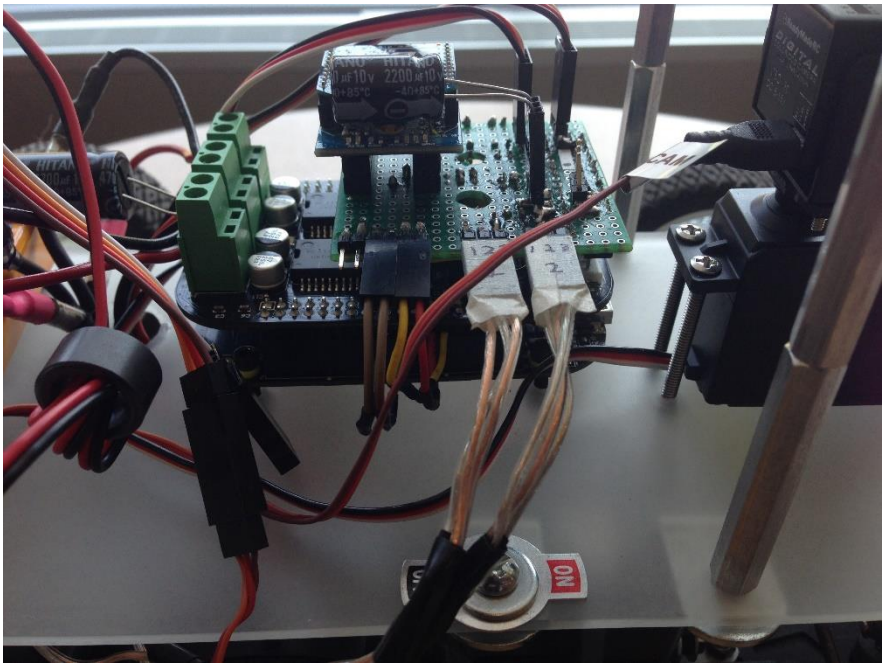


Figure 11. - Adapter Board on top of Motor Shield Side View 1

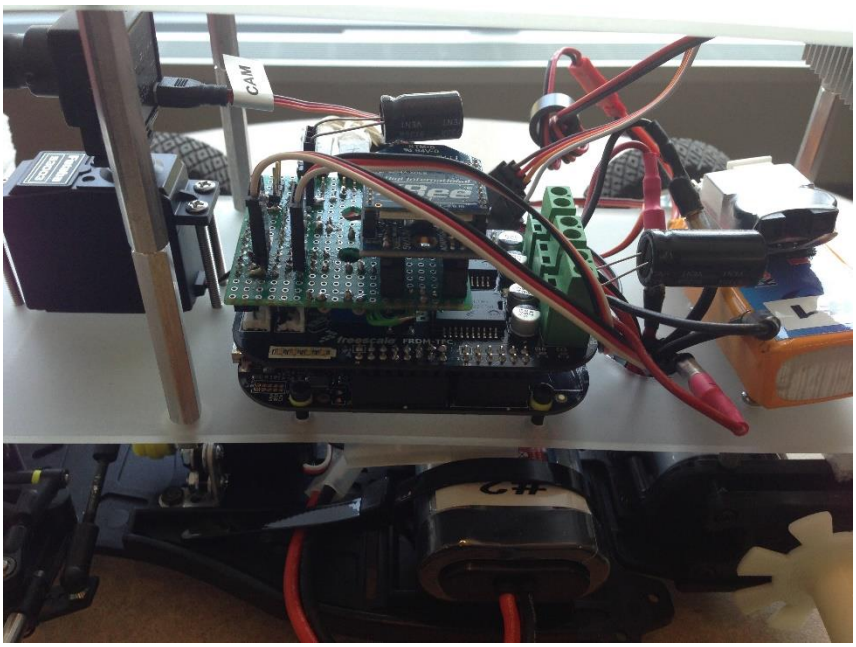


Figure 12. - Adapter Board on top of Motor Shield Side View 2

F. Other Modifications

Certain other modifications were made to the overall setup to reduce noise, and obtain better signals as inputs.

1. Ferrite Core

A ferrite core was added as a choke to the two batteries being used to power the whole apparatus. One choke was placed on the positive and negative terminals on the RC car battery, as seen in Figure 13, and another choke was added to the RC Camera battery as well, as seen in Figure 14. The whole purpose of the core is to prevent eddy currents as the battery is being used to power the motors.

Figure 12. - Ferrite Core Choke on RC Car Battery

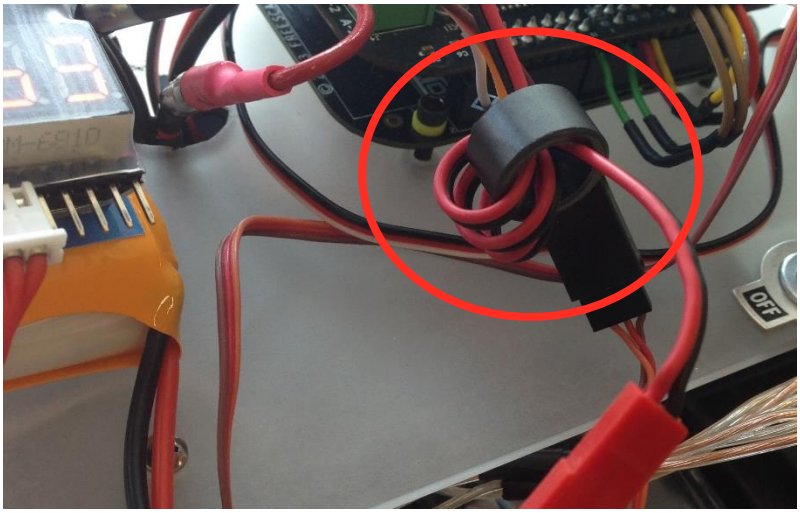


Figure 13. - Ferrite Core Choke on RC Camera Battery

2. **Filtering Capacitors**

Due to the use of H-Bridges to control the motor speed there was a lot of noise on the power terminals of the motor shield which in turn caused all the pins to have the high frequency noise. To remove this high capacity capacitors were added on the battery power terminal as well as the 3.3 volt and ground line on the Adapter Board. These capacitors can be seen in Figure 14 and 15 below.

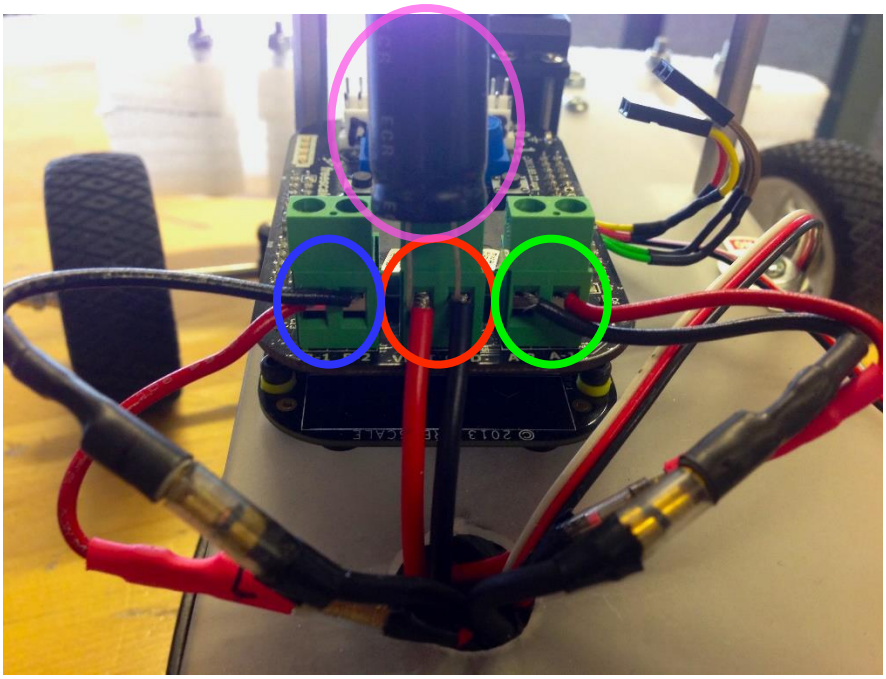


Figure 14. - Capacitor Added Over Battery Terminals

As seen in Figure 14, the center harness of the motor shield allows for connection of the battery power. The left is connected to the positive terminal and the right is connected to the negative terminal of the battery. The capacitor added is also seen in the figure.

To the right can be seen the green screw-in harness of the right motor. The connections for this had to be reversed so as to make the car go forward as the motor on the right side is flipped. The positive terminal of the motor is connected to the negative terminal of the harness and vice versa.

To the left can be seen a green screw-in harness of the left motor. The connections for this were similar to the power harness connection with the positive side of the harness (left) connected to the positive terminal of the motor and the negative side of the harness (right) connected to the negative terminal of the motor.

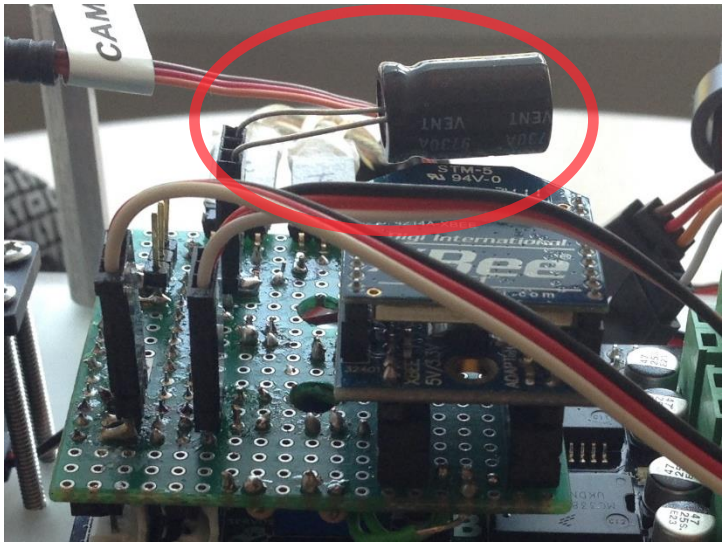


Figure 15. - Capacitor Added Over Power and Ground Pins

In Figure 15 can be seen the capacitor added on top of the power pins for the adapter board. This capacitor sits over the 3.3V and the GND pin which powers all the circuitry on the adapter board. These capacitor placements, known as decoupling capacitors or bypass capacitors, are generally placed across the power and ground for any electronics that needs to be isolated from noise.

Furthermore, to reduce the motor noise, additional capacitors were added to the motor terminals as can be seen in Figure 16 and 17 below.

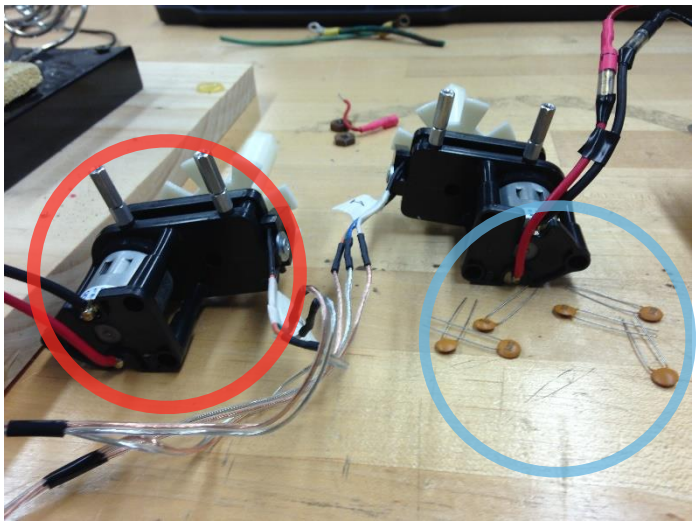


Figure 16. - Motors without Capacitors

Figure 17. - Motors with Capacitors

One major drawback to working with motors is the large amounts of electrical noise they produce which can interfere with your sensors and can even impair your micro-controller by causing voltage dips on your regulated power line. Large enough voltage dips can corrupt the data in the micro-controller or cause it to reset.

The main source of motor noise is the commutator brushes, which can bounce as the motor shaft rotates. This bouncing, when coupled with the inductance of the motor coils and motor leads, can lead to a lot of noise on your power line and can even induce noise in nearby lines.

The solution is to solder capacitors across your motor terminals. Capacitors are usually the most effective way to suppress motor noise, and as such we recommend you always solder at least one capacitor across your motor terminals. Typically you will want to use anywhere from one to three 0.1 µF ceramic capacitors, soldered as close to the motor casing as possible. For applications that require bidirectional motor control, it is very important that you do not use polarized capacitors!

The capacitors seen in Figure 17 were soldered across the motor terminals directly for each motor to achieve sufficient noise suppression. Instead of electrolytic capacitors the ceramic capacitors were used to account for the dual direction nature of the motors since both reverse and forward direction operations are being performed by the RC car.

Other precautions that were observed to reduce noise were that the motor power wires were kept very short and the motor lead wires were spiraled around each other to decrease the noise as well. Care was also given to keep the motor power cables away from the electronics, such as the XBee, for fear of induced currents on the signal lines.

3. **Power Switch**

A power switch was added to the car for safety purposes. Although it is not included in the Freedom board documentation, if the board is powered by an external battery source in addition

with the USB programming interface, the micro controller seemed to get damaged every time. The cause for this was unknown but the solution was a very simplistic one. A power switch, as seen in Figure 15, was added to turn the battery power off whenever the microprocessor needed to be programmed. The schematic can be seen in Figure 16.

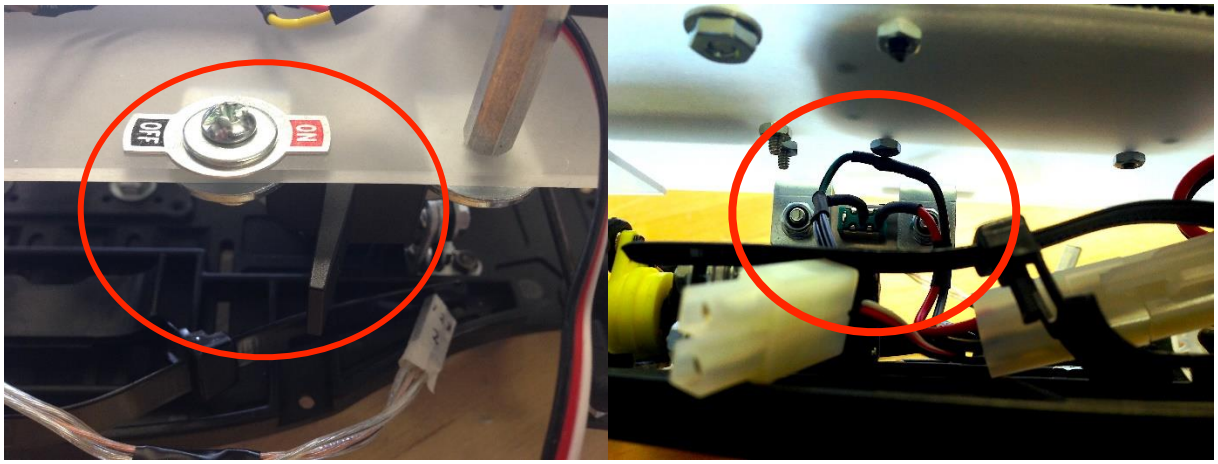


Figure 15. - Power Switch and Wire Connections Placed Discretely On Car Chassis

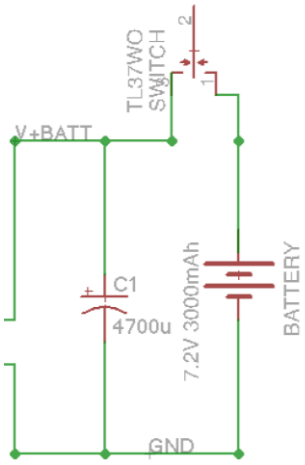


Figure 16. - Schematic for Switch

4. **Encoders**

To successfully apply the use of control systems and a PID loop the wheel speeds were needed to be measured. For this encoder sensors were attached to the wheels. The sensors used were the QRB1134 sensors. The wiring for these sensors can be seen in Figure 17 below. The collector, emitter, anode, and cathode internal wirings can be seen in the sensor datasheet or in Figure 18.

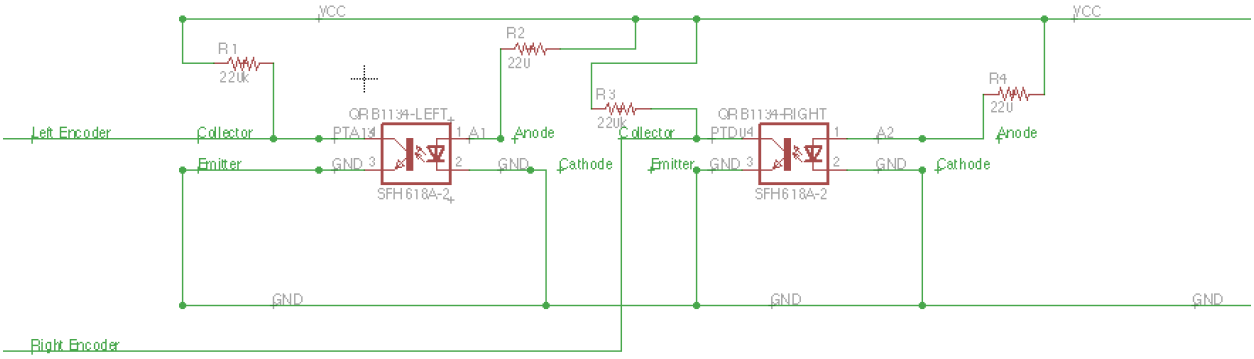


Figure 17. - Encoder Sensor Wiring Diagram

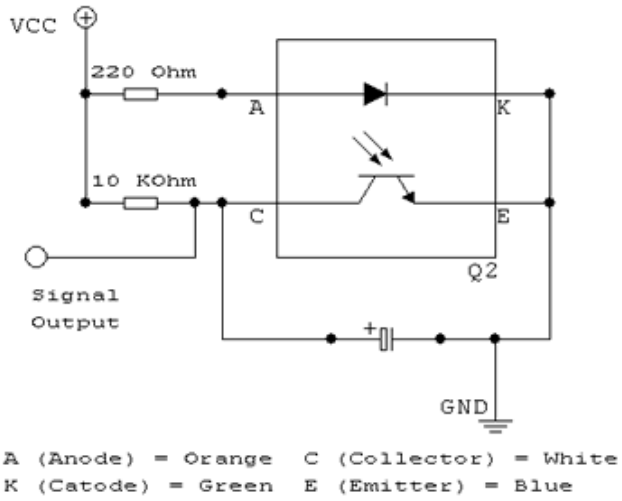


Figure 18. - Encoder Sensor Hookup

5. **Camera TX**

The camera and transmitter used for the car are standalone modules that can be seen in the links under the “Video RX” Section. Nothing should have to be modified with the setup apart from replacing the separate 3S LiPo 11.1V 35C battery for the transmitter when the battery indicator warning goes off. The transmitter has a heat sink for thermal dissipation at the back of the car and has no user settings. The camera is powered through this transmitter and data is transmitted through the same cable. The servo that articulates the camera is controlled off of the microcontroller shield.

Testing of the runtime of the camera and transmitter from a single LiPO battery is as follows. Starting at a max voltage of 12.5V, the camera and transmitter broadcast a good quality image for 2 hours and 55 minutes, setting the voltage indicator off at 10.1V. During this time the heat sink reached a steady state temperature which is warm to the touch but not hot, which is a worst

case airflow scenario and is desirable. This is significantly longer than the runtime of the car battery, so that metric is ideal.

VI. Wireless Communication Information

A. XBee Distance Stats and Info

For this project, XBee 802.15.4 XB24 2mW chip antenna modules are used for the data transmission. From the detailed XBee guide, these XBees operate in the 2.4 GHz range and have an operating distance significantly affected by Line of Sight and other interference sources.

Range testing of the XBees was done in the Brinkman hallway in Building 9. A table was set up at the end of the hallway and data was successfully collected from the Xbee all the way to the other end, which was measured to be approximately 141 ft. The XBees also transmitted through glass to the outside, but the range was greatly reduced. Walls seemed to cause packet loss after the 141 foot mark, but for our purposes this is more than adequate.

B. Camera Distance Stats and Info

Range testing of the wireless camera modules was done in the Brinkman hallway of building 9 as well. Starting with the monitor and receiver at one end of the hallway, the image was still acceptable quality at 141 ft (the end of the hallway). The image had some interference while moving at all distances. The image was still broadcast as far away as Xerox auditorium, estimated at over 200 ft away with many obstacles in between. Based on these results it is clear that the camera range surpasses the XBee range, which was the target of our design.

Following testing, it seems that interference in the image quality can be caused by obstacles, outside signals, and sharp movement of the transmitter. Most notably, metal objects tend to cause significant interference and should be removed from the operating area.

A rendering of the field of view relative to what a person would see normally when using the camera can be seen below. Only the horizontal limitations are considered, as it is difficult to limit the view in the vertical direction. As can easily be seen, the limitations of camera FPV are notable but still reasonable for driving purposes.

C. X-CTU, Updating, and Bricking Issues

The primary debugging program for the XBees is the X-CTU software from Digi. Download the "XCTU Next Gen Installer, v. 6.1.0, Windows x32/x64" and install the software (this is the newest version). In this program you can "Add" or "Discover" the XBee Devices connected over USB using the buttons on the top left of the interface. Once you go through these initializations (should be all the default settings: 9600 baud, 8 data bits, 1 stop bit, no parity, no flow control,

etc.), you can modify the "Configuration" and monitor the "Console" outputs using the buttons on the top right of the interface.

If you simply double click the listed XBee device, some basic settings that are worth mentioning can be found below. The channel setting for both XBee devices (yes, you have to repeat this process with both XBees) should be the same (here we are on channel C) as well as the PAN ID's of both devices. Similarly, DD, IC, and IR should be the same value for both XBees. To make the XBees communicate directly with only the each other specifically, one XBee has to be set with DL and MY values (here we used 10 and 20 to keep it simple). The other XBee should be set with DL equal to the MY value of the other XBee, and the MY value of the second XBee should be the same as the DL value of the other XBee. By default the DL and MY values on both XBees will be 0, which means there is no direct pairing. For further settings and more detailed explanations of the functioning of the program, consult the Digi website and the Getting Started Guide for our specific modules.

XBee	**Console**	**Car**
Parameter	***Value***	***Value***
Ch	C	C
PAN ID	3332	3332
DL	10	20
MY	20	10
DD	10000	10000
IC	0	0
IR	0	0

An additional program that is useful for debugging is the Putty terminal window application. This can be configured to read what data is being transmitted to and from the XBees. This program is a bit more in depth for how it can be used, so consult tutorials for doing so.

The XBees used for this project seem to be prone to "bricking" or going into a mode that yields them unusable when used incorrectly, so some information will be given here on the issue. The following link here can give some insight into how to sort out the problem, but the general gist can be summed up as follows.

- Connect the XBee that seems to be causing problems (normally the LEDs will not blink quickly as they should when data is being sent) to the Xbee Explorer USB breakout board. Connect this to the computer and open X-CTU.
- The newer version of X-CTU may have some troubleshooting functionality (we have not had any issues since the new version was released), but what you can do is set the Baud to 115200 and the flow control to Hardware. This forces the XBee into its max hardware settings. Once you change this setting, try reading the XBee data to see if the current version is recognized.
- Once done with this (working or not) change the XBee settings back to 9600 Baud and hardware flow control and repeat the process. After this you should be able to remove the flow control and use the XBee as intended. Some important settings to check are below.
- Protocol should be XB24 802.15.4 and the version this was developed with is 10EC.

It would be advised for this setup to never update the firmware for either XBee, as the current configuration is stable and further changes are unnecessary,

D. Using Xbee's with MBED

XBee modules were used to transmit data between the console and the car. In MBED, the XBee is treated as a serial device. In the console code, PTD3 and PTD2 are the RX and TX pins that connect to the XBee's data pins. In order to transmit data, the printf and scanf functions were used. Seen below are some statistics gathered for data transmission of the XBees used.

XBEE data rate = 250,000 bits/sec
Packet size = 48 bits
Maximum packet frequency = 5208 packets/sec

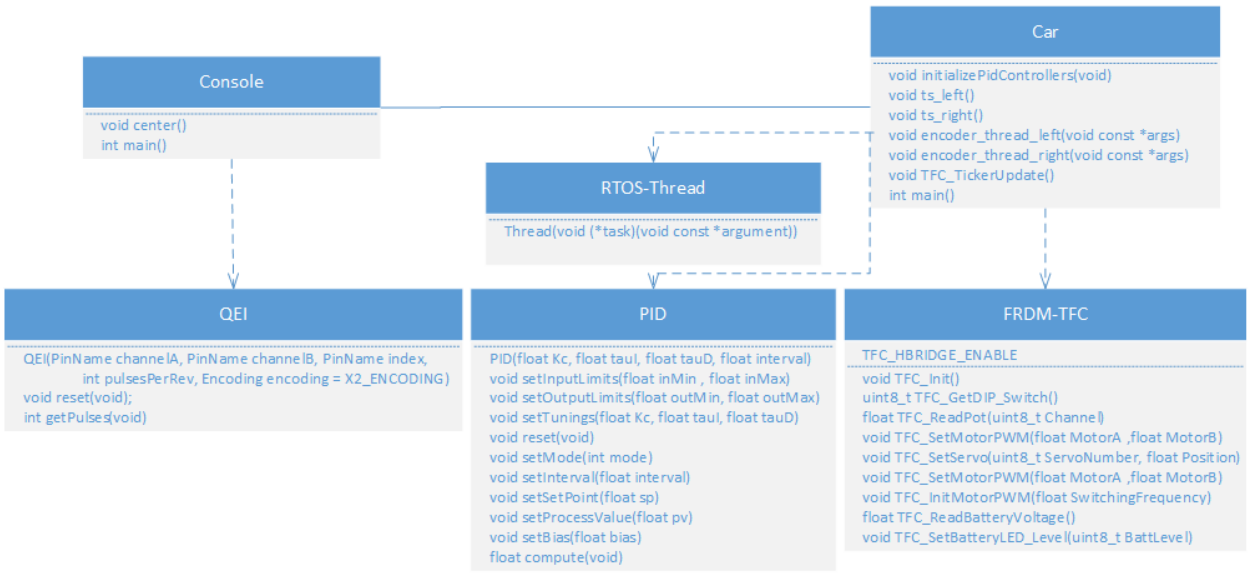
VII. KL25Z and MBED

As both the console and car use KL25Z microcontrollers, we used mbed as our primary compiler. This online software is openly available and has a wealth of documentation and premade libraries submitted by an online community of users that allow for easy implementation of many basic functions. For official directions and getting started with the KL25Z in MBED< the best reference is the handbook dedicated to the board. The major steps required in setting up the board with any new computer are as follows:

1. Upgrade to the latest firmware.
2. Install the Windows serial port driver.
3. Create and download a program to the KL25Z.
4. Press the reset button to run the newest program.

A. Coding Logic Flow Chart

The following UML was created to show an overview of the coding done in this setup. This shows the interplay between the console and car microcontrollers, along with the libraries used for on each system.



B. Console Code Documentation

The console code makes use of the MBED QEI (quadrature encoder) library in order to capture the position and direction of the Logitech wheel. A digital interrupt was connected to the centering signal from the Logitech wheel. This provided the ability to correctly zero the position and direction of the wheel.

Analog inputs were used to read the potentiometer values from the Logitech gas and brake pedals. The Logitech console drew power from a 5V DC adapter. This provided the console with a constant voltage so that the potentiometer values remained in the same range. The range for the pedals was determined by recording the raw values when the pedal was not pressed and when it was fully pressed. This range was determined to be quite large. Unsigned characters were used

to transmit the data, therefore, the pedal range was compressed down via a scaling factor. Similarly, a scaling factor was also used to compress the wheel position's range.

The unsigned char was chosen to be the transmission data type. The char type was well suited for this task because of its minimal bit size. The char type is 8 bits, versus the integer type which is 32 bits. Since wireless bandwidth should be conserved, characters use is critical in order to maximize message throughput.

The code first calls the libraries and initializes the PC (USB) and XBee (TX: PTD3 and RX: PTD2) communications. It then sets up to read analog in values on PTB1 for the gas pedal, PTB0 for the brake pedal, and reads QEI values using PTD5 and PTD0 with 624 pulses per revolution (based on testing). PTA13 is setup as an interrupt pin for the zeroing signal, and sets PTB8 as a digital in pin for enabling and disabling two way communications using the switch in the console.

```
//Libraries
#include "mbed.h"
#include "QEI.h"

//SERIAL Information
Serial pc(USBTX, USBRX);
Serial XBee(PTD3, PTD2);
int baud_rate = 9600;

//CONSOLE PIN Information
AnalogIn gas_pedal(PTB1);
AnalogIn brake_pedal(PTB0);
QEI wheel (PTD5, PTD0, NC, 624);
InterruptIn zero_sig(PTA13);
DigitalIn enableTwo(PTB8); //toggle two way communication
```

The wheel range is initialized at 650 to make sure that the values of the wheel are always inside the range (since sometimes the count gets a bit messed up at higher turning speeds). Then the wheel, gas, brake, and encoder variables are initialized.

```
//CONVERSION Variables
//Steering Wheel
int wheel_range = 650; //established through testing, larger than actual
range so no characters are lost
int wheel_int; //wheel values
int wheel_pos_char; //character range conversion for positive wheel values
(extend range to 93*2 characters)
int wheel_neg_char; //character range conversion for negative wheel values
(extend range to 93*2 characters)
//Pedals
int gas_val; //gas pedal values
int brake_val; //brake pedal values
int gas_int; //character range conversion of gas pedal values
int brake_int; //character range conversion of brake pedal values
//Encoders
unsigned char encoder_left_char; //received left encoder character range data
```

```
unsigned char encoder_right_char; //received right encoder character range
data
```

The centering ISR is setup to only reset the wheel count whenever the interrupt is triggered by the wheel passing the center position with the optical gate. Debugging code is also commented out which can help for testing.

```
//Centering ISR
void center(){
    //pc.printf("Center\r\n"); //for debugging
    //Reset count to 0 when falling edge detected
    wheel.reset();
}
```

The main loop contains three major parts. It reads the raw wheel and pedal information. Then it adjusts the ranges and checks for invalid data. Finally, it transmits the data to the car, via the XBee.

The first part enables the serial communications and interrupt ISR on falling edges.

```
int main() {
    //Initialize XBee and PC communications
        pc.baud(baud_rate);
        XBee.baud(baud_rate);

    //Interrupt Centering ISR on signal falling edge
        zero_sig.fall(&center);
```

The while loop then runs indefinitely for transmitting data. The 20 ms wait time is setup to mimic the car code wait times used in the PID loop. This value was established experiementally through testing and works very well for smooth vehicle turning without overloading the transmissions.

```
    while(true) {
    //XBee Data Transmission
        wait(0.05); //data tx / rx speed control
```

The pulse count of the encoder is then sampled and stored (with 624 pulses for a full revolution, or one side of the steering stop to the other). The gas and brake pedal values are then captured and converted to values between 35 and 126 for transmission purposes. This gives adequate resolution for user input. The range of 35 to 126 was established so that no bad characters are sent that interfere with communications reading and writing (such as carriage return, new line, and quotes).

```
    //Record steering values
        wheel_int = (int)(wheel.getPulses());

    //Record pedal values and conver to character ranges
        gas_int = 126-(gas_pedal.read_u16()-15000)/500; //should output 35-
        126
        brake_int = (brake_pedal.read_u16()-15000)/460+35; //should output
        35-126
```

Wheel pulses inside of the range -650 to 650 are then converted to positive or negative characters if the wheel is clockwise of center or counterclockwise of center, respectively. This is done to give the steering wheel effectively twice the range of values for greater resolution. The values are converted to the range of 35-126, and depending on which side of center the wheel is on, the other character is written to the lowest value, 35. If the reading is outside of the range, the code does nothing.

```
//Constrain steering range and convert to character range
    if(wheel_int >= -1.0f*wheel_range && wheel_int < wheel_range){
        if(wheel_int > 0) {
            wheel_pos_char = (wheel_int/7 + 35); //35-126
            wheel_neg_char = 35;
        }
        else {
            wheel_pos_char = 35;
            wheel_neg_char = (wheel_int/7 - 35)*-1; //35-126
        }
    }
    else{
    }
```

Due to the springs in the pedals not always returning the values to the same position, the values of the pedals were always capped to the range of 35 to 126 to make sure no bad transmissions occur.

```
//cap gas values to account for potentiometer inconsistency
    if(gas_int < 35){
        gas_int = 35;
    }

    if (brake_int < 35){
        brake_int = 35;
    }

    if(gas_int > 126){
        gas_int = 126;
    }

    if (brake_int > 126){
        brake_int = 126;
    }
```

For debugging, it is useful to print out the values that are being sent as characters so that you can see what might be going wrong with the transmissions. This is commented out in the code.

```
//Debugging

  //pc.printf("%d\t%d\t%d\t%d\n\r",wheel_pos_char,wheel_neg_char,gas_int,
  brake_int);
  //pc.printf("Enable: %d\n\r",enableTwo);
```

Once everything is setup, the input is convered to unsigned characters and sent over XBee as four characters with a start character of “!” and an ending character of a new line and carriage

return. If the write buffer is full already, the code does nothing, making the transmissions are not being overloaded.

```
//XBee Data Transmission
    if(XBee.writeable()){
        XBee.printf("!%c%c%c%c\n\r",(unsigned
        char)wheel_pos_char,(unsigned char)wheel_neg_char,(unsigned
        char)gas_int,(unsigned char)brake_int);
    }
    else{
    }
```

Two-way communications have not been able to be resolved, as enabling this bit of code constantly causes the communications to disconnect so the car runs off on it's own. We have narrowed down the problem to the scanf command on the console, and none of the workarounds we tried seemed to work. A switch was added then that enables or disables the functionality, which is checked first. A 20 character buffer is established to give plenty of room for the message being sent from the car. The first two characters are then sent to the last character in the range, 126, and the code checks to see if there is data in the buffer to be read. If this is readable and there are a full 3 characters to be read, the first two characters are assigned to characters in the code.

```
//XBee Data Reception
    //Assign a 3 character buffer
    if(enableTwo){
        char message[20];
        message[1] = 126;
        message[2] = 126;
        if(XBee.readable()){
            if(XBee.scanf("%3s",&message) == 1) {

                //Store Individual Characters
                encoder_left_char = message[1];
                encoder_right_char = message[2];
```

Once the messages are read, the car characters are combined with the characters from the user input and sent to the computer over the USB connection as 6 characters with a new line and a carriage return. If there are not three characters to be read or if the read buffer is empty, the code does nothing.

```
                //Send Characters
                pc.printf("%c%c%c%c%c%c\n\r",(unsigned
                 char)encoder_left_char,(unsigned
                 char)encoder_right_char,\
                 (unsigned char)wheel_pos_char,(unsigned
                 char)wheel_neg_char,(unsigned char)gas_int,(unsigned
                 char)brake_int);
            }//if scanf
        }//if readable
        else{
        }//else readable
    }
}//while true
```

```
}//main
```

C. Car Code Documentation

1. KL25Z Motor Shield Functions

Several functions from the TFC motor shield library were used in order to easily interface with the KL25Z motor shield MBED library.

```
#include "TFC.h"

/* Initialize Motors and HBridge */
void Init(){
    TFC_Init();
    TFC_SetMotorPWM(0,0);
    TFC_HBRIDGE_ENABLE;
}

/* Sets the PWM value for each motor.
*
*   @param MotorA    The PWM value for HBridgeA.
*     The value is normalized to the floating point range of -1.0 to +1.0.
*   @param MotorB    The PWM value for HBridgeB.
*     The value is normalized to the floating point range of -1.0 to +1.0.
*
*   -1.0 is 0% (Full Reverse on the H-Bridge) and 1.0 is 100%
*     (Full Forward on the H-Bridge)
*/
void SetMotorPWM(float MotorA ,float MotorB){
    TFC_SetMotorPWM(MotorA, MotorB);
}

/* Sets the servo channels
*
*   @param ServoNumber  Which servo channel on the FRDM-TFC to use (0 or 1).
*     0 is the default channel for steering.
*   @param Position     Angle setting for servo in a normalized (-1.0 to 1.0)
*     form.
*/
void SetServo(int ServoNumber, float Position){
    TFC_SetServo(ServoNumber, Position);
}
```

2. Code Selection and Modifiable Parameters

DIP Switch Settings										
Switch 1	Switch 2	Switch 3	Switch 4	Binary	Kc	Ti	Camera Scale	Steering Scale	Max Speed	Setting
0	0	0	0	0	-	-	0.25	0.33	-	Adjustable Max Speed Cap
1	0	0	0	1	3	1	0.5	0.5	1.0	PID Optimal Straight
1	1	0	0	3	3	1	0.5	0.5	0.6	PID Optimal Turning
1	1	1	0	7	-	-	0.25	0.5	-	Open Loop
1	1	1	1	15	1	1.07	0.5	0.5	1.0	PID Old Straight
0	1	1	1	14	1	1.07	0.25	0.33	0.6	PID Old Turning
Any Other Combination				-	-	-	0.25	0.33	-	Adjustable Max Speed Cap No Transmit

For the car code, the KL25Z DIP switch was employed to allow the user to easily switch between a PID enabled car functionality and a straight user input to car output setup. Following some testing, it was determined that for the PID enabled version, the optimum amount of turning for both the camera and steering wheels was around 30 degrees to aptly demonstrate the application of a controls algorithm at the desired speeds, which corresponds to a scaling factor of 0.5. In the straight input setup, the camera only turns half this distance and the wheels only turn two thirds as much. Extended turning of the wheels at higher speeds tends to make the car handle too quickly for the average user, and the camera turning angle tends to make users overshoot corrections upon coming out of corners.

```
//DIP Switch Configuration
switch_setting = TFC_GetDIP_Switch();
if (switch_setting == 0){ //PID Control and Open Loop Disabled
      PID_Enable = false;
      Open_Loop_Enable = false;
      camera_scale = 0.25f;
      steering_scale = 0.33f;
}
else if (switch_setting == 1){ //PID Control Enabled, Open Loop Disabled
      PID_Enable = true;
      Open_Loop_Enable = false;
      camera_scale = 0.5f;
      steering_scale = 0.5f;
      leftController.setTunings (3.0f, 1.0f, 0.0f); //Kc, tauI, tauD (floats)
      rightController.setTunings (3.0f, 1.0f, 0.0f); //Kc, tauI, tauD
      (floats)
      max_speed = 1.0f; //(m/s) used for PID output limits and setpoint
      calculations
      initializePidControllers(); //PID Initialization
}
else if (switch_setting == 3){ //PID Control Enabled, Open Loop Disabled
      PID_Enable = true;
      Open_Loop_Enable = false;
      camera_scale = 0.5f;
      steering_scale = 0.5f;
      leftController.setTunings (3.0f, 1.0f, 0.0f); //Kc, tauI, tauD (floats)
```

```
        rightController.setTunings (3.0f, 1.0f, 0.0f); //Kc, tauI, tauD
        (floats)
        max_speed = 0.6f; //(m/s) used for PID output limits and setpoint
        calculations
        initializePidControllers(); //PID Initialization
}
else if (switch_setting == 7){ //PID Control Disabled, Open Loop Enabled
        PID_Enable = false;
        Open_Loop_Enable = true;
        camera_scale = 0.25f;
        steering_scale = 0.5f;
}
else if (switch_setting == 15){ //PID Control Enabled, Open Loop Disabled
        PID_Enable = true;
        Open_Loop_Enable = false;
        camera_scale = 0.5f;
        steering_scale = 0.5f;
        leftController.setTunings (1.0f, 1.07f, 0.0f); //Kc, tauI, tauD
        (floats)
        rightController.setTunings (1.0f, 1.07f, 0.0f); //Kc, tauI, tauD
        (floats)
        max_speed = 1.0f; //(m/s) used for PID output limits and setpoint
        calculations
        initializePidControllers(); //PID Initialization
}
else if (switch_setting == 14){ //PID Control Enabled, Open Loop Disabled
        PID_Enable = true;
        Open_Loop_Enable = false;
        camera_scale = 0.5f;
        steering_scale = 0.5f;
        leftController.setTunings (1.0f, 1.07f, 0.0f); //Kc, tauI, tauD
        (floats)
        rightController.setTunings (1.0f, 1.07f, 0.0f); //Kc, tauI, tauD
        (floats)
        max_speed = 0.6f; //(m/s) used for PID output limits and setpoint
        calculations
        initializePidControllers(); //PID Initialization
}
else{ //PID Control and Open Loop Disabled
        PID_Enable = false;
        Open_Loop_Enable = false;
        camera_scale = 0.25f;
        steering_scale = 0.33f;
}
```

For the functionality of the car, the only major control parameter is the cap_default. This can be used to cap the speed PWM of the car directly, which can be any value between 0.6 and 1.0 to help draw data curves for establishing the deadzone. With the value limited, you simply run the car in a straight line at full speed and record the speed of the car. We used a timing gate to measure this, but the encoders are also a valid option.

```
//Variable Parameters for Performance
float cap_default = 1.0f; //(PWM) adjustable for no PID output capping
```

3. **Initialization and Declaration**

The libraries included correspond to those seen in the UML diagram given previously. Each will be discussed as it is used in the code.

```
//Libraries
#include "rtos.h"
#include "mbed.h"
#include "TFC.h"
#include "PID.h"
```

The PID rate set for our car is 0.05, or 20 updates per second, as this was determined to be adequate. The PID constants were determined using the characterizations seen in the Controls Section. Due to the fact that both motors showed essentially the same performance, the same values were used for both left and right PID controllers. The max_speed variable is declared so it can be modified later on in the code.

```
//PID Information
#define RATE  0.05
#define Kc    1.0
#define Ti    1.0
#define Td    0.0
PID leftController(Kc, Ti, Td, RATE);
PID rightController(Kc, Ti, Td, RATE);
float max_speed; //(m/s) adjustable for PID speed calculations
```

The calculation parameters were initialized as floats as storage space is not currently a limitation. Only the gas and wheel floats need an initialization as all others are calculated from these quantities. The Booleans are used for enabling and disabling PID and open loop controls later in the code.

```
//CALC Information
bool PID_Enable;
bool Open_Loop_Enable;
float gas_float = 0 ;
float wheel_float = 0;
float steering_servo_float;
float camera_servo_float;
float steerplus;
float left_wheel_speed;
float right_wheel_speed;
float out_speed;
float in_speed;
float left_target_speed;
float right_target_speed;
```

The hardware information declares the variables used by the switches, potentiometers, servos, and battery level reader. The steering and camera servo center defaults are declared separately so that they can be modifed in the future without issue.

```
//HARDWARE Information
int switch_setting;
int steering_pot = 0;
int cap_pot = 1;
```

```
int servo_channel = 0;
int camera_channel = 1;
float steering_center_default = 0.25f; //adjustable center steering servo
position value
float camera_center_default = 0.25f; //adjustable center camera servo
position value
float camera_scale;
float steering_scale;
float steering_pot_value;
float cap_pot_value;
float battery_volt;
float steering_center;
float cap;
```

The serial communication from the XBee is initialized with TX on PTD3 and RX on PTD2. The pc serial can also be enabled for debugging, but this interface cannot be used at the same time that the board is powered with the battery, so debugging is limited on the car currently. The baud rate variable was added for quick modification.

The unsigned characters correspond to these values read in from the XBee to be converted to user input. The encoder left and right integers are used to send data back over the XBees to the console.

```
//SERIAL Information
Serial XBee(PTD3, PTD2);
int baud_rate = 9600;
unsigned char wheel_pos_char;
unsigned char wheel_neg_char;
unsigned char gas_char;
unsigned char brake_char;
int encoder_left_int;
int encoder_right_int;
```

The encoder signals are read as interrupts on pins PTD0 for left and PTA13 for right. These pins look for high states of over 2.0V and low states of below 0.8V. The timers are used to measure pulsewidths between consecutive falling edges caused by one set of teeth passing each optical switch. The timeout value is used to limit the pulsewidths read by the encoder so that the code can determine when the car is not moving, rather than how long the last pulsewidth read was. The min_pw and max_pw were added for transmission purposes and to deal with some of the noise issues from the h-bridges. Values faster than 5 ms as impossible for our car given out limitations, and values slower than 96 ms basically mean the car is moving slow enough to consider it not moving. The pulsewidth values for each side are written to this maximum value at the start, and a previous value variable is declared for each encoder to store the last received value.

```
//INTERRUPT Information
InterruptIn encoder_left(PTD0);
InterruptIn encoder_right(PTA13);
Timer timer_left, timer_right;
float time_out = 150.0f; //(ms) between pulses too long for encoders to be
detecting a speed
float min_pw = 5.0f; //(ms) minimum pulsewidth that should be considered as a
possible speed
```

```
float max_pw = 96.0f; //(ms)maximum pulsewidth that is possible for
transmission
float pulsewidth_left_val = max_pw;
float pulsewidth_right_val = max_pw;
float pulsewidth_left_val_prev = pulsewidth_left_val;
float pulsewidth_right_val_prev = pulsewidth_right_val;
```

The PID setup only needs the input limits and output limits specified as well as auto mode for each controller. The max speed variable (in meters per second) is employed for quick modification of the inputs, while the outputs are fixed at 0.0 (min) to 1.0 (max) to correspond to the same range of PWM values that can be written to the motors. The PID controller works on scaling the inputs and working in percentages, so attention to units in these assignments determines the functionality of the controller.

```
//PID Initialization
void initializePidControllers(void){
    leftController.setInputLimits(0.0, max_speed); //(m/s) travel speed
    leftController.setOutputLimits(0.0, 1.0); //working range of car
    leftController.setMode(AUTO_MODE);
    rightController.setInputLimits(0.0, max_speed); //(m/s) travel speed
    rightController.setOutputLimits(0.0, 1.0); //working range of car
    rightController.setMode(AUTO_MODE);
}
```

4. **Encoder Wheel Speed**

The MBED RTOS library was used to provide multithreading support to the car program. Multithreading was needed in order to handle the interrupts from the wheel encoders, which have separate interrupt service routines (ISR's) and codes as well.

For each main interrupt thread, the timer is first reset (to 0) and started when the thread is first called in the main loop. The mode of the interrupt pin is then called as a "PullNone," which means that it does not use the internal pullup resistor of the KL25Z board for reading input. The KL25Z board does not have an internal pulldown resistors, so this step is critical as external pullup resistors are used in the encoder circuit on the XBee protoboard that correctly limit the ranges of the output to trip the encoder. The interrupt is then attached to the corresponding ISR using falling edges as triggers.

```
//INTERRUPT Code
    //LEFT Main Thread
    void encoder_thread_left(void const *args){
        timer_left.reset();
        timer_left.start();
        encoder_left.mode(PullNone);
        encoder_left.fall(&ts_left);
    }

    //RIGHT Main Thread
    void encoder_thread_right(void const *args){
        timer_right.reset();
        timer_right.start();
        encoder_right.mode(PullNone);
        encoder_right.fall(&ts_right);
    }
```

On trigger, the ISR's use the timer read function in milliseconds (as integers) to read the elapsed time since the last falling edge. The time is reset for timing the next interrupt. These ISR's are kept short to reduce processing time on triggering.

```
//LEFT ISR
void ts_left(){
      pulsewidth_left_val = timer_left.read_ms();
      timer_left.reset();
}

//RIGHT ISR
void ts_right(){
      pulsewidth_right_val = timer_right.read_ms();
      timer_right.reset();
}
```

5. **Main Loop**

The main loop of the code first sets up the XBee communications, initializes and enables the needed TFC library components for the motors, sets the h-bridge PWM to 9000 Hz (to eliminate the buzzing noise), and sets the motors to 0 so the car does not move.

```
//Main Loop
int main(){
    //Initialize Serial Comms
        XBee.baud(baud_rate);

    //TFC Setup
        TFC_Init();
        TFC_InitMotorPWM(9000); //switching frequency of H-Bridges that
         yields no audible noise (1000-9000 Hz)
        TFC_SetMotorPWM(0,0); //make sure motors are off
        TFC_HBRIDGE_ENABLE;
```

The DIP switch setting is then read as noted in the previous section for selecting the car code. The encoder threads are then initialized.

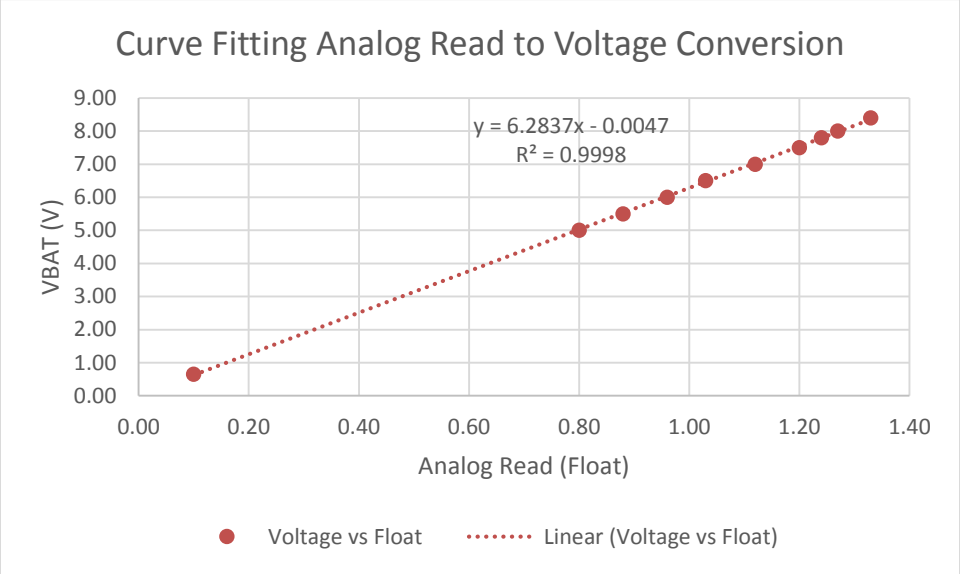
```
    //Thread Setup
        Thread thread_l(encoder_thread_left);
        Thread thread_r(encoder_thread_right);
```

A buffer (message) size of twenty characters is used for the code as that gives plenty of room for the message for the console. The message is only four characters and a new line character being sent from the console. The rest of the loop runs indefinitely on a while (true) loop.

```
    char message[20];
    while(true){
```

Before running any of the serial-based commands, the code checks the TFC hardware for potentiometer settngs and battery voltage, converting and storing these values for later use. The servo adjustment gives about 5 degrees of change on either side of the zero for fine turning of the steering servo. The speed capping servo limits the speed down from 1.0 PWM to 0.5 PWM at the minimum to prevent people from driving too fast during the demonstrations.

To calculate the battery voltage thresholds, the TFC shield and KL25Z board were powered by a variable power supply and the associated readings were recorded and plotted in the chart below.



Based on these values the different voltage thresholds for the battery level indicators were set in the code. The car simply reads the battery voltage and lights up from 0 to 4 LEDs on the voltage range of 6 to 8.4V.

```
//Hardware Adjustments
        steering_pot_value = TFC_ReadPot(steering_pot); //(-1 to 1) float
        potentiometer reading
        cap_pot_value = TFC_ReadPot(cap_pot); //(-1 to 1) float potentiometer
        reading
        steering_center = steering_pot_value*0.05f+steering_center_default;
        //(0.2 to 0.3 PWM) servo center range adjustment
        cap = cap_default -(cap_pot_value+1.0f)*0.25f;  //(0.5 to 1.0 PWM)
        speed cap range adjustment

        battery_volt = TFC_ReadBatteryVoltage()*6.28f; //(V) voltage of
        battery
        if (battery_volt >= 7.8f){ //voltage cutoff threshold for 4 LEDS
            TFC_SetBatteryLED_Level(4);
        }
        else if(battery_volt >= 7.2f){ //voltage cutoff threshold for 3 LEDS
            TFC_SetBatteryLED_Level(3);
        }
        else if(battery_volt >= 6.6f){ //voltage cutoff threshold for 2 LEDS
            TFC_SetBatteryLED_Level(2);
        }
        else if(battery_volt >= 6.0f){ //voltage cutoff threshold for 1 LEDS
            TFC_SetBatteryLED_Level(1);
        }
        else{ //voltage cutoff threshold for 0 LEDS
            TFC_SetBatteryLED_Level(0);
```

```
        }
```

To deal with issues encountered when data transmission is lost (holding onto the last received command and having the car take off), a readable command was employed to first check and see if there is any data in the buffer before trying to perform any operations. It also checks to make sure that the length of the message is 5 characters so that no short packets are being read.

If there is no data in the buffer, an else condition at the end of the code writes both motors to stop moving and sets the servos back to straight, stopping the car instead of allowing it to take off. This still does not always work and should be looked into for future work.

```
    //XBee Data Reception
        if(XBee.readable() && XBee.scanf("%5s",&message) == 1){
```

The message data is then parsed into individual characters for steering wheel, gas pedal, and brake pedal inputs.

```
        //Store Individual Characters
            wheel_pos_char = message[1];
            wheel_neg_char = message[2];
            gas_char = message[3];
            brake_char = message[4];
```

The steering data employs both a positive and negative character to extend the resolution of the steering wheel range to 186 characters. If the encoder is clockwise of center, the negative character gets written to the lowest value (35) and the wheel float value is based on the converted positive character, less the lowest value, and scaled from 0 to 1. A similar operation is employed for counterclockwise of center.

```
//STEERING Calculations
            if((int)wheel_neg_char == 35) {
                wheel_float = ((int)wheel_pos_char - 35)/-92.0f;
            }

            else {
                wheel_float = ((int)wheel_neg_char - 35)/92.0f;
            }
```

The “steerplus” variable is then calculated, which converts the wheel float to an absolute value of the radian measurement of the angle of the wheels. If this value is near 0, it writes the variable to a value of 0.001 to be small, constant, and non-zero for further calculations.

```
            steerplus = wheel_float*0.611f; //If steer is zero, make it
            something small [rad]
            if (steerplus < 0){
                steerplus = steerplus*-1.0f;
            }

            if ( steerplus < 0.001 ) {
                steerplus = 0.001;
            }
```

Steering and camera servo PWM values are then calculated using the wheel float using the scaling factors declared in the initialization (different for each running scenario). The centering value declared in the initialization is added to each calculation as this is the PWM corresponding to the center position of each servo with our current setup. This value may vary depending on configuration of the servo horn on the servo as well as the PWM being sent by the specific microcontroller being used.

```
steering_servo_float = wheel_float * steering_scale +
steering_center; //-30 degrees to 30 degrees, depending on
scaling
camera_servo_float = wheel_float * camera_scale +
camera_center_default;  //-30 degrees to 30 degrees, depending on
scaling
```

As a final check, the servo values are only written to the corresponding servo if the values are within the range of -1 to 1 to prevent damaging the servo motors.

```
if(steering_servo_float > -1.0f && steering_servo_float < 1.0f &&
camera_servo_float > -1.0f && camera_servo_float < 1.0f){
    TFC_SetServo(servo_channel,steering_servo_float);
    TFC_SetServo(camera_channel,camera_servo_float);
}
```

To prevent reading random values repeatedly from the encoders when the car is stopped corresponding to the last pulsewidth seen, a zeroing function is used on each encoder. This function reads the time since the last interrupt and sets the value to the maximum pulsewidth (126) if it has been more than 150 ms, resetting the timer again afterwards.

```
//Encoder Data Zeroing
    if (timer_left.read_ms() >= time_out){
        pulsewidth_left_val = max_pw;
        timer_left.reset();
    }
    if (timer_right.read_ms() >= time_out){
        pulsewidth_right_val = max_pw;
        timer_right.reset();
    }
```

Similarly, the median pulsewidths are then capped to prevent any impossible values from making it through which correspond to noise, as noted in the initialization. For very short pulsewidths (noise data), the pulsewidth value is written to the previous good value so that that data stream is slightly cleaner to look at.

```
//Encoder Data Capping
    if (pulsewidth_left_val >= max_pw){
        pulsewidth_left_val = max_pw;
    }
    else if (pulsewidth_left_val <= min_pw){
        pulsewidth_left_val = pulsewidth_left_val_prev;
    }
    else{
    }
```

```
    if (pulsewidth_right_val >= max_pw){
        pulsewidth_right_val = max_pw;
    }
    else if (pulsewidth_right_val <= min_pw){
        pulsewidth_right_val = pulsewidth_right_val_prev;
    }
    else{
    }
```

For the transmission of the data over the XBee to the console, the encoder values are translated by 30 to always be in a safe range for character transfer.

```
//XBee Data Transmission
    encoder_left_int = pulsewidth_left_val + 30;
    encoder_right_int = pulsewidth_right_val + 30;
```

We now have to send the encoder data back to the console. Similarly, this uses a writeable command that first checks to make sure the buffer on the console is not already filled with data. If it is, the car does not send the data at all. If the buffer is empty, the character encoder pulsewidths are then transmitted as a string with a new line character. Having this in line with the code seems to work the best based on our testing.

```
//XBee Data Transmission
    if (XBee.writeable()){
        XBee.printf("%c%c\n",(unsigned
        char)encoder_left_int,(unsigned char)encoder_right_int);
    }
    else{
    }
```

Finally, the wheel speeds seen are written to 0 if the max pulsewidth is currently being seen. For any other values the pulsewidth is converted to m/s using the formula developed in the Physics Section based on the geometry of the encoder.

```
//ENCODER Conversions
    if (pulsewidth_left_med == max_pw){
        left_wheel_speed = 0;
    }
    else {
        left_wheel_speed = 16.83/pulsewidth_left_med;
    }

    if (pulsewidth_right_med == max_pw){
        right_wheel_speed = 0;
    }
    else {
        right_wheel_speed = 16.83/pulsewidth_right_med;
    }
```

The gas float value (0 to 1) is established by taking the relative difference between the gas and brake characters. This was a simple coding solution that allows for no output to the motors when both pedals are being pressed (as a safety).

During testing with the car, a deadzone for the gas pedal was required. This was caused by the spring used to return the pedal to the zero position being too weak, which sometimes lead to static input values without the user even touching the pedal. This was accounted for in the coding through testing by ignoring inputs below a certain limit (0.25). The modification was done here as it is most effective to account for after all other data conversions have been completed.

```
//THROTTLE Calculations
    gas_float = ((int)gas_char)/92.0f - ((int)brake_char)/92.0f;

    if (gas_float <= 0.25 && gas_float >= -0.25){ //Pedal spring does
    not always bring pedal back to zero
        gas_float = 0;
    }
```

Now the actual PID loop is employed. For gas floats great than 0 (not stopped or going in reverse), if the PID is enabled in the initialization, the outer wheel speed and inner wheel speed (in ft/s) are calculated using the formulas determined in the Controls Section.

```
    if (gas_float > 0){ //going forward, implement PID control

        if (PID_Enable == true){
        //PID Calculations
            out_speed = max_speed*gas_float*(1 +
            0.353*tan(steerplus)); //tangential speed of outer side
            of car [m/s]
            in_speed = max_speed*gas_float*(1 -
            0.353*tan(steerplus)); //tangential speed of inner side
            of car [m/s]
```

Based on the wheel position, if the car is making a left or right turn is then established, which establishes the assignment of outer and inner wheel speed to either left or right wheel as the set point to be achieved by the car using the control system.

```
            if (wheel_float < 0){ //turning left
                leftController.setSetPoint(in_speed);
                rightController.setSetPoint(out_speed);
            }
            else{ //turning right
                leftController.setSetPoint(out_speed);
                rightController.setSetPoint(in_speed);
            }
```

The process values (what feedback the system is seeing) is then set as the encoder speeds calculated earlier, directly this time as the wheel designation corresponds directly to the controller with the same name.

```
            leftController.setProcessValue(left_wheel_speed);
            //encoder value
            rightController.setProcessValue(right_wheel_speed);
            //encoder value
```

The PID library compute function calculates the target output PWM for the car based on the PID algorithm, set points, process values, and input / output limits.

```
            left_target_speed = leftController.compute();
            right_target_speed = rightController.compute();
```

Due to limitation in the motor outputs due to noise, the computed output PWM's are then limited to 0.75 PWM. This highest value is also scaled back from a 1.0 PWM to allow the outer wheel to spin at its max speed (at around 1.0 PWM) when the car centerline speed is less than 1.0 PWM (basically a geometry limitation). Around 1.0 PWM the car experiences excessive noise in the data, which we want to avoid.

```
            left_target_speed = left_target_speed*0.75f; //(0.0 -
            0.75 PWM) range for max speed noise issue
            right_target_speed = right_target_speed*0.75f; //(0.0 -
            0.75 PWM) range for max speed noise issue
        }
```

If open loop control is enabled, the car basically takes the wheel speed calculations from the PID loop and directly writes those values to the inside and outside wheels based on the steering angle. This does not use any encoder feedback, hence the open loop control.

```
        else if (Open_Loop_Enable == true){
            out_speed = 0.8f*gas_float*(1 + 0.353*tan(steerplus));
            //outside wheel PWM maximum less than 1.0
            in_speed = 0.8f*gas_float*(1 - 0.353*tan(steerplus));
            //inside wheel PWM scaled the same as the outside wheel

            if (wheel_float > 0){ //turning left
                left_target_speed = in_speed;
                right_target_speed = out_speed;
            }
            else{ //turning right
                left_target_speed = out_speed;
                right_target_speed = in_speed;
            }
        }
```

If the PID is not enabled and the gas float is greater than zero, we want to map the user throttle input directly to the motor PWM for both wheels in the range of 0.5 to 1.0 PWM. A cap is then implemented to limit user speed if necessary. This can also be helpful for testing, as multiple fixed data points can be taken for max speed at a given PWM for curve fitting and establishing the motor deadzone.

```
        else{
        //Running with capped speed and no PID
            left_target_speed = gas_float/2+0.5f; //0.5 - ~1.0 range
            for max speed issue
            if (left_target_speed > cap){ //cap speed value
                left_target_speed = cap;
            }
            right_target_speed = left_target_speed;
        }
    }
```

If the gas float is zero in either case, set all speeds to zero so the car does not move.

```
            else if (gas_float == 0){ //stopped, write motors to not move
                out_speed = 0;
                in_speed = 0;
                left_target_speed = 0;
                right_target_speed = 0;
            }
```

Similarly to the PID disabled case above, if the gas float is less than zero in either case, write the user input directly to both wheels in the range of 0.5 to 0.75. This is limited to less than one because people tend to drive too quickly in reverse and run into objects.

```
            else { //going in reverse, just use the gas_float, scaled
                left_target_speed = gas_float/4-0.5f; //0.5 - 0.75 speed ok
                for reverse
                right_target_speed = left_target_speed;
            }
```

Finally, in one write the motor speeds are actually written to the motor encoders. Here there is also some debugging code for printing the values back to a putty terminal window.

```
            //Writing Motor Speeds and Debugging
                TFC_SetMotorPWM(right_target_speed, left_target_speed);

                //pc.printf("%c\t%c\t%c\t%c\t%.2f\t%.2f\t%.2f\n\r",wheel_pos_c
                har,wheel_neg_char,gas_char,brake_char,gas_float,wheel_float,s
                ervo_float);
                //pc.printf("%.2f\t%.2f\t%.2f\t%.2f\t%.2f\t%.2f\t%.2f\t%.2f\n\
                r",pulsewidth_left_med,pulsewidth_right_med,left_wheel_speed,r
                ight_wheel_speed,out_speed,in_speed,left_target_speed,right_ta
                rget_speed);
                //pc.printf("%.2f\t%.2f\t%.2f\t%.2f\t%.2f\t%.2f\t%.2f\n\r",lef
                t_wheel_speed,right_wheel_speed,gas_float,out_speed,in_speed,l
                eft_target_speed,right_target_speed);
        }//if readable
```

Mentioned above, if the car loses data reception, the car needs to be written to stop so that it does not take off with the last command given to it.

```
        else{//if data lost, stop car movement completely
            TFC_SetMotorPWM(0,0);
            TFC_SetServo(servo_channel,0.26f);
            TFC_SetServo(camera_channel,0.26f);
        }//else readable
    }//while loop
}//main loop
```

VIII. Testing

A. Uploading Code

There are separate codes that have been developed in MBED for the console and car which can be uploaded to the KL25Z boards on the car and console using the direction found in Section VII. These codes are also documented in Section VII so that changes can be made for future modifications.

B. Logging Data

1. Serial Data

The design of the car and console code is to transmit data back to a computer as serial over a USB cable connected to the Open SDA port of the console KL25Z. This data comes in the form of six characters followed by a new line and a carriage return that represent the following quantities.

String Character	Value	Quantity	Range	Units
1	Carriage Return	N/A	N/A	nd
2	Encoder Left Char	Left Encoder Pulsewidth	35-126	ms
3	Encoder Right Char	Right Encoder Pulsewidth	35-126	ms
4	Wheel Pos Char	Scaled Wheel Displacement CW	35-126	nd
5	Wheel Neg Char	Scaled Wheel Displacement CCW	35-126	nd
6	Gas Int	Scaled Gas Pedal Displacement	35-126	nd
7	Brake Int	Scaled Brake Pedal Displacement	35-126	nd
8	New Line	N/A	N/A	nd

Characters two and three come directly from the car over XBee, recorded as the pulsewidths seen between encoder falling edges in milliseconds. Characters four through seven are the same values as those sent to the car from the console based on user input from the steering wheel, gas pedal, and brake pedal. The first and last character are for data formatting. The character ranges of 35-126 were established based on the unsigned character limitation with normal ASCII characters. The minimum of 35 was established to not try and use any bad transmission characters, (new line, carriage return, etc.) along with the quote character (34), since this is used in VBA for reading strings.

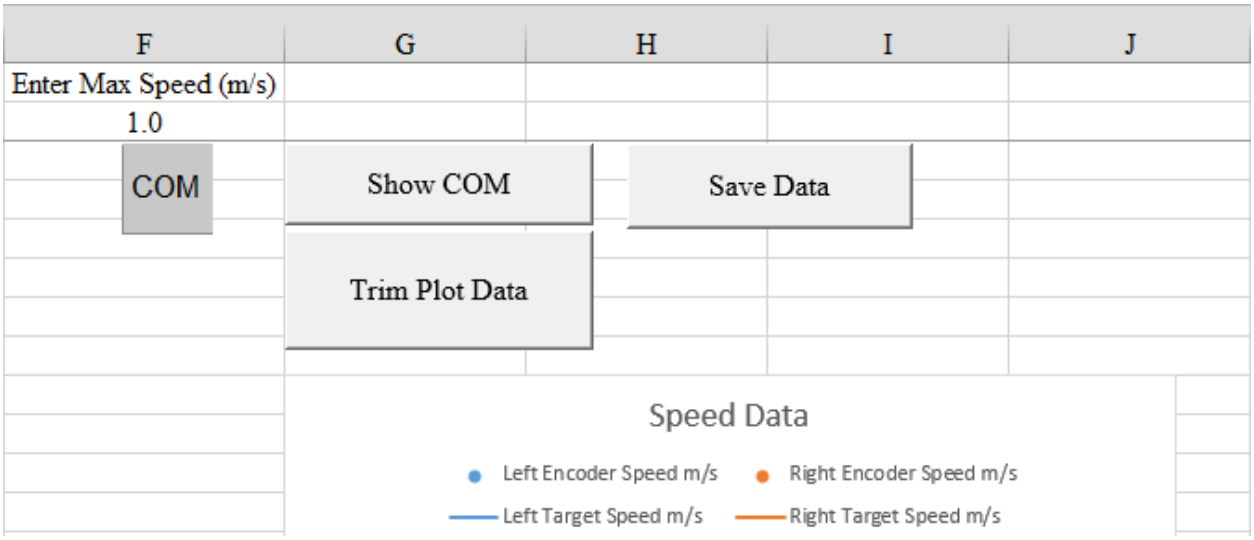
2. Using the Excel Sheet

The following documentation is for the 2013 version of Excel. Previous versions still support the ActiveX control and main VBA functionality, but the macros do not work in previous versions as VBA has changed its syntax for chart formatting. A working version for 2010 can also be found on EDGE, which effectively just has the same main macro and slightly modified plotting macros.

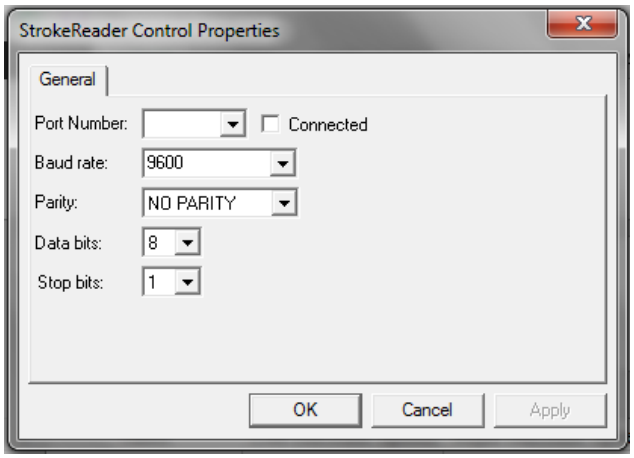
In essence, the strategy is to give the computer the same raw data that the car is seeing before it is converted to useable quantities to better understand how the car is functioning and to make data transmission easier. To log data, an Excel VBA code was developed that relies on the StrokeReader ActiveX control for Excel. This control is a serial port interface with event-driven, asynchronous data transfer that can be downloaded here. Due to budgetary limitations we area only using the free version, but a license only costs $7 so it might be worth the investment for

future development. This interface was developed for scanning barcodes, but it provides a convenient interface for all serial data communications. The basic background VBA macro associated with this control was modified extensively for our application in the specific Excel sheet being used. For additional information about including this control in other applications a good tutorial can be found here.

In the macro enabled Excel sheet that is included in the Final Documents Subdirectory on EDGE, the interface you will see on the right-hand side of the screen is as seen below. The small ActiveX control box labeled "COM" will not be visible, but this is made visible by clicking the "Show COM" button. The "Enter Max Speed (m/s)" box (F2) is used as the input for the coding as it is in the car code. Normal values will be either 1.0 for straight runs and 0.6 for turns, depending on the DIP switch setting.



If you double click the "COM" ActiveX control, you will get a window that looks like the image below. This is used to record the data directly. The port number for data transmission will vary from computer to computer, but it can be found by typing "Device Manager" into the Windows search bar. Under the "Ports" heading you should see the Freescale board with the associated port number listed when the device is connected. The rest of the settings stay the same as below.



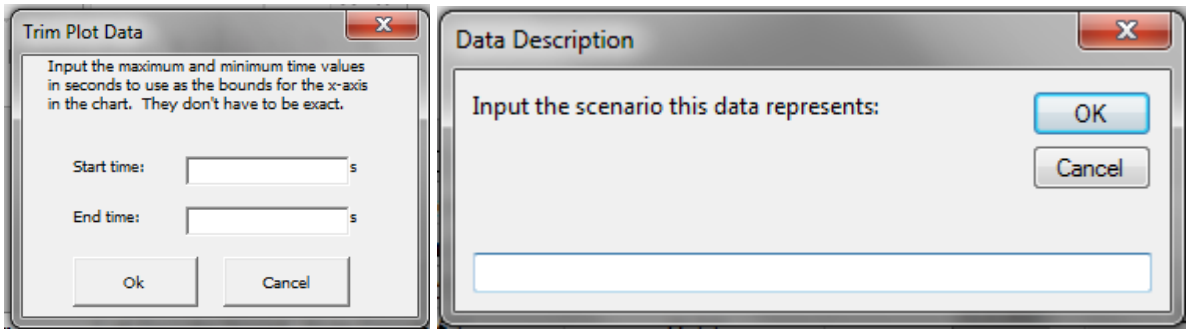
To take data, the console has to be powered and the USB has to be connected. The car also has to be on. First press the reset on the car KL25Z, then press the reset on the console KL25Z. The car XBee LEDs for TX and RX should light up and blink quickly (one blue, one red), and the RSSI LED should be steady red (meaning signal strength is strong). On the console, the red RSSI and blue connection LEDs should be on. If this does not work, first check to make sure that the wiring harnesses in the console on the perfboard are well seated in the pin headers, as this can cause disconnection.

To check if data is being read correctly on the port you are using, you can first open a Putty window, chose the Serial connection type, enter the COM number of the port, and hit open. You should see a stream of data characters being printed with carriage returns. If the data ever stops, hit the reset button on the console and it should begin again. Again, this is signaled by the blinking LEDs on the car XBee.

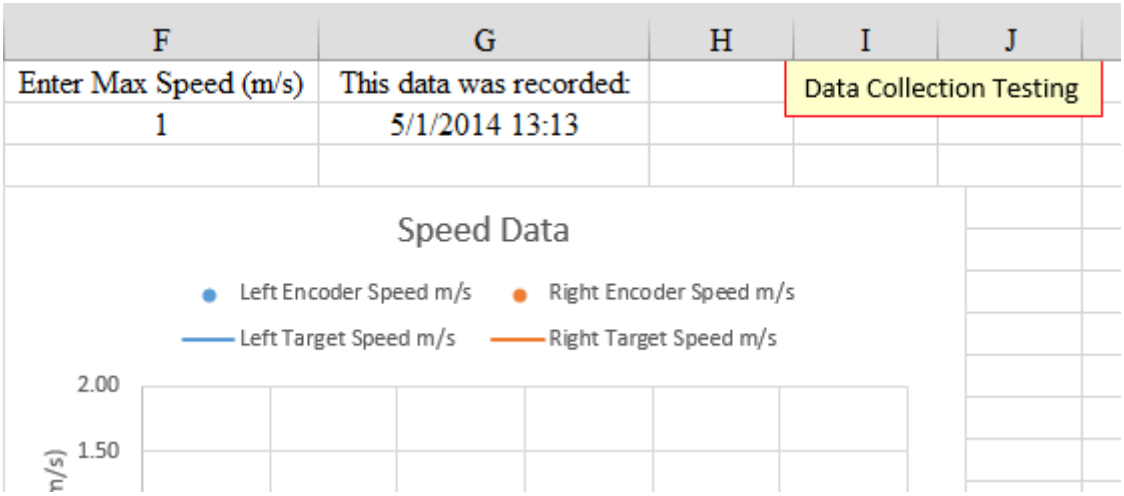
Once data is being read correctly, all that needs to be done in Excel is checking the "Connect" box in the ActiveX control window as above and clicking "Apply." You should see data start to be displayed in the Excel document as below, even without clicking "Ok" in the ActiveX control. To stop data collection, simply uncheck the Connect box and click Apply again.

	A	B	C	D	E
1	Timer	Left Encoder Speed	Right Encoder Speed	Left Target Speed	Right Target Speed
2	s	m/s	m/s	m/s	m/s
3	0.016	0.58	0.84	0.53	0.33
4	0.109	1.87	0.84	0.53	0.33
5	0.254	0.58	0.29	0.53	0.33
6	0.402	0.60	1.40	0.53	0.33
7	0.547	0.60	0.29	0.53	0.33
8	0.695	0.60	0.73	0.53	0.33
9	0.840	0.62	0.43	0.53	0.33
10	0.988	0.60	1.29	0.53	0.33
11	1.133	0.58	0.34	0.53	0.33

The way the VBA macro works in the background is that once you click "Ok" in the ActiveX control window following data collection, the program will scan the five data columns for data and plot these points with formatting in cell G9 of the active sheet. If you want to trim the data quickly you can hit the "Trim Plot Data" button above the graph and enter the start and end time of the data you want to look at in the window as seen below left.



If the data set is good, click the "Save Data" button and you will get a window as above right. Enter a quick name for that data set that records what was being done in the testing, for reference, and click "Ok." The data will be saved in a separate sheet, along with the plot and title label as seen below.



3. **VBA Program Documentation**

The VBA program is quite involved, but it can be modified by pressing Alt+F11 and selecting VBAProject(*File_Name*) >> Microsoft Excel Objects >> Sheet1 (COM). Additionally you can use the Developer Tab in Excel. If your Developer Tab is not visible, follow these directions. In the Developer Tab, click the "Design Mode" button and double click on the "COM" ActiveX control. This will open the code.

a) *Initialization*

The first part of the code is just initializing the variables in the code for completeness. VBA does not require this, but by uncommenting the top line the hard data type functionality will be enabled, which may help for checking with issues caused by modification.

The received characters include the buffer, cell printing, buffer length, and character string arrays used for data reception and parsing. The calculation integers are the limits used in the mbed code along with the integer conversions of the ASCII data characters. The calculation doubles are the same floats used in the mbed code for storing the values calculated from the physics and controls equations. The reference time value is initialized as public to enable functionality throughout the program cases.

```
'Option Explicit

'Received Characters
Dim buf As String 'Buffer for incoming serial data
Dim cell_idx As Integer 'The row number cell to store a received barcode
Dim LF As String
Dim data_chars As String
Dim encoder_left_char As String
Dim encoder_right_char As String
Dim wheel_pos_char As String
Dim wheel_neg_char As String
Dim gas_char As String
Dim brake_char As String

'Calculation Integers
Dim max_pw As Integer
Dim encoder_left_int As Integer
Dim encoder_right_int As Integer
Dim wheel_pos_int As Integer
Dim wheel_neg_int As Integer
Dim gas_int  As Integer
Dim brake_int As Integer
Dim pulsewidth_left_val As Integer
Dim pulsewidth_right_val As Integer

'Calculation Doubles
Public time_ref As Double 'global reference time reset when connection
checked
Dim max_speed As Double
Dim time_system As Double
Dim time_sample As Double
Dim left_wheel_speed As Double
Dim right_wheel_speed As Double
Dim wheel_float As Double
Dim gas_float As Double
Dim steerplus As Double
Dim out_speed As Double
Dim in_speed As Double
Dim left_target_speed As Double
Dim right_target_speed As Double
```

b) *Received Data*

The program starts at the Private Sub line, and the general structure is a set of cases that correspond to events caused by checking (connect) or unchecking (disconnect) the “Connect” box in the ActiveX control. When the connection is disconnected, you can see that the program simply runs the plot macro, which will be documented later. When the connection is connected, the program selects all of the existing data in columns A through E and clears the contents. It also initializes the cell row index for printing the first line of data, the reference time (Timer is an internal system clock call function in VBA that returns the number of seconds since midnight), the max speed constant, and the max pulsewidth constant. The last two values are the same as those initialized in the mbed code.

```
Private Sub StrokeReader1_CommEvent(ByVal Evt As StrokeReaderLib.Event, _
                                    ByVal data As Variant)
  Select Case Evt  'Can be EVT_DISCONNECT or EVT_DATA or EVT_SERIALEVENT
```

```
Case EVT_DISCONNECT 'if USB serial port adapter is just disconnected from
 the PC
    plot 'goes to plot sub
    Range("A1").FormulaR1C1 = "Timer"
    Range("A2").FormulaR1C1 = "s"
    Range("B1").FormulaR1C1 = "Left Encoder Speed"
    Range("B2").FormulaR1C1 = "m/s"
    Range("C1").FormulaR1C1 = "Right Encoder Speed"
    Range("C2").FormulaR1C1 = "m/s"
    Range("D1").FormulaR1C1 = "Left Target Speed"
    Range("D2").FormulaR1C1 = "m/s"
    Range("E1").FormulaR1C1 = "Right Target Speed"
    Range("E2").FormulaR1C1 = "m/s"
    Range("F1").FormulaR1C1 = "Enter Max Speed (m/s)"
    Range("A1").Select

Case EVT_CONNECT
    ActiveSheet.Range("a3",
     ActiveSheet.Range("a3").End(xlDown)).ClearContents
    ActiveSheet.Range("b3",
     ActiveSheet.Range("b3").End(xlDown)).ClearContents
    ActiveSheet.Range("c3",
     ActiveSheet.Range("c3").End(xlDown)).ClearContents
    ActiveSheet.Range("d3",
     ActiveSheet.Range("d3").End(xlDown)).ClearContents
    ActiveSheet.Range("e4",
     ActiveSheet.Range("e4").End(xlDown)).ClearContents
    cell_idx = 2 'start at row 3
    time_ref = Timer 'reset timer column
    max_speed = ActiveCell.Value 'm/s actual max speed
    max_pw = 96 'ms pulsewidth
```

When the program receives serial data, it then enters the “DATA” event and adds the received data to the buf string. The program than uses the “InStr” function to look through the buf string for the new line character (ASCII decimal 10), recording the length of the data to that character as an integer in LF. If less than 8 characters are received (the length of the whole string we are sending), it parses that data off of the buffer and exits the loop, and waiting for new data.

```
Case EVT_DATA  'Incoming serial data
    buf = buf + data 'Incoming serial data accumulated in buf

    Do
        'Read from Buffer
        LF = InStr(buf, Chr(10)) 'Scan buffer for ASCII <LF>
        If LF < 8 Then 'Waiting for full ASCII string to <LF> to be
        received
            buf = Right(buf, Len(buf) - LF) 'Cut the parsed data from the
            buffer
            Exit Do
        End If
```

The system time at the time of data reception is then recorded as time_system. The time_sample is then calculated as the time difference between this time and the reference time, effectively making a timer that starts at roughly 0 seconds for recording data sample times.

```
'Establish Time
time_system = Timer
time_sample = time_system - time_ref
```

Data parsing uses some of VBAs canned string functions. First, the Left function is used to copy the string of characters from the buffer up to and including the new line character from the left side of the string to the new string data_chars. The Mid function then parses off each character individually from the data_chars string according to the position in the string for 1 character length. Since the first character is the carriage return, this value is ignored and the data starts at the second position.

```
'Parse Data from Buffer and Remove
data_chars = Left(buf, LF)

encoder_left_char = Mid(data_chars, 2, 1) 'Parse data from
data_chars string starting at position 2 with 1 character length
encoder_right_char = Mid(data_chars, 3, 1)
wheel_pos_char = Mid(data_chars, 4, 1)
wheel_neg_char = Mid(data_chars, 5, 1)
gas_char = Mid(data_chars, 6, 1)
brake_char = Mid(data_chars, 7, 1)
```

Now that the data is stored in individual characters, the string just parses is removed from the buffer using the Right function (from the right side of the string) for the length of the total buffer less the distance to the new line character.

```
buf = Right(buf, Len(buf) - LF) 'Cut the parsed data from the
buffer
```

With the data removed, the characters are now converted back to the corresponding decimals from 35 to 126, as done in the mbed code.

```
'Convert Received Data
encoder_left_int = Asc(encoder_left_char) 'convert ASCII
characters to integers
encoder_right_int = Asc(encoder_right_char)
wheel_pos_int = Asc(wheel_pos_char)
wheel_neg_int = Asc(wheel_neg_char)
gas_int = Asc(gas_char)
brake_int = Asc(brake_char)
```

The pulsewidth values are calculated by removing 30 from the integer value of each data character, which is the opposite of the data transmission step in the car code.

```
'Convert Encoder Data
pulsewidth_left_val = encoder_left_int - 30 'convert to original
pulsewidths
pulsewidth_right_val = encoder_right_int - 30
```

The conversion from the pulsewidths to wheel speeds is identical to that seen in the car code in VBA format.

```
If pulsewidth_left_val = max_pw Then
    left_wheel_speed = 0
Else
    left_wheel_speed = 16.83 / pulsewidth_left_val
End If
If pulsewidth_right_val = max_pw Then
    right_wheel_speed = 0
Else
    right_wheel_speed = 16.83 / pulsewidth_right_val
End If
```

The conversion of steering data to wheel float and steerplus values is also identical to the car code for consistency.

```
'Convert Steering Data
If wheel_neg_int = 35 Then
    wheel_float = (wheel_pos_int - 35) / -92
Else
    wheel_float = (wheel_neg_int - 35) / 92
End If

steerplus = Abs(wheel_float * 0.611) 'If steer is zero, make it
something small [rad]

If steerplus < 0.001 Then
    steerplus = 0.001
End If
```

The conversion of gas and brake values to gas float and speed calculations used in the PID loop are also identical to the car code, with the exception that obviously the VBA code does not include the ability to calculate the PID output the same way as the car code works. This functionality may be a nice area to explore in future work, but for the time being we are only concerned in comparing the signal we are applying to the setpoint of the control loops (left and right target speeds as calculated by out speed and in speed assigned using the steering logic) with the wheel speed from the encoders that we are using as the process variable in the control loops. This allows us to see the square value inputs we are applying against the actual speed we are seeing from the encoders.

```
'Convert Throttle Data
gas_float = gas_int / 92 - brake_int / 92

If gas_float <= 0.25 And gas_float >= -0.25 Then 'Pedal spring
does not always bring pedal back to zero
    gas_float = 0
End If

If gas_float > 0 Then 'going forward, implement PID control

    'PID Calculations
    out_speed = max_speed * gas_float * (1 + 0.353 *
     Tan(steerplus)) 'tangential speed of outer side of car [ft/s]
    in_speed = max_speed * gas_float * (1 - 0.353 *
     Tan(steerplus)) 'tangential speed of inner side of car [ft/s]
```

```
                If wheel_float <= 0 Then 'turning left
                    left_target_speed = in_speed 'showing target max speed in
                    m/s
                    right_target_speed = out_speed
                Else 'turning right
                    left_target_speed = out_speed
                    right_target_speed = in_speed
                End If

            ElseIf gas_float = 0 Then 'stopped, write motors to not move
                out_speed = 0
                in_speed = 0
                left_target_speed = 0
                right_target_speed = 0
            Else 'going in reverse, just use the gas_float, scaled
                left_target_speed = gas_float / 4 - 0.5 '0.5 - 0.75 speed ok
                 for reverse
                right_target_speed = left_target_speed
            End If
```

Finally, the values calculated are written to individual cells using the Cell function (row, column) and incrementing the cell index each time. The placement of the cell index before the cell assignments is why the data starts printing in row three.

```
            'Write to Cells
            cell_idx = cell_idx + 1 'increment the row number of cell where
            the barcode will be stored
            Cells(cell_idx, 1) = time_sample
            Cells(cell_idx, 2) = left_wheel_speed
            Cells(cell_idx, 3) = right_wheel_speed
            Cells(cell_idx, 4) = left_target_speed
            Cells(cell_idx, 5) = right_target_speed
        Loop
  End Select
End Sub
```

c) *Plot Macro*

The macros used for plotting and recording data as not as critical to the basic functioning of the program, but they do make data collection and evaluation more efficient. Again, these macros only work for 2013 and should be modified for 2010 functionality. These macros were created using Excel's macro recording functionality, so they will only be minimally documented as it is very easy to record your own macros that do anything you want to the data sets created. These macros can be found in in the VBA interface under VBAProject(*File_Name*) >> Modules >> Module 1 and Module 2.

The plot function first deletes all charts in the main sheet and creates a new chart in cell G9.

```
Sub plot()
'deletes the old plot and creates a new one of all the data
    Dim chtObj As ChartObject
        For Each chtObj In ActiveSheet.ChartObjects
        chtObj.Delete
        Next
    Sheets("COM").Shapes.AddChart2(240, xlXYScatter).Select
```

```
    ActiveChart.SetSourceData Source:=Range("COM!$A$1:$E$425")    'repopulate
    chart
    With Selection
    .Top = Range("G9").Top                          'adjust top
    .Left = Range("G9").Left
    End With
```

For all charts in the active sheet, the following formats the plot with titles, colors, axis labels, legends, and data marker types. This section was recorded directly using the record functionality.

```
'*************** Format Chart ************************************
For Each chtObj In ActiveSheet.ChartObjects
chtObj.Activate
ActiveChart.ChartTitle.Select
    Selection.Format.TextFrame2.TextRange.Characters.Text = "Speed Data"
    With Selection.Format.TextFrame2.TextRange.Characters(1,
    10).ParagraphFormat
        .TextDirection = msoTextDirectionLeftToRight
        .Alignment = msoAlignCenter
    End With
    With Selection.Format.TextFrame2.TextRange.Characters(1, 5).Font
        .BaselineOffset = 0
        .Bold = msoFalse
        .NameComplexScript = "+mn-cs"
        .NameFarEast = "+mn-ea"
        .Fill.Visible = msoTrue
        .Fill.ForeColor.RGB = RGB(89, 89, 89)
        .Fill.Transparency = 0
        .Fill.Solid
        .Size = 14
        .Italic = msoFalse
        .Kerning = 12
        .Name = "+mn-lt"
        .UnderlineStyle = msoNoUnderline
        .Spacing = 0
        .Strike = msoNoStrike
    End With
    With Selection.Format.TextFrame2.TextRange.Characters(6, 5).Font
        .BaselineOffset = 0
        .Bold = msoFalse
        .NameComplexScript = "+mn-cs"
        .NameFarEast = "+mn-ea"
        .Fill.Visible = msoTrue
        .Fill.ForeColor.RGB = RGB(89, 89, 89)
        .Fill.Transparency = 0
        .Fill.Solid
        .Size = 14
        .Italic = msoFalse
        .Kerning = 12
        .Name = "+mn-lt"
        .UnderlineStyle = msoNoUnderline
        .Spacing = 0
        .Strike = msoNoStrike
    End With
    ActiveChart.SetElement (msoElementPrimaryCategoryAxisTitleAdjacentToAxis)
    Selection.Caption = "Time"
```

```
Selection.Format.TextFrame2.TextRange.Characters.Text = "Time (s)"
With Selection.Format.TextFrame2.TextRange.Characters(1,
8).ParagraphFormat
    .TextDirection = msoTextDirectionLeftToRight
    .Alignment = msoAlignCenter
End With
With Selection.Format.TextFrame2.TextRange.Characters(1, 8).Font
    .BaselineOffset = 0
    .Bold = msoFalse
    .NameComplexScript = "+mn-cs"
    .NameFarEast = "+mn-ea"
    .Fill.Visible = msoTrue
    .Fill.ForeColor.RGB = RGB(89, 89, 89)
    .Fill.Transparency = 0
    .Fill.Solid
    .Size = 10
    .Italic = msoFalse
    .Kerning = 12
    .Name = "+mn-lt"
    .UnderlineStyle = msoNoUnderline
    .Strike = msoNoStrike
End With

ActiveChart.Axes(xlValue).Select
ActiveChart.SetElement (msoElementPrimaryValueAxisTitleNone)
Selection.HasTitle = True
ActiveChart.Axes(xlValue, xlPrimary).AxisTitle.Text = "Speed (m/s)"
```

The following code can be modified to adjust y-axis limits quickly, as these are critical to data viewing and interpretation. Currently they are from 0 to 2 m/s, which completely encompasses our target speeds.

```
'Legend and y-axis limits
ActiveChart.SetElement (msoElementLegendTop)
ActiveChart.Axes(xlValue).MaximumScale = 2
ActiveChart.Axes(xlValue).MinimumScale = 0

'Change target speeds to lines, not points
ActiveChart.FullSeriesCollection(4).Select
With Selection.Format.Line
    .Visible = msoTrue
    .ForeColor.ObjectThemeColor = msoThemeColorAccent2
    .ForeColor.TintAndShade = 0
    .ForeColor.Brightness = 0
End With
Selection.MarkerStyle = -4142
ActiveChart.FullSeriesCollection(3).Select
With Selection.Format.Line
    .Visible = msoTrue
    .ForeColor.ObjectThemeColor = msoThemeColorAccent1
    .ForeColor.TintAndShade = 0
    .ForeColor.Brightness = 0
    .Transparency = 0
End With
Selection.MarkerStyle = -4142
```

```
Next
End Sub
```

d) *Trimming Data Macro*

Similar to the plot macro above, this macro is only used when trimming data and was recorded using Excel's macro functionality. It uses the User Form that can be found under VBAProject(*File_Name*) >> Forms >> TrimPlotUserForm and is displayed by clicking the button on the main sheet.

```
Sub TrimPlotData()
'trims the maximum and minimum time on the chart to focus on an area of
interest
TrimPlotUserForm.Show
End Sub
```

To access the code, you must open the above mentioned form, right click the form, and click "View Code." It is as seen below.

```
Private Sub CommandButton1_Click()

tstart = Val(TextBox1.Value)
tstop = Val(TextBox2.Value)

Dim chtObj As ChartObject
        For Each chtObj In ActiveSheet.ChartObjects
        chtObj.Select
        ActiveChart.Axes(xlCategory).MinimumScale = tstart
        ActiveChart.Axes(xlCategory).MaximumScale = tstop
        Next
Unload Me
End Sub

Private Sub CommandButton2_Click()
Unload Me
End Sub

Private Sub Label1_Click()

End Sub

Private Sub UserForm_Initialize()
'Empty NameTextBox
TextBox1.Value = ""

'Empty PhoneTextBox
TextBox2.Value = ""

End Sub
```

e) *Saving Data Macro*

Finally, the save data macro is called in Module 1 as below.

```
Sub ShowCOM()
'makes the COM block visible
Sheets("COM").Shapes("StrokeReader1").Visible = True
```

```
End Sub
```

The actual code is found in Module 2 as below. The macro copies and formats the data to a new sheet and formats the headers.

```
Sub savedata()
Dim WS As Worksheet

Set WS = Sheets.Add
    'copy, paste, and format data
        Sheets("COM").Range("A:F").Copy
        WS.Range("A:F").PasteSpecial Paste:=xlPasteValues
        WS.Range("A:A").NumberFormat = "0.000"
        WS.Range("B:E").NumberFormat = "0.00"
        WS.Range("F:F").NumberFormat = "0.0"
        WS.Columns("A:E").ColumnWidth = 15.6
        WS.Columns("F:F").ColumnWidth = 19
        WS.Columns("G:G").ColumnWidth = 20

    'Format headers
    With WS.Range("A:G")
        .HorizontalAlignment = xlCenter
        .VerticalAlignment = xlBottom
        .WrapText = False
        .Orientation = 0
        .AddIndent = False
        .IndentLevel = 0
        .ShrinkToFit = False
        .ReadingOrder = xlContext
        .MergeCells = False
    End With

    WS.Shapes.AddChart2(240, xlXYScatter).Select
    ActiveChart.SetSourceData Source:=WS.Range("$A$1:$E$425")    'repopulate
    chart
    With Selection
    .Top = Range("F4").Top                          'adjust top
    .Left = Range("F4").Left
    End With
```

Formatting the chart is very similar to the code seen earlier in the plot macro.

```
'*************** Format Chart ************************************
For Each chtObj In WS.ChartObjects
chtObj.Activate
ActiveChart.ChartTitle.Select
    Selection.Format.TextFrame2.TextRange.Characters.Text = "Speed Data"
    With Selection.Format.TextFrame2.TextRange.Characters(1,
    10).ParagraphFormat
        .TextDirection = msoTextDirectionLeftToRight
        .Alignment = msoAlignCenter
    End With
    With Selection.Format.TextFrame2.TextRange.Characters(1, 5).Font
        .BaselineOffset = 0
        .Bold = msoFalse
        .NameComplexScript = "+mn-cs"
```

```
        .NameFarEast = "+mn-ea"
        .Fill.Visible = msoTrue
        .Fill.ForeColor.RGB = RGB(89, 89, 89)
        .Fill.Transparency = 0
        .Fill.Solid
        .Size = 14
        .Italic = msoFalse
        .Kerning = 12
        .Name = "+mn-lt"
        .UnderlineStyle = msoNoUnderline
        .Spacing = 0
        .Strike = msoNoStrike
    End With
    With Selection.Format.TextFrame2.TextRange.Characters(6, 5).Font
        .BaselineOffset = 0
        .Bold = msoFalse
        .NameComplexScript = "+mn-cs"
        .NameFarEast = "+mn-ea"
        .Fill.Visible = msoTrue
        .Fill.ForeColor.RGB = RGB(89, 89, 89)
        .Fill.Transparency = 0
        .Fill.Solid
        .Size = 14
        .Italic = msoFalse
        .Kerning = 12
        .Name = "+mn-lt"
        .UnderlineStyle = msoNoUnderline
        .Spacing = 0
        .Strike = msoNoStrike
    End With
    ActiveChart.SetElement (msoElementPrimaryCategoryAxisTitleAdjacentToAxis)
    Selection.Caption = "Time"
    Selection.Format.TextFrame2.TextRange.Characters.Text = "Time (s)"
    With Selection.Format.TextFrame2.TextRange.Characters(1,
    8).ParagraphFormat
        .TextDirection = msoTextDirectionLeftToRight
        .Alignment = msoAlignCenter
    End With
    With Selection.Format.TextFrame2.TextRange.Characters(1, 8).Font
        .BaselineOffset = 0
        .Bold = msoFalse
        .NameComplexScript = "+mn-cs"
        .NameFarEast = "+mn-ea"
        .Fill.Visible = msoTrue
        .Fill.ForeColor.RGB = RGB(89, 89, 89)
        .Fill.Transparency = 0
        .Fill.Solid
        .Size = 10
        .Italic = msoFalse
        .Kerning = 12
        .Name = "+mn-lt"
        .UnderlineStyle = msoNoUnderline
        .Strike = msoNoStrike
    End With

    ActiveChart.Axes(xlValue).Select
    ActiveChart.SetElement (msoElementPrimaryValueAxisTitleNone)
```

```
    Selection.HasTitle = True
    ActiveChart.Axes(xlValue, xlPrimary).AxisTitle.Text = "Speed (m/s)"
```

Again, the new chart y-axis limits are set here, which could be modified to be more efficient.

```
    'Legend and y-axis limits
    ActiveChart.SetElement (msoElementLegendTop)
    ActiveChart.Axes(xlValue).MaximumScale = 3
    ActiveChart.Axes(xlValue).MinimumScale = 0

    'Change target speeds to lines, not points
    ActiveChart.FullSeriesCollection(4).Select
    With Selection.Format.Line
        .Visible = msoTrue
        .ForeColor.ObjectThemeColor = msoThemeColorAccent2
        .ForeColor.TintAndShade = 0
        .ForeColor.Brightness = 0
    End With
    Selection.MarkerStyle = -4142
    ActiveChart.FullSeriesCollection(3).Select
    With Selection.Format.Line
        .Visible = msoTrue
        .ForeColor.ObjectThemeColor = msoThemeColorAccent1
        .ForeColor.TintAndShade = 0
        .ForeColor.Brightness = 0
        .Transparency = 0
    End With
    Selection.MarkerStyle = -4142
Next
```

The rest of the code makes the window into which you input the saved data name and creates the text box on the new sheet accordingly.

```
Sheets("COM").Activate  'Re-selects the COM worksheet

'Make a note on the circumstances of your data
note = InputBox("Input the scenario this data represents:  ", "Data
Description")
WS.Range("G1") = "This data was recorded:"
WS.Range("G2") = Now

'Add a textbox with the note in it
Dim shp As Shape
Set shp = WS.Shapes.AddTextbox(msoTextOrientationHorizontal, 100, 100, 200,
50) ' add shape
With shp
.TextFrame.Characters.Text = note           'add text to display
.Top = Range("J1").Top                      'adjust top
.Left = Range("J1").Left                    'adjust left
.TextFrame.AutoSize = True                  'turn on autosize
.Fill.ForeColor.RGB = RGB(255, 255, 204)    'choose fill color
.Line.Weight = 1                            'adjust width
.Line.ForeColor.RGB = RGB(255, 0, 18)       'choose color
.Line.DashStyle = msoLineSolid              'choose style
End With
```

```
End Sub
```

C. Implementing Model Changes

On the car you can turn on and off the PID controls, depending on what type of testing you want to do. If the PID controls are off, the “cap” variable is the only parameter that needs to be changed, as this sets the limit of the PWM set to the motor at max throttle input. Reasonable values are within the range of 0.6 to 1.0 PWM. This allows for data collection at different fixed speeds. It is important to note that there is a finite resolution of this limit which seems to be two decimal places.

With the PID controls enabled, the PID coefficients on the car and the max speed value on the car and in the VBA code are critical to collecting correct data. The max speed will limit the target values of the throttle input so that data without noise can be collected. This value decreases with increased turning angle to around 0.6 m/s, with a max value of 1 m/s for straight runs of the car with a step input.

D. Verifying Results

Of course, the goal of this testing is to see that the wheel speeds and setpoints of the car are around the same value and that the approach to these values reflects changes in the PID coefficients accordingly with the theory. Some sample data can be seen in the Physics Theory section. The current setup is limited by undersized motors and signal noise from the h-bridges when operating at high PWM values, so few modifications are currently possible. See the Difficulties Documentation in the Additional Information section for more information on these limitations.

IX. Additional Information

A. Imagine RIT

The demonstration at Imagine RIT went very well, and the exhibit was a huge hit with both children and adults. The basic setup we used was keeping the car in the direct drive mode (no PID or open loop feedback) with a capped max speed of around 0.7 PWM. The console was near the front of the Freescale Cup track area, and participants drove the car around one of two tracks in the morning session while students were testing their cars for the competition. Later in the afternoon (following the competition) the entire track was available for participants to drive. We actually had a line all the way up until 5 PM, then the students from the competition wanted to drive the car without speed limits. Needless to say, the exhibit was awesome.

Participants were given a few laps to drive the car (~5) and were asked a series of short survey questions afterwards to gain feedback on their experience. Most of these questions collected data for the test plan document, but we also had a comments section for any input given. The comments can be seen below.

Appearance:

- It was good. I like it very much.
- It looks like it took a long time to build.
- The "-X-" on top is cool.
- The project is professional
- The setup looks cool and safe

Suggestions:

- Brighter colors for the components could improve experience.
- The car needs fiberglass body.
- The steering is a little sensitive.
- We could make the controls a little better.
- We could make it a little faster.

Issues:

- The car gave a participant vertigo.
- Young children need to sit on parent’s laps because they cannot touch the pedals.

Observations:

- Pulling on the wheel can break it, since it is not load bearing.
- It is hard to separate the camera delay from the control delay.
- Sometimes the car wouldn't drive straight and the video cut out.

As the car speed was limited using a potentiometer on the car and operators were already busy enough trying to deal with the influx of people, adjustments could not be made to increase speed on an individual basis. Having this adjustability on the console would be a nice touch for future work, but would require changing our communications significantly. The sensitivity of the controls could be adjusted in a similar fashion.

Though our design attempted to make the console adjustable for people of all sizes, very small children seem to be outside of this range. For the future, it would be helpful to make the pedals fixed to the table base and adjustable in height with around a foot of travel upward. More

adjustment in the height of the seat and being able to reduce the height of the top table surface may be desirable. Effectively the lack in adjustability made it difficult for children to press the pedals, which would move on the table base and had to be held in place. Since the seat had to be low to reach the pedals, children were below the height of the steering wheel, which made them pull downwards to see the screen. This broke the clamping mechanism, which had to be held in place manually by a team member. This was replaced by a bolted connection. An additional option may be a small console specifically set up for small participants, now that the functionality is fully developed. With this, some additional time could be spend on the aesthetics of the car and console (fiberglass body, brighter colors, etc.)

B. Controls Class User BOM

Controls Class User BOM (screws not included)				
Number	Item	Qty. needed	Total Price	Purpose/Description
#	nd	#	$	nd
1	**Microcontroller**	1	0.00	Freescale KL25Z, 1 is included in chassis kit
2	**Wireless Tx/Rx kit**	1	84.99	Transmit driving controls (XBee kit)
3	**Motor Controller**	1	0.00	Freescale Motor Shield, Included with chassis kit
4	**Batteries**			
4.1	Car Battery w/Connectors	1	9.99	3000 mAh NiMH
5	**Battery Chargers**			
5.1	Car battery charger	1	7.99	To charge the car batteries
6	**Drive wheel encoder system**			
6.1	Encoder	2	23.14	Encoders are 3D-printed
6.2	Optical Switches	2	8.00	For wheel speed sensing
7	**Console**			
7.1	Computer, mouse, keyboard	1	0.00	Run the entire Console
8	**Car Chassis Kit**	1	200.00	Freescale Cup chassis, KL25Z, motor shield
9	**Assorted other components**			
9.1	5mm rod (x3ft)	1	3.62	For new wheel layout
9.2	3mm E-rings (x100)	1 for whole class	0.00	
9.3	5mm E-rings (x100)	1 for whole class	0.00	
9.4	8-32 locknuts (x100)	1 for whole class	0.00	
9.5	2.8" diameter wheels (x2)	2	18.98	
9.6	bearings (x10)	1	9.99	
9.7	8x10x.1" Lexan sheet	1	3.98	Adapter Plate, chassis modifications
9.8	Stand-offs .75 in	2	1.22	stand-offs for back Lexan plate
9.9	Stand-offs 2 in	2	1.94	stand-offs for front of Lexan plate
		Total Price	373.84	

Number	Source
#	nd
1	http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=FRDM-KL25Z&tab=Buy_Parametric_Tab&nodeId=015210045A&pspll=1&fromSearch=false
2	http://www.parallax.com/product/32440
3	https://community.freescale.com/docs/DOC-93914
4	
4.1	http://www.tenergy.com/11204-01
5	
5.1	http://www.tenergy.com/01025
6	
6.1	http://edge.rit.edu/edge/P14226/public/Detailed%20Design%20Subdirectory/CAD
6.2	http://www.digikey.com/product-detail/en/QRB1134/QRB1134-ND/187533
7	
7.1	Provided
8	http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=TFC-KIT&tab=Buy_Parametric_Tab&fromSearch=false
9	
9.1	http://www.mcmaster.com/#88625k64/=r1qj8m
9.2	http://www.mcmaster.com/#98543a101/=r1qh4b
9.3	http://www.mcmaster.com/#98543a112/=r1qih6
9.4	http://www.mcmaster.com/#90631a009/=r1qkau
9.5	http://www.amain.com/product_info.php/cPath/1_25_2981_2983/products_id/251671/n/Vaterra-Gravel-Spec-Pre-Mounted-Tire-Set-2
9.6	http://www.amain.com/product_info.php/cPath/1576_63/products_id/5728/n/ProTek-R-C-5x11x4mm-Metal-Shielded-Speed-Clutch-Bearings-10
9.7	http://www.homedepot.com/p/LEXAN-10-in-x-8-in-Polycarbonate-Sheet-31-GE-XL-1/202090134?N=5yc1vZbrdg
9.8	http://www.mcmaster.com/#93505a825/=r1rg66
9.9	http://www.mcmaster.com/#91780a770/=r1rcyn

C. **Difficulties Documentation**

Many unforeseen challenges were tackled in the course of completing this project. This Section mentions many of them as a means to prevent future groups working on this project from having to deal with the same issues.

Difficulties Documentation		
	Difficulties	**Solution**
Electrical	FRDM KL25Z Board has 3.3 V inputs only	Use drop down resistors and a level shifter to change sensor voltage ranges
	Centering signal was not reaching high enough voltage levels when going through the level shifter	This was due to the level shifter receiving its power from the FRDM board, which cannot provide enough power to achieve the desired voltages. The level shifter is now powered from the MOMO's 5V power supply.
	Encoder values are very noisy	Capacitors were added to the motors and power supplies, but this was not enough. Encoders are still noisy. Problem comes from the H-Bridge.
	The system was very electrically noisy in general.	Ferrite cores were added to the batteries to help reduce noise propagation.
	Wires often slipped out of their positions during operation	Headers and harnesses were created for each wire so the wires fit more securely and could not be misplaced in the boards.
	Camera feed would occasionally drop out while operating indoors	Replaced antennas to give the wireless signal more axes of operation. Indoors is still not best for operation, but rooms with higher ceilings are best.
	Connecting and disconnecting the battery for testing was inconvenient and inefficient.	A simple switch was added to the chassis that allows for easy code development and testing on the chassis.
	The addition of a separate LiPo battery added the possibility of the voltage dropping too low and making the battery unusable during testing.	A LiPo battery level indicator with audible alarm was added to signal when voltage is getting low.
	Default motor wires included unnecessary connectors that made connecting to the TFC shield inconvenient.	New wires were added at the optimal length for the chassis layout.
	The encoder wires were also possibly targeted as a cause of the noise seen, as they sit directly on the motors.	Tin foil was added over the motor area around the wires to protect against signal interference.
	Video receiver was designed to be powered by LiPo batteries at fixed voltage ranges that did not readily correspond to DC wall adapter values.	A voltage regulator was added to a DC wall adapter to provide the desired supply voltage.

	Issues were encountered with trying to power the console Freedom board with external power sources while allowing for serial connection over the Open SDA port.	A separate cord and USB adapter (or the computer when data logging) are used to power the console board through the Open SDA port from those used to power the rest of the console.
	Dead pins on the console Freedom board were encountered in interfacing (PTE1 and PTD7).	Suitable working pins were substituted to enable functionality.
Computer	The motors were emitting a high pitched noise while the car was powered on and the H-bridges were supplying a 50% duty cycle signal.	The motors were initialized with 9000 Hz frequency using the TFC library, which reduced audible noise completely.
	Hardware interrupts from the wheel encoders were disrupting the wireless communication. It caused noticeable lag in the system.	The solution was to multithread the car code. A thread for each encoder was created.
	Communication problems were difficult to debug when the data was redirected through Matlab.	The solution was to remove Matlab and transmit the data directly through the XBee's.
	There was difficulty with calibrating the Logitech pedals. The pedals used potentiometers, however originally, they drew power from a USB port on the computer. The problem was that the range of potentiometer values changed for no apparent reason. It was determined that the USB ports on the computer did not output 5V consistently.	The solution was to use a 5V DC wall adapter to power the pedals. This resulted in a fixed, consistent range of potentiometer values.
	It is difficult to keep track of the NiMH battery voltage powering the car.	Analog read and battery LED indicators built into the TFC shield and TFC library were used to signal decreasing voltage levels.
	The center position of the steering servo sometimes needed to be adjusted slightly during testing for straight driving.	Potentiometer reading functionality on the TFC shield was enabled using the TFC library to allow for fine tuning of the servo center on the fly during testing.
	The max speed of the car during demos needed to be adjustable to allow for different drivers during the demonstrations.	Potentiometer reading functionality on the TFC shield was enabled using the TFC library to allow for limiting of the max speed of the car during forward movement when control systems are disabled.

	Issues were encountered with the internal pullup resistors used on the Freedom board and TFC shield that were interfering with the use of interrupt pins for encoder reading.	External pullup resistors were removed from the TFC shield. Separate pins from those used by the motor shield for speed sensing (PTA1 and PTA2) since these pins disable the UART functionality of the board for debugging. Separate external pullup resistors were calibrated and integrated into the XBee shield that yielded good voltage ranges for the encoder circuitry. The PULLNONE functionality of the interrupts was used.
	Noise issues caused by the motor shield led to multiple configuration attempts for powering the XBee shield, as can be seen by the extra open pins on the shield.	Power and ground for the shield was pulled from the 3.3V source for the speed sensors from the TFC shield to reduce external wiring.
	XBee communications kept dropping out once two-way communications were enabled.	Clearing the area around the front of the car XBee of electrical components was observed to greatly reduce the issue.
	Two-way data communications were otherwise inconsistent.	A compromising range of 50 ms between transmissions was reached, and printf and scanf commands in the code were enclosed in readable and writeable conditions to prevent buffer overflow.
	Excessive encoder noise was encountered at higher H-bridge PWM values.	H-bridge saturation was limited in coding to 0.75 PWM for the PID control. The max speed output of the PID was also limited to a range which did not produce much noise in the signal separately for straight driving and turning, as turning requires higher PWM values to the outside wheel.
	Steering servo values and camera servo values are frequently different due to slop in the steering servo gearing and frequent contact with objects. It was also determined through testing that adjusting the camera the same range as the wheels only worked for PID control (slow speed) and needed to be limited during higher speed driving.	Completely separate assignments for the camera and steering servos were established in the code.

	Issues were encountered with sending multiple values over the Xbee's for signals.	Transmission signals were converted to ASCII characters in the range of 35-126 to simply transmission. This range allows for the start bit of 33 corresponding to "!" and does not interfere with the new line or carriage return characters found at 32 and below. With the XBee's in transparent mode, they only read the characters as characters and not other transmission functions so this is safe.
	Character data transmission limited the resolution of the steering wheel, which has a large range.	Separate positive and negative characters corresponding to displacements CW and CCW of center, respectively, were used to transmit the steering signal.
	Separate modes of operation were necessary for PID control, open loop control, data transmission, standard driving with capped max speed, and other parameters that could be easily selected without uploading new code.	Fixed configurations are enabled through coding at startup based on TFC shield DIP switch settings.
	Encoder were reading static non-zero values when the car was not moving corresponding to the last pulsewidth length that triggered the interrupt.	A timeout was added to the code that writes the speed of the wheels to zero if a long time has elapsed since the last interrupt was detected.
	Median filtering was targeted as a possible solution to the issue of noise in the encoder data.	After testing it was determined that a five point median filter was actually making the data worse due to the consistent high frequency nature of the noise. Capacitors cannot be used as low pass filters as this reduces the square wave produced by the encoder circuit to a useless range.
	Noise pulsewidths in the encoder readings sometimes yielded values smaller than anything possible by the car. Very slow movement also yielded values outside the range that could be transmitted to the console using characters.	Encoder pulsewidths were capped at high and low values. If the value seen was unrealistically small, the value is set to the previous reasonable value. If the value is too large, it is set to the maximum pulsewidth which corresponds to no movement.
	The gas pedal spring data not always return the pedal back to exactly the same position, so the car would sometimes statically run without input.	Throttle and brake input was added together to produce only one signal from the user. A deadzone was established to ignore input under a certain limit that corresponded to the highest value seen in the throttle return range.

	Gas float (throttle input) values at 0 were causing strange output from the PID loop. The reverse functionality also was to be isolated from the PID loop.	Separate conditions for no throttle input and reverse were made that bypass the PID loop.
	The car kept taking off with the last signal received when data transmission was lost.	The motors are written to zero when data is not received, but this still does not seem to work. Speeds in reverse and otherwise were limited to mitigate this issue.
	One-way communications never have the issue of dropping signals.	Functionalities in both car and console code for turning on and off data transmission were made.
	Wheel range pulse signals from the quadrature encoder varied depending on the speed of turning the wheel from one extreme to the other.	A range of 650 was set that is larger than any values seen by a small margin to allow for reading of all values within range. A centering signal was also read in as an interrupt to reset the values to around zero whenever the center was passed. This sensor was already in the console electronics.
	Gas pedal and brake pedal voltage ranges differed in orientation from low to high and high to low.	Ranges were converted to be near the same from low to high for simplicity.
	Steering wheel, gas pedal, and brake pedals sometimes read inconsistent values.	Value maximum and minimum caps were added to prevent the transmission of bad characters over XBee.
Controls	Steering inputs near zero led to singularities in the wheel speed calculations.	Steering inputs less than 0.001 radians were limited to that value to correspond to roughly a straight signal without division by zero.
	Accumulated error needed to be reset when running the car from time to time.	Every time the car comes to a stop the error is reset for the next movement.
	Assignment of the correct wheel to assign outer and inner speeds based on turning signal was confusing.	The car was tested unloaded at slow speeds while holding one wheel constant to determine when PID was functioning correctly.
	Open loop feedback led to the possibility that the outside wheel could be written to a value greater than 1.0 PWM.	Both inside and outside wheel speeds were limited linearly so that the highest value the outside wheel could see is slightly less than 1.0 PWM. This logic did not need to be employed in the PID loop as saturations at 0.75 PWM were already in place due to noise issues.

Mechanical	The original car suspension system added no functionality to the car and made mounting components difficult.	The suspension system was removed and removed with fixed connections that were reinforced with other structures to provide a solid platform.
	Little room was available for mounting electrical components on the original chassis, and the position over the motor posed noise issues for the microcontroller.	A Lexan mounting plate that spans the entire chassis was created that increases the rigidity of the frame and provides ample mounting room for electrical components far away from motor interference areas.
	The electrical components added that need to be accessed and monitored frequently were otherwise unprotected from outside forces.	A Lexan protection plate was mounted on standoffs above the electronics to allow for easy access from the sides while protecting from impacts from the top.
	The car's existing bumper was very small and provided little protection.	Front and rear polyethylene foam bumpers were designed and added to the chassis that adequately protect the components from front and rear impacts.
	Stock tires and wheels on the chassis did not provide adequate ground clearance for getting over small obstacles and did not allow for the addition on encoder functionality.	New tires and wheels with normal mounting connections for RC cars were purchased for the car. Transaxles and a rear axle was fabricated to extend the car's track width to support the larger wheels and provide adequate room for 3D printed encoders to be mounted to the rear wheels.
	Standard mounting location for the servo on the chassis was inconvenient, used weak components, and in the way of more useful supports for the Lexan mounting plate.	The servo orientation was reversed and more robust mounting brackets were used to mount the servo for easy accessibility.
	Fabricated rear axles was making a squeaking noise while turning from rubbing on the inside of the encoder plastic.	Grease was put on the rear axle that reduced the noise completely and reduce rolling resistance.
	Encoder optical switches were not reading the same values.	The switches were moved so that an optimal distance of 3 mm was achieved between the emitter and encoder face. The mounting locations are somewhat adjustable to accommodate this issue.
	Encoder optical switch wires were very stiff and could not be used with chassis.	Wires replaced with flexible speaker wire and servo headers for easy connection to the XBee shield.
	Quick connection of the light camera battery and voltage monitor were necessary for quick change out.	Velcro was used to secure both components in a minimal configuration.

	Multipath interference was targeted as the cause of noise in the camera signal.	The transmitter was mounted as high as possible on the console and the transmitter was moved to the highest point on the car to reduce the number of objects in the way of the signal path between these components.
	People of different heights were going to be using the console for driving and would be frequently getting into and out of the seat for testing and demonstrations.	Height adjustability was added to the seat along with mobility of the desk to bring the pedals and steering wheel closer to the user.
	Console electrical components needed to be mounted in the console for protection. Access to the inside of the console is also frequent necessary for monitoring of component performance.	An access hatch was created in the console, along with the addition of mounting locations for the interfacing electronics and access points for sub connections.
	Data logging, programming, and MATLAB usage made version control of programs problematic for the final deliverable.	A computer was procured for the project which was secured to the console as a dedicated programming station.
	The console needed to be mobile for various demonstrations and testing.	A robust subframe was fabricated for connecting the chair, desk, and computer. Casters were added to allow for easy moving. A pull rope with quick mounting catch was also made to making movement easier.
	The image of the camera became blurry on the screen after testing.	There is a small set screw that holds the lens at a fixed adjustment on the camera that can come lose through vibration. This was adjusted until the image was clear again and tightened.