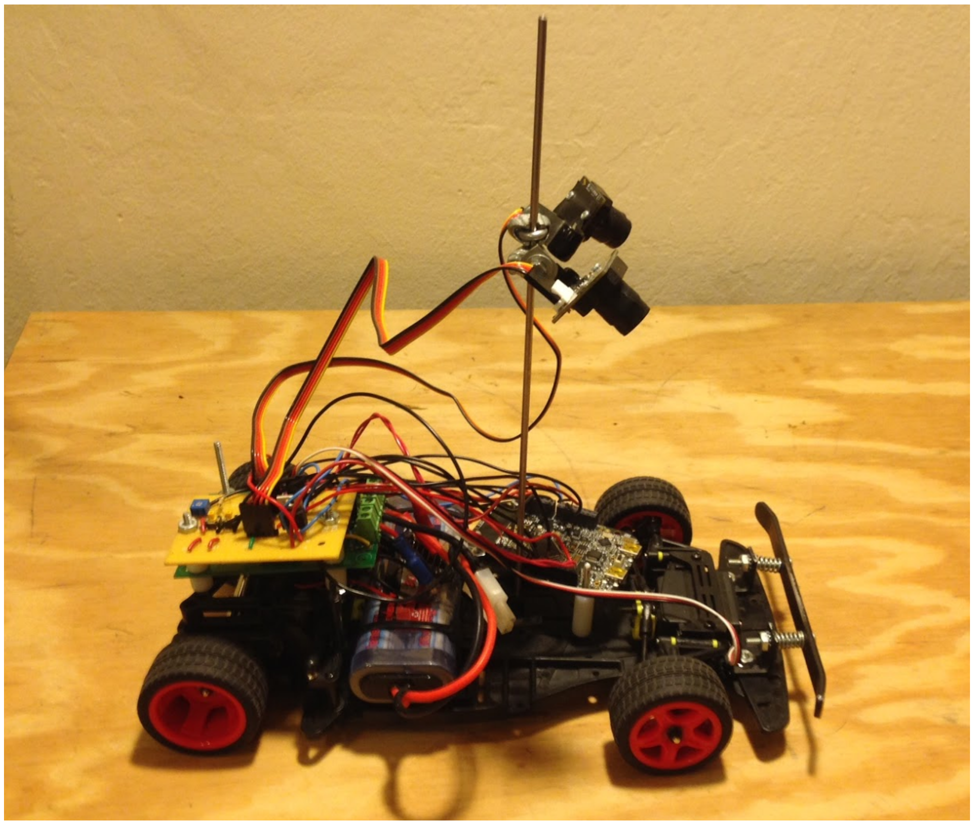
EEC 195
Autonomous Vehicle Design Project
Fall 2014 – Winter 2015



TEAM BBEJ

Brian Kong
Bruce Krietzer
Earnest Sayles III
Justin Scaccianoce

Final Report: 3/19/2015

I. Overview (Executive Summary)

The basic approach taken in this design project was to try to overcome challenges as they presented themselves because so little was known about what would be necessary to achieve the required performance. Beginning with just getting the car around the track, we took note of performance issues and deficiencies, and systematically addressed each problem preventing us from achieving a competitive performance. Due to the fact that we didn't have a real outline or procedure to follow, we had to work as a team to on each problem that presented itself and work creatively to find a viable solution. Some of the more important challenges included camera placement and data acquisition, reliable steering and motor control, and the seamless integration of these concepts to create a competitive project.

Our design was handled as a team with the tasks being split amongst the team. Justin Scaccianoce handled a majority of the coding in our design. The other tasks such as hardware, motor control design, troubleshooting our vehicle, report write-ups and record keeping were handled by the entire team collaboratively. The task percentages are as follows:

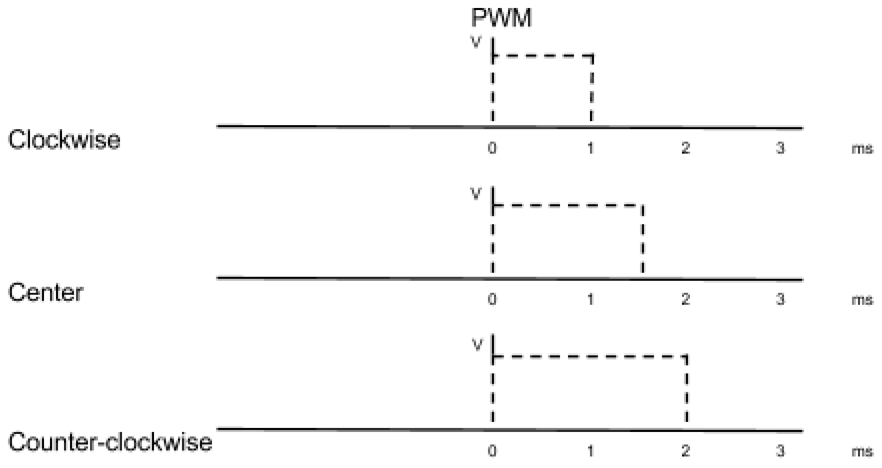
Team Member	Effort Percentage
Justin Scaccianoce	30%
Earnest Sayles III	23%
Brian Kong	23%
Bruce Krietzer	23%

II. Technical Reports

Brian Kong - Steering Control Loop

Hardware

The steering control loop is mandated by pulse width modulation (PWM). This pulse width is sent to the servo at a frequency of 50Hz and ranges from 1ms to 2ms. At 1ms, the servo is fully turned in the clockwise direction. Conversely, the servo is fully turned in the counterclockwise direction when the pulse width is at 2ms. In addition, the servo is centered at the midpoint of 1.5ms. Throughout a turn, an average power of 2.34W was drawn from the servo. This PWM output signal is sent to the servo as TPM1. The pulse width sent into the servo is dependent on the values calculated from the ping pong buffers.



Determining Pulse Width

Since the servo relies on a pulse width input, it's important to send just the right amount of pulse at specific times. In our system, the amount of pulse width sent to the servo is dependent on the error values calculated from the ping pong buffers. This also means that the cameras have a direct effect on the function of the servo; factors such as camera placement, the room's lighting, and sensitivity threshold will play an important role in the servo's performance.

The camera buffer calculations yield error values from both the left and right camera. These error values are then tested through a series of if-else statements in the steering control section of the code.[1] An average of four error values are checked every 16 cycles. Depending on the error value from each camera, the servo would turn left, right, or straight. In addition to each left or right turn, there are smaller adjustment conditions which consider minimal corrections during high speed straight sections.

For left and right turn conditions, the left and right error values were checked. In the case of a left turn, the right error values from the right camera are tested. The conditions are met when these error values are within a certain threshold. At the same time, the opposite camera error value is also tested; the left camera error value must be under a certain amount of error in order to satisfy the left turn condition:

```
else if ((ErrorR>=32 && ErrorR < 128) && (AveL<=0x0F) && !(ErrorR<(AveR-0x1A)) &&
!(ErrorR>(AveR+0x1A)))
```

When the conditions have been met, a pulse width is set and sent to the servo via TPM1:

```
W1 = (4600-(((ErrorR1)*1600)/(65))); //calculate value to be sent to servo
```

However, sending too great of a pulse width would cause the servo to turn too much. Turning too far in one direction can cause a wheel to rub against the chassis and effectively jam the wheel during a hard turn. In order to fix this, the amount of pulse width was limited to a certain value so that an excessive amount of pulse would not be sent to the servo:

```
if ( W1 > 3250) // 3250 pulse limit
  TPM1->CONTROLS[0].CnV = W1; // send calculated value if pulse limit hasn't been reached
else
```

[1] See steering control logic in Appendix

```
TPM1->CONTROLS[0].CnV = 3250; // if pulse limit reached, send limit to servo
```

This pulse width is proportional to the error value from the right camera in the case of a left turn. The average of the left and right error values were also considered in order to restrict unwanted steering during a turn. This effectively produced smooth turning results and eliminated a significant amount of twitch throughout the turn.

When there is a small amount of error on either cameras, the servo will be centered. The threshold values for the centering of the servo are not symmetrical due to imperfections in hardware. These imperfections include the mounting position of the cameras, compensation for the alignment of the car's wheels, and tie rod adjustments connecting the front wheels to the servo. When this small error is detected, the pulse width is sent to the servo via TPM1 to be centered.

The threshold values in these if-else statements are a product of trial and error testing on the track. Each condition had to be fine-tuned so that the car can perform well on all turns of the track.

Control

For regular left and right turns, differential error values were considered in calculating the pulse width to be sent to the servo. This would allow the car to predict turns as it runs around the track. This was especially useful when coming down a hill into a turn. The car has to be able to predict the turn due to the increased speed coming down the hill. The car is also blind for a certain amount of time on the hill due to the direction that the cameras are facing. This is also useful when coming across tight turns or chicanes. However, proportional was still the dominant control in order to quickly react to changes based on high error values. When the car needs to make minimal changes, the differential error values were not considered in order to make quick adjustments.

Bruce Krietzer - Edge Detection

Detecting the edges of the track by our pair of line-scan cameras was pivotal in finding and utilizing error values for feedback control. These error values were used in both speed control and servo control to help the car make its way around the course. For line detection, we were presented with two different line detection schemes in lab; one dealt with a voltage threshold while the other dealt with slope thresholds. We decided to go with a slope threshold algorithm to detect the lines of the track. Our reasoning was that detecting a sharp slope would be simpler to implement and would more effectively ignore noise or lens aberrations; while detecting for a certain voltage threshold could prove problematic if lighting conditions weren't satisfactory.

Slope Threshold Line Detection

By displaying the output of the linescan cameras on an oscilloscope, we were able to note that a sharp negative spike in the signal is displayed when the pixels go from white to black and a sharp positive spike is displayed in the other direction. Our goal was to set threshold values for negative and positive slopes where slopes below the threshold would be ignored and large slopes would be used to detect a line on the track.

In order to implement a slope threshold line detection scheme, we utilized the data stored in the ping pong buffers. The data from the buffers give us a certain intensity (light or dark) corresponding to a certain index value (pixel number). Our field of view was 129 pixels wide for each camera.

Code Breakdown

We used a central difference equation (derivative) in our code to calculate the slope.[2]

data2=((CenterStr[i+1]-CenterStr[i-1])/2)

The variable ‘data2’ would store new slope values as the index moved from i=2 until it reached i=126. Even though the array ‘CenterStr[128]’ holds 129 values (one for each pixel), we start the index at 2 and end at 126 so that boundary conditions do not cause glitches in the code. If i=1 or i=127 were considered, the algorithm would catch the boundary pixel values and might miscalculate a slope. Also, due to lens aberration the boundary pixels might not be as accurate. It is also simpler to exclude these pixels in the code than to preset boundary conditions because of varying lighting situations.

Each side of the track had a slightly different line detection scheme. For the left side, in order to find a local max value we initialized a local variable “high” to equal zero. Inside of the while loop that is counting from i=2 until it reaches 126, an ‘if’ statement is asking if the current ‘data2’ value is greater than ‘high’ (which starts at 0); if so, make high equal to that data2 value and set another variable ‘locmax’ equal to the index at that present ‘data2’ slope. The point is to constantly have ‘locmax’ equal to the pixel index at the center of where we believe the black line to be, or the center of where the slope is the greatest.

The right side implements a similar code[3] except we are asking if ‘data2’ is lower than a ‘low’ variable initialized at 0xFF. This finds the greatest slope in the *other* direction (noting that data2 is a signed integer) and sets a value ‘locmin’ equal to the index at that slope.

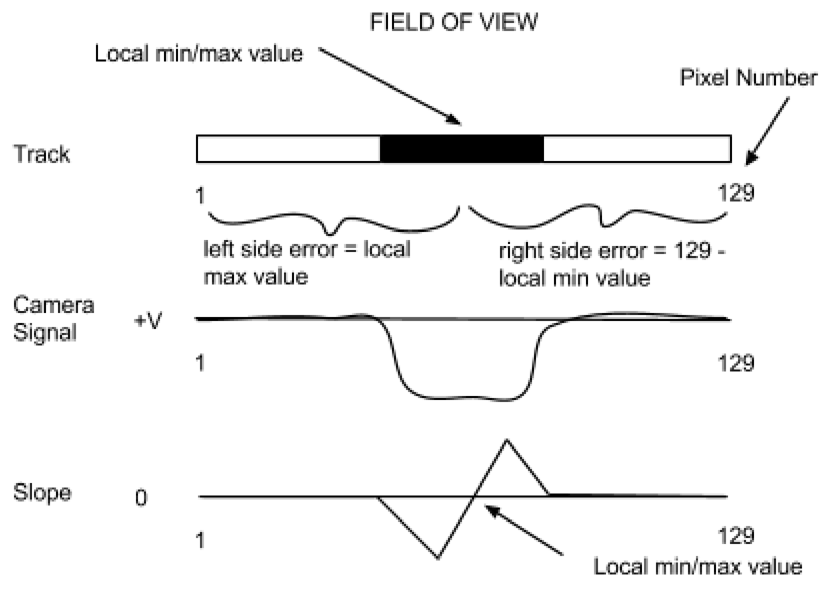
Due to unwanted noise and slow rates of change caused by lens aberration, we had to choose a threshold value for the slope so that small slopes would be ignored. We found a reasonable

[2] See function block: int LeftEdgePosition(int CenterStr[128]) in Appendix

[3] See function block: int RightEdgePosition(int CenterStr[128]) in Appendix

threshold by analyzing the values that were printed to a terminal when we had a camera pass over a line. At the end of the while loop counting from i=1 until i=126, if the variable ‘high’ is *not* greater than that certain threshold, then an error variable is set to equal 0x00 (no error). If this slope variable does exceed the threshold (and is considered a change from white to black or vice versa), then an error is set. For the left side error, the error variable is equated to the ‘locmax’ index value and pushed to an error buffer.

For the right side error, the error variable is equated to 129 - ‘locmin’ and pushed to an error buffer. It is important to also note that we did not push the error to the error buffer unless it was greater than a certain preset error value. This was done to keep our car in the center of the track given our camera setup and to ignore the edge pixels as much as possible. The diagram below gives a better visual on how these error values were calculated.:



<u>Earnest Sayles III - Cameras</u>

<u>Hardware</u>

One design factor that we had to decide was how many cameras we wanted to utilize in our final design, and we decided that using a two camera system would be best. Both cameras were mounted on a pole that is centered on the car at a specific height of 7.5 inches in order to achieve the proper angle looking outward. Understanding that the hypotenuse of the camera to the ground is equal to the width of the 128 pixels allowed us to determine the mounting height, giving us a camera width of about 12 inches. We utilized a measuring tape to make sure we were getting accurate heights and widths, as well as eliminating overlap of the cameras.

<u>Camera Angles</u>

The two cameras are slightly angled outward towards the track edges. We did this so the field of view for the cameras will be inside the track edges, but still able to quickly detect the edges as it deviates from the center of the track. This positioning helps with limiting oscillation. We took this approach to the placement because, instead of a line following design, our design avoids high error values which are created as the track edge progresses towards the center pixel of the camera. One expected tradeoff would be an overlapped field of vision, but we overcame this by positioning our cameras so our field of vision has a "V" shape with the center of the V pointing directly ahead of the car. Using the current camera positioning each camera sees only one line at a time, on either side of the car, with no overlap. For the vertical angle, we adjusted the cameras so they were seeing at the perfect distance ahead. If the cameras were to look downward, closer to the car, they would not be able to anticipate turns in the track; this would cause a delay in data for our derivative term causing the car to ride off the track. However, if the cameras were looking too far in front of the car, they could read false data beyond the track. This same issue occurs once hills are integrated and the car starts to point upward on the hill. Understanding these two limitations helped us find the right angle in the middle which we used an app to maintain.

Camera Focus

The process to get the optimal focus was to hook up the cameras to the oscilloscope and to actually evaluate the wave signal that was being produced. Naturally the camera has a fault slope on both edges when it is out of focus, so we wanted to have the widest and most positive band; eliminating the fault edges allows for the camera to better read when it is hitting the edge of the track rather than having an inaccurate threshold. After we found the proper focus position we used tape to hold the camera focus in place for a consistent quality focus.

Integration Time

The integration time is vital in assuring that the cameras are able to take in an correct amount of data. The different factors of any given room or track dictate whether or not the integration time should be increased or decreased. For example, if it is a dim room the integration time needs to be longer in order to fully read and process the value of the pixel that is coming in. However, due to dirt and gaps on the track, we must at times shorten the integration time so that we don’t process that false data. The minimum clock frequency is 33.75us at a max clock frequency of 8 MHz utilizing the following equation.

$$T = (1/max\ clk\ frq) \times (n\text{-}18)pixels + 20us$$

Ping-Pong Buffers

In order to maximize efficiency within the code it is important to have nonstop acquisition and interpretation of the data. The way this works is that while one buffer, 128 pixels, is being written to by the program the other is being read by the code within the ADC interrupt. As a result of the two camera system it is essential to implement ping-pong buffers with both cameras. Essentially both cameras have one buffer that is being written to and an alternative that is being read from. The code alternates between writing to both camera buffers one pixel value at a time; it alternates within every clock cycle. This allows both buffers to be filled at the same rate so that it can respond to either edge of the track appropriately. After both of those buffers are filled the code ping pongs both sets of camera buffers to read the new data in the ADC; it alternates between both cameras read and write buffers every time the SI signal is called.

ADC Function

First the ADC is triggered by the PIT handler. Then, at the end of each ADC, another ADC cycle is started; this cycle occurs 256 times, once for each pixel of the appropriate buffers camera. When triggered it is assigned a set value of either Set = 1 or Set = 2. This set value indicates which pair of ping pong buffers will be filled in the ADC. During the succession, a counter keeps track of how many conversions are being completed to insure that the ADC covers enough cycles. It also alternates between the two cameras based on the count value, which determine even and odd, to make sure that the data is being interpreted for each camera and edge of the track at the same rate. The actual clock signal is only cycled once, when each camera has been read, to insure that there are only 129 cycles per succession.

Justin Scaccianoce - Speed Control Loop

Basic Operation

One of the most important components of our vehicle is the speed control or motor control loop. In our design, motor control is very dynamic and plays a big role in our ability to negotiate turns competitively. Our motor control design has many subtle aspects but our principal concept is that our vehicle should go as quickly as possible down straight sections of track and break hard and fast while turning to quickly pivot 90 degrees.

Basic operation is as follows: An initial speed is selected using a single potentiometer. This establishes a constant pulse width value which will drive both motors. This pulse width, or speed, will be used as reference during vehicle operation and will be the steady state condition. The reference pulse width is modified in a number of ways during operation but ultimately, our motor control loop is managed by a single differential controller. We effectively use our error differential to reduce or increase pulse width, or change the direction of the motor completely.

Derivative Values

The choice to rely completely on the error derivative for plant control came after several attempts of using solely proportional control, then PD, then less proportional, then solely differential control. It allows the vehicle to respond, slow down, only when there is a sudden increase in error, which is almost always do to a fast approaching turn. In this way it is stopping as it enters a turn and accelerating as it leaves a turn.

Derivative: Acquisition and Thresholding

The derivative value is found using an averaging equation that is fed by a buffer of the four most recent error values. Diff = ((Error[0]+3*Error[1]-3*Error[2]-Error[3])/((6)*T));

Here, our time constant T was chosen to be .01 seconds because our PIT frequency was most often adjusted to frequency of 100Hz and so our error values are received at roughly .01 second intervals. It is important to note that the differential value is stored as a signed integer and processed as such. Values may be erroneous when this is not considered.

To make the differential values useful to speed control, modeling of different track scenarios was required. In our control design, the range of differential values was very large but this allowed for different ranges of values to be associated with the many different track features.

High Derivative Value

For example, assuming a speed of 8 ft/sec entering a quarter turn from center track, the differential value could be as high as 600-1600 error/sec. This would be an extreme case, and requires the most deceleration in the quickest time. To accomplish this, we actually reverse both motors to a relatively fast speed and keep them there for a predetermined amount of time after the initial high differential condition is recognized. This is done by starting a count down period of around 150 cycles of our Main-loop, during which time all other motor controls are locked out. Then, the motors are returned to the forward direction and the speed is momentarily made very high to accelerate after stopping.

Low Derivative Value

In less severe conditions, a differential value between 200 and 600 could indicate the vehicle is approaching a turn at a lower speed or navigating a long sloping turn. In these cases, the pulse width is changed as a function of the differential. When values are below ~400 just a fraction of the diff. is subtracted from the pulse width and above that the entire value is subtracted. This ensures the vehicle maintains an appropriate speed when navigating obstacles, but again allows it to return to speed quickly as it leaves them.

Negative Derivative Value

Using negative differential values further helps the vehicle to accelerate back to the desired speed when leaving turns. As the vehicle turn out of and away from a turn, the differential value is momentarily negative. At this point the pulse width is momentarily made larger then it set value and vehicle is able to quickly accelerate out of the turn.

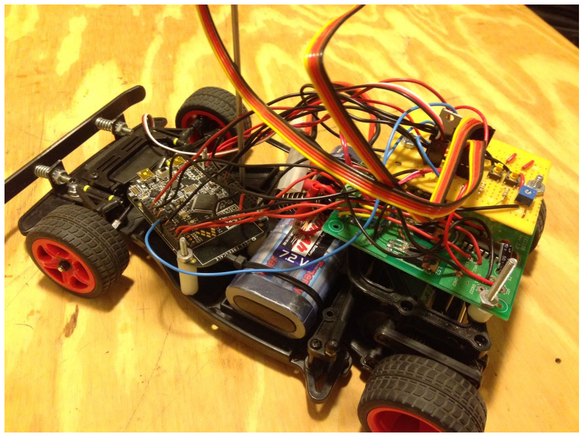
Hardware

Freescale Cup shield

When competing with Freescale rules, we take advantage of the Freescale Cup Shields' two on board MC33887 H-Bridges to control each motors. They are pre-configured to conveniently allow enabling/disabling through a single pin, and motor operation with two input pins per H-Bridge. We relied on unipolar pulse width modulation, in which one input to the H-Bridge was held either high or low, forward or reverse, and the second input provided with a square wave signal whose pulse width determined motor speed.

NATCAR Motor-Control Design

When competing with Natcar rules, we were required to assemble our own motor control hardware. We chose to use a simple, single, full H-Bridge to control both motors. In this way, we lost independent control of each motor, but retained the ability to quickly reverse the motors direction, which was an important aspect of our control logic. This was not big loss because we had already determined that independent motor control was of little benefit to our current control logic. Any traction control that we could have benefited from using independent motor control is lost when are main control strategy involves entering a turn at full reverse.



Code Description

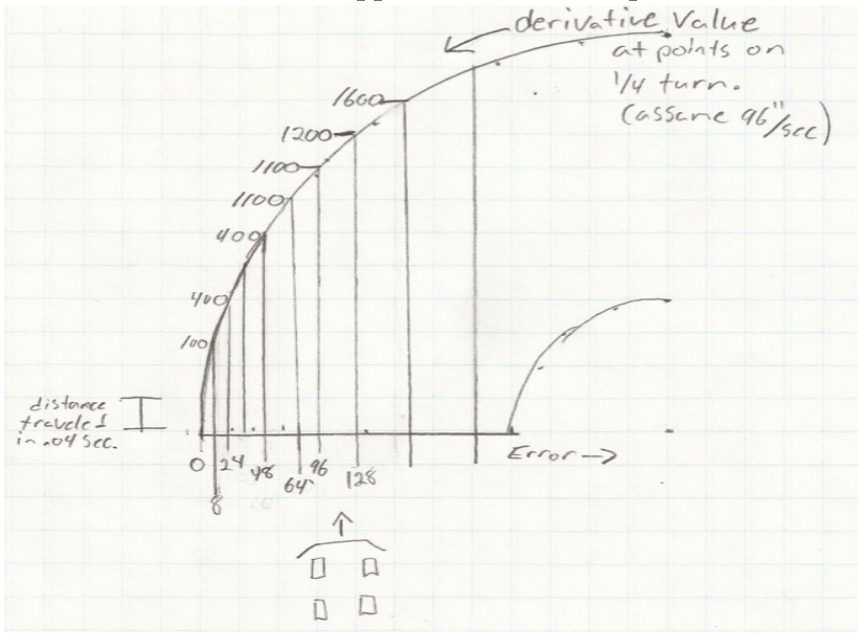
The code is very straight-forward for the control logic. On each cycle of the main loop, the derivative value is obtained from the differentiating function. Then a series of else statements compare the value and function accordingly. The motor is actually controlled in a couple of ways. It can be either enabled or disabled by a single pin, It can be driven in reverse or forward operation by making one pin per motor high or low, and the speed can be changed by changing the pulse width of the signal sent to either motor. To extend the breaking time, a variable Break is used. Once break is incremented above zero, it will lock out all other motor controls until it reaches 200 and resets.

Break-Down of Motor-Control

Derivative Value (Error/Second)	Action Taken
0-120	No action taken
120-650	Current Speed – [((Der. Value)*(1/2))/10] % duty cycle
650-3000	Reverse at 80% duty cycle for 200 main-cycles
-500-0	Current Speed + [((-Der. Value)*(1/10))/5] % duty cycle

*Duty cycle is measured from 0 to 1000, where 500 represents 50% duty cycle.

Derivative values as car approaches different points of ¼ turn:



III. Design and Performance Summary

Very quickly we started off progressing in the right direction seeing a lot of success in our early checkpoints. We were also successful in implementing various designs before settling upon the design that we used for our time trials. Over the course of the quarter we were able to construct a successful and fairly fast car. We created a custom motor control board that was innovative in the fact that we put two motors in parallel across an H-Bridge. Also, we managed to overcome a variety of problems as a team to make our car competitive. We learned and practiced new programming skills as well as technical skills that we will be able to apply in the future.

One improvement we could have made to the process would be to decide on a permanent design to perfect. Throughout the design process we saw benefits to various designs including the number of cameras and their placement. We tried multiple placements and quickly found out, with each one, that there were various trade offs. Instead of trying new camera placements and focusing on getting purely faster times, we recognize that it would have beneficial to focus on one design and work on consistency within our laps around the track. This would have allowed more planning, troubleshooting, and enhancing of performance all for our final cars design.

IV. Safety

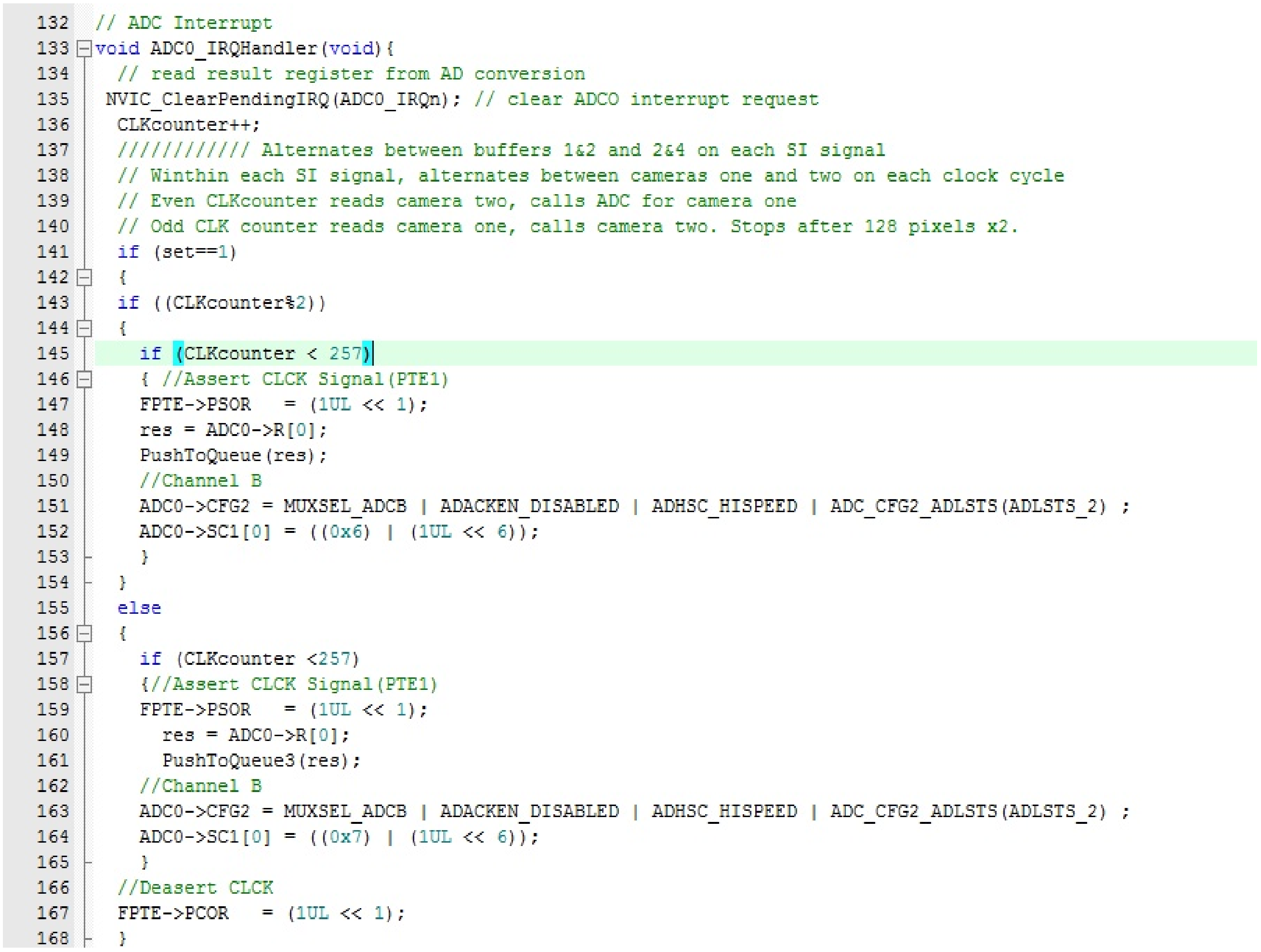
Safety goggles and proper soldering techniques were applied during the assembly of our motor control board and any other soldering that needed to be done. We installed a metal bumper on the front of the car to protect against collisions that may occur at the edges of the track or oncoming traffic in practice runs. The bumper was spring loaded to help ease the load on the actual car.

Also, "Emergency Stop" code was written in order to stop our car if it veered off the course. This was done to avoid being reckless and ensure the safety of both our classmates and our vehicle. This was implemented by simply adding a few lines of extra code to both the line detection and motor control sections. As pixel values were being read in and used to detect a line, we took an average value of the 125 pixels that were considered during the line detection code. Then, in the main function block (where the motor control code is), we checked to see if that average value was below a certain threshold; if the value is below the threshold (only seeing black) we shut off the motors thus stopping our car.

V. Appendixes

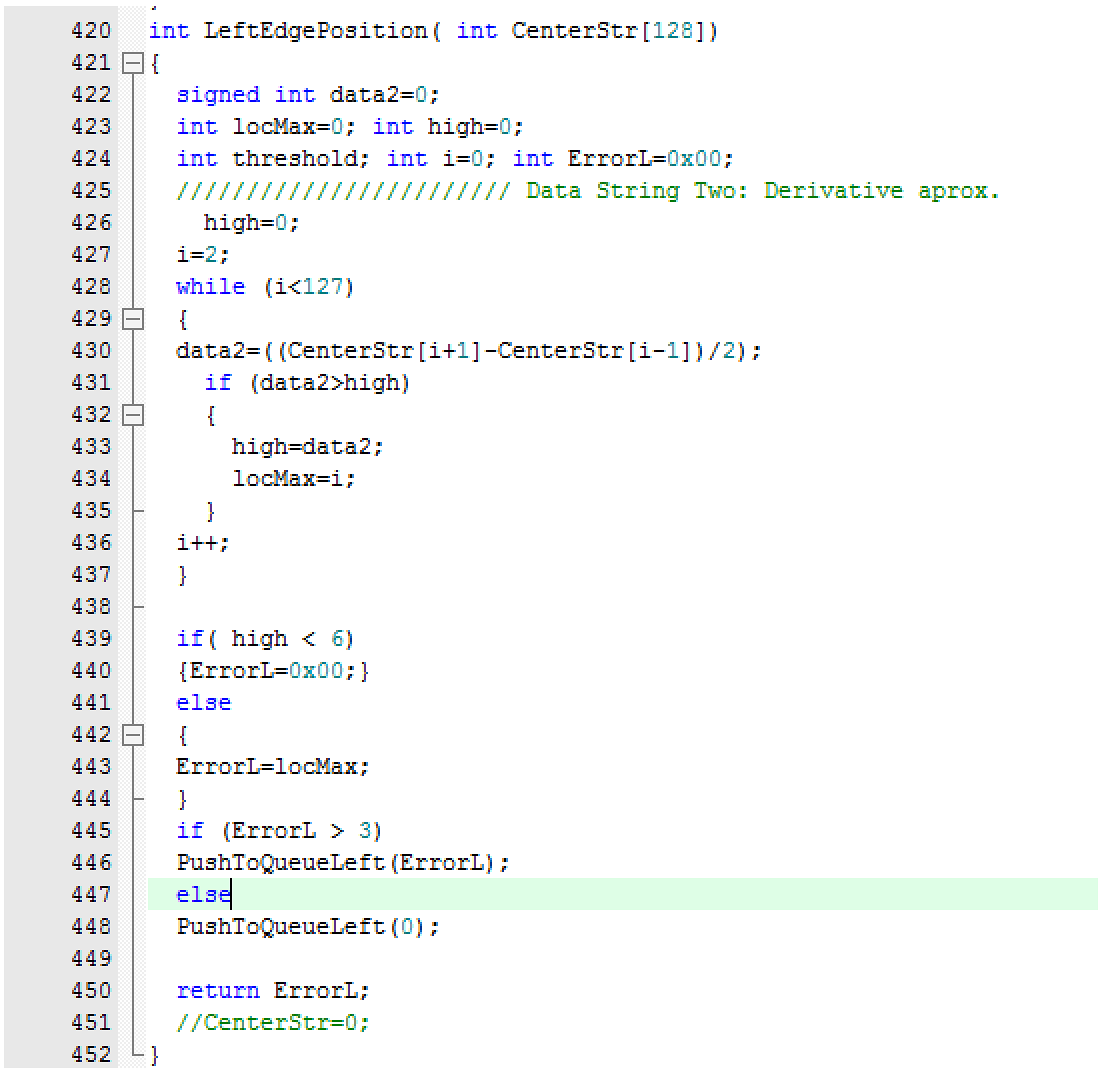
Code Examples

Example Of ADC Code:

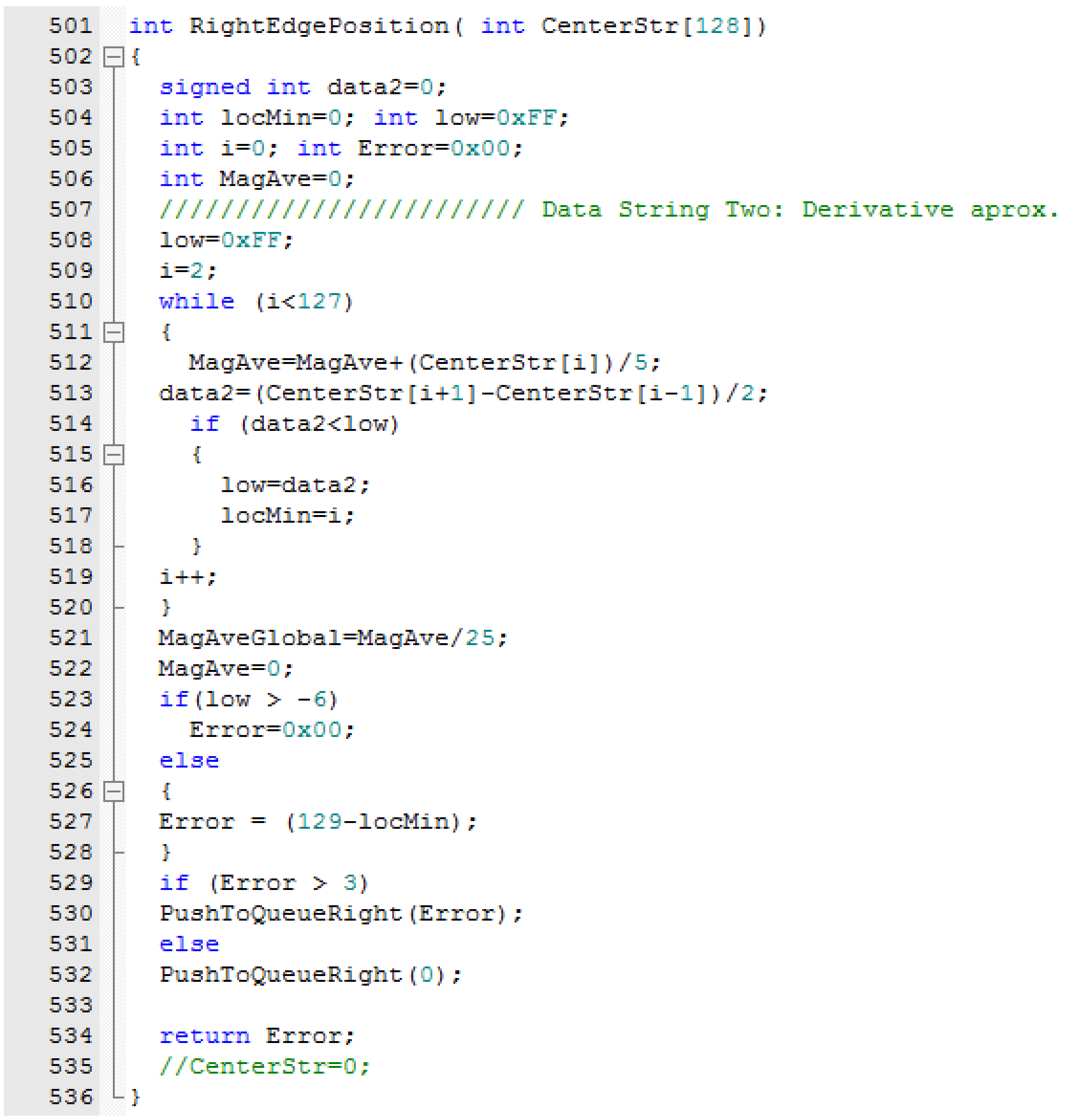


```
// ADC Interrupt
void ADC0_IRQHandler(void){
  // read result register from AD conversion
 NVIC_ClearPendingIRQ(ADC0_IRQn); // clear ADCO interrupt request
  CLKcounter++;
  //////////// Alternates between buffers 1&2 and 2&4 on each SI signal
  // Winthin each SI signal, alternates between cameras one and two on each clock cycle
  // Even CLKcounter reads camera two, calls ADC for camera one
  // Odd CLK counter reads camera one, calls camera two. Stops after 128 pixels x2.
  if (set==1)
  {
  if ((CLKcounter%2))
  {
    if (CLKcounter < 257)
    { //Assert CLCK Signal(PTE1)
    FPTE->PSOR   = (1UL << 1);
    res = ADC0->R[0];
    PushToQueue(res);
    //Channel B
    ADC0->CFG2 = MUXSEL_ADCB | ADACKEN_DISABLED | ADHSC_HISPEED | ADC_CFG2_ADLSTS(ADLSTS_2) ;
    ADC0->SC1[0] = ((0x6) | (1UL << 6));
    }
  }
  else
  {
    if (CLKcounter <257)
    {//Assert CLCK Signal(PTE1)
    FPTE->PSOR   = (1UL << 1);
      res = ADC0->R[0];
      PushToQueue3(res);
    //Channel B
    ADC0->CFG2 = MUXSEL_ADCB | ADACKEN_DISABLED | ADHSC_HISPEED | ADC_CFG2_ADLSTS(ADLSTS_2) ;
    ADC0->SC1[0] = ((0x7) | (1UL << 6));
    }
  //Deasert CLCK
  FPTE->PCOR   = (1UL << 1);
  }
```

Example of edge detection code:

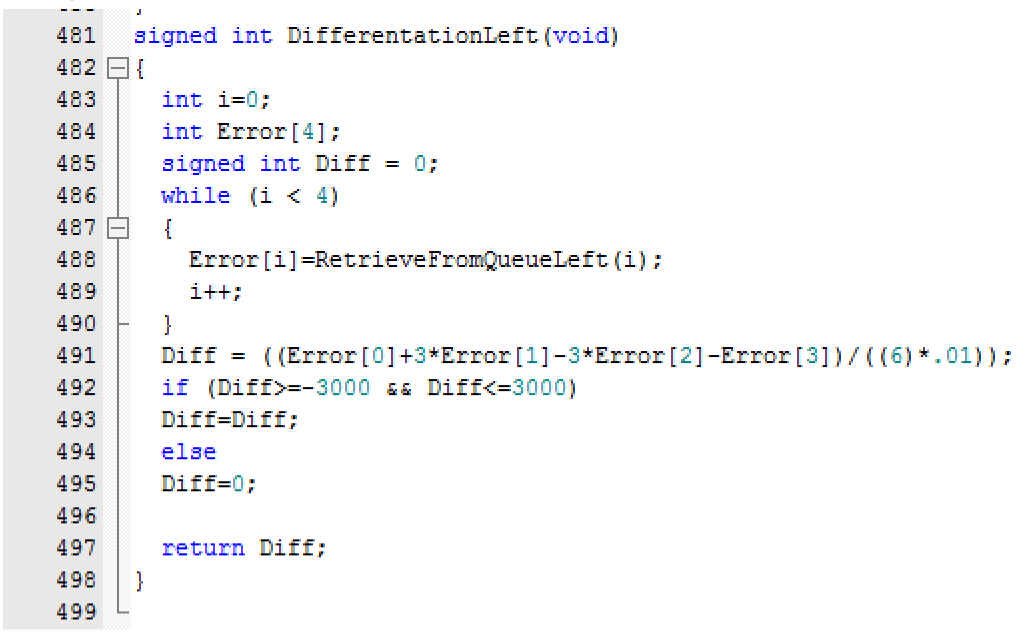


```
int LeftEdgePosition( int CenterStr[128])
{
  signed int data2=0;
  int locMax=0; int high=0;
  int threshold; int i=0; int ErrorL=0x00;
  /////////////////////// Data String Two: Derivative aprox.
    high=0;
  i=2;
  while (i<127)
  {
  data2=((CenterStr[i+1]-CenterStr[i-1])/2);
    if (data2>high)
    {
      high=data2;
      locMax=i;
    }
  i++;
  }

  if( high < 6)
  {ErrorL=0x00;}
  else
  {
  ErrorL=locMax;
  }
  if (ErrorL > 3)
  PushToQueueLeft(ErrorL);
  else
  PushToQueueLeft(0);

  return ErrorL;
  //CenterStr=0;
}
```



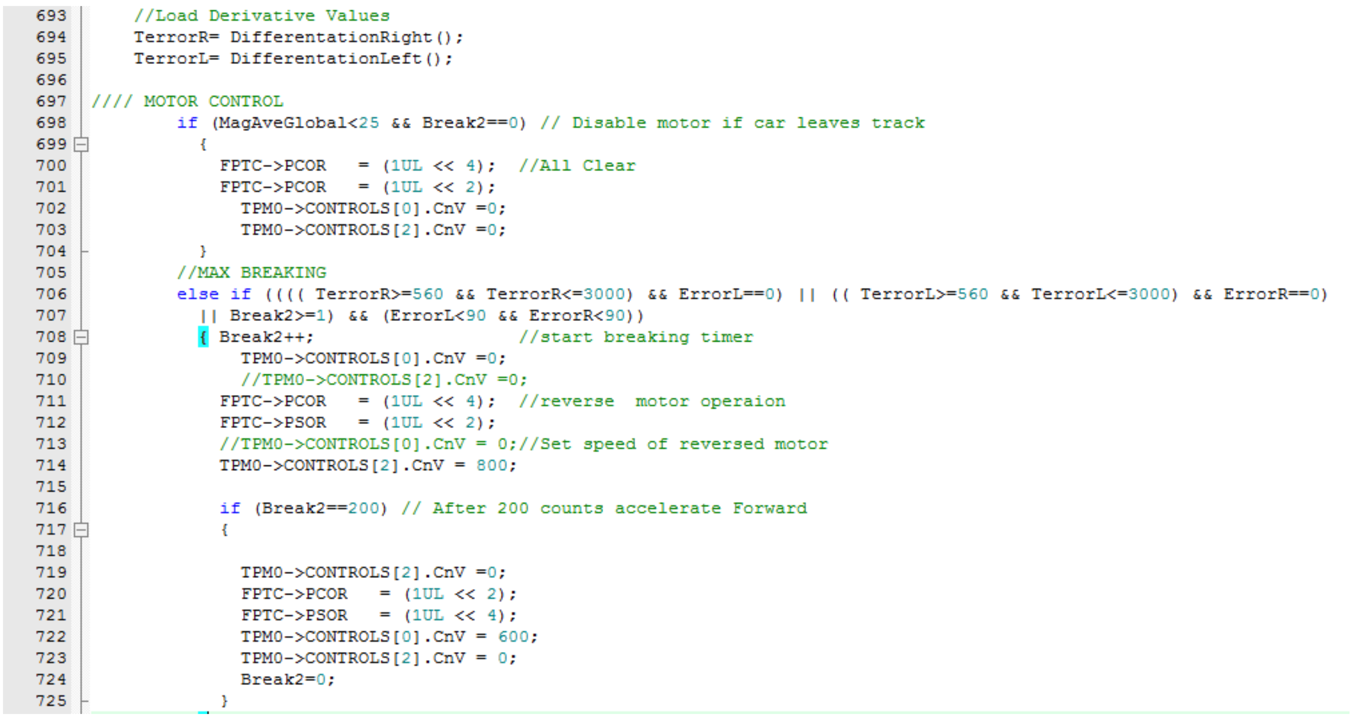
```c
int RightEdgePosition( int CenterStr[128])
{
  signed int data2=0;
  int locMin=0; int low=0xFF;
  int i=0; int Error=0x00;
  int MagAve=0;
  //////////////////////// Data String Two: Derivative aprox.
  low=0xFF;
  i=2;
  while (i<127)
  {
    MagAve=MagAve+(CenterStr[i])/5;
  data2=(CenterStr[i+1]-CenterStr[i-1])/2;
    if (data2<low)
    {
      low=data2;
      locMin=i;
    }
  i++;
  }
  MagAveGlobal=MagAve/25;
  MagAve=0;
  if(low > -6)
    Error=0x00;
  else
  {
  Error = (129-locMin);
  }
  if (Error > 3)
  PushToQueueRight(Error);
  else
  PushToQueueRight(0);

  return Error;
  //CenterStr=0;
}
```

Example of derivative of error function:

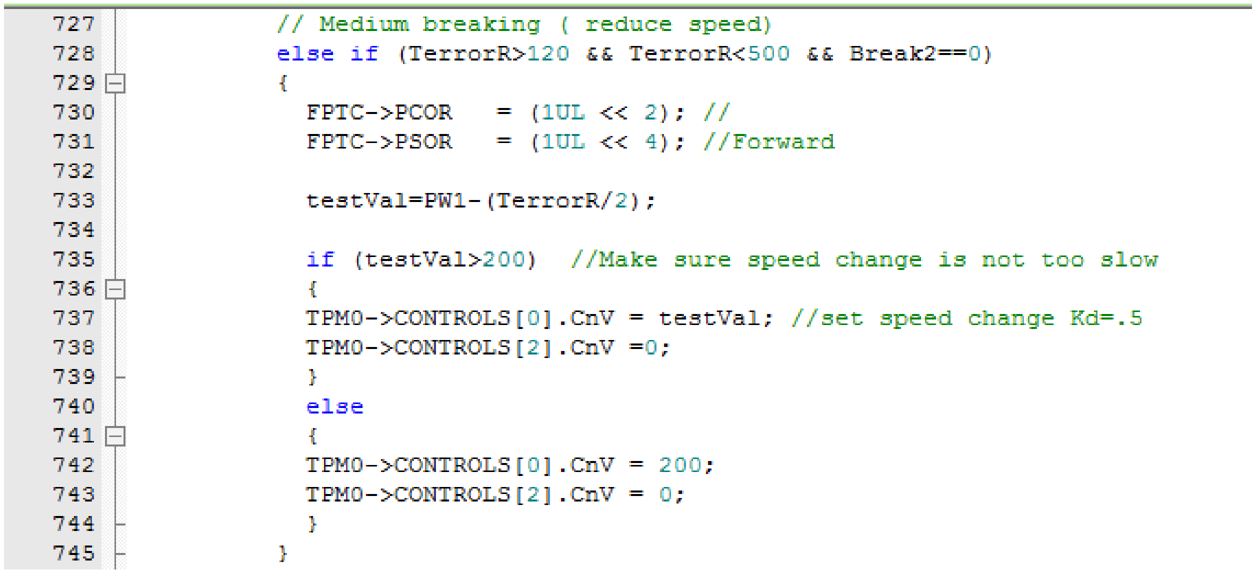


```
signed int DifferentationLeft(void)
{
  int i=0;
  int Error[4];
  signed int Diff = 0;
  while (i < 4)
  {
    Error[i]=RetrieveFromQueueLeft(i);
    i++;
  }
  Diff = ((Error[0]+3*Error[1]-3*Error[2]-Error[3])/((6)*.01));
  if (Diff>=-3000 && Diff<=3000)
  Diff=Diff;
  else
  Diff=0;

  return Diff;
}

```

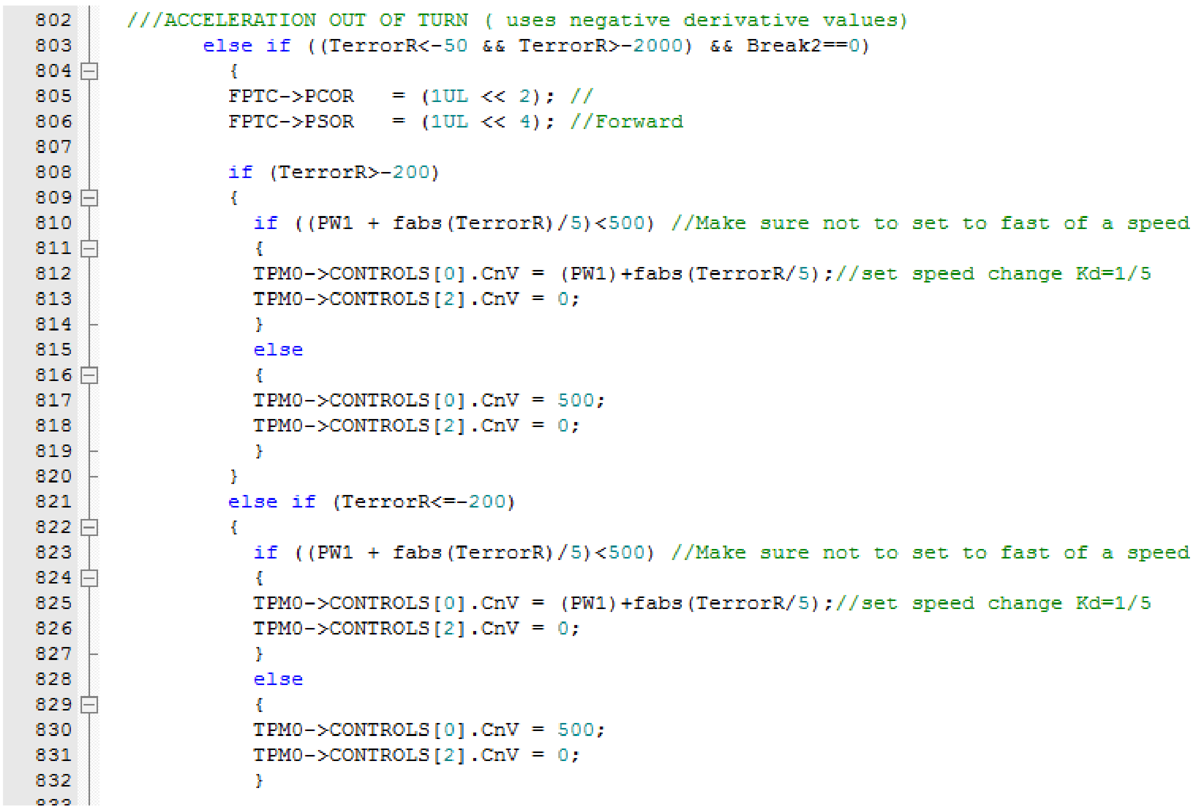
Example of motor control code (breaking/reducing speed):



```
    //Load Derivative Values
    TerrorR= DifferentationRight();
    TerrorL= DifferentationLeft();

//// MOTOR CONTROL
        if (MagAveGlobal<25 && Break2==0) // Disable motor if car leaves track
          {
            FPTC->PCOR   = (1UL << 4);  //All Clear
            FPTC->PCOR   = (1UL << 2);
              TPM0->CONTROLS[0].CnV =0;
              TPM0->CONTROLS[2].CnV =0;
          }
        //MAX BREAKING
        else if (((( TerrorR>=560 && TerrorR<=3000) && ErrorL==0) || (( TerrorL>=560 && TerrorL<=3000) && ErrorR==0)
          || Break2>=1) && (ErrorL<90 && ErrorR<90))
          { Break2++;                    //start breaking timer
              TPM0->CONTROLS[0].CnV =0;
              //TPM0->CONTROLS[2].CnV =0;
            FPTC->PCOR   = (1UL << 4);  //reverse  motor operaion
            FPTC->PSOR   = (1UL << 2);
            //TPM0->CONTROLS[0].CnV = 0;//Set speed of reversed motor
            TPM0->CONTROLS[2].CnV = 800;

            if (Break2==200) // After 200 counts accelerate Forward
            {

              TPM0->CONTROLS[2].CnV =0;
              FPTC->PCOR   = (1UL << 2);
              FPTC->PSOR   = (1UL << 4);
              TPM0->CONTROLS[0].CnV = 600;
              TPM0->CONTROLS[2].CnV = 0;
              Break2=0;
            }
```

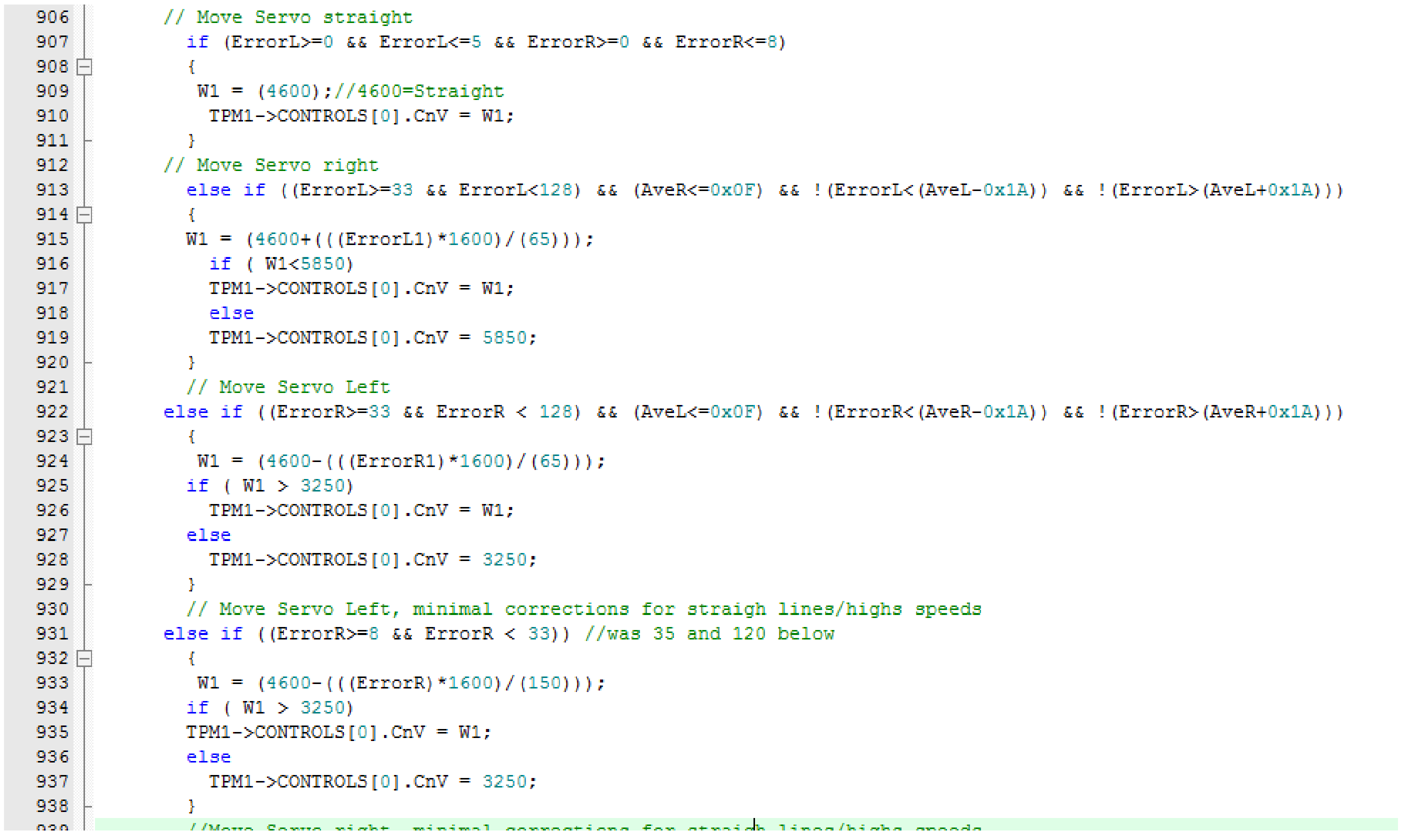


```
          // Medium breaking ( reduce speed)
          else if (TerrorR>120 && TerrorR<500 && Break2==0)
          {
            FPTC->PCOR    = (1UL << 2); //
            FPTC->PSOR    = (1UL << 4); //Forward

            testVal=PW1-(TerrorR/2);

            if (testVal>200)  //Make sure speed change is not too slow
            {
            TPM0->CONTROLS[0].CnV = testVal; //set speed change Kd=.5
            TPM0->CONTROLS[2].CnV =0;
            }
            else
            {
            TPM0->CONTROLS[0].CnV = 200;
            TPM0->CONTROLS[2].CnV = 0;
            }
          }
```



```
///ACCELERATION OUT OF TURN ( uses negative derivative values)
      else if ((TerrorR<-50 && TerrorR>-2000) && Break2==0)
        {
        FPTC->PCOR    = (1UL << 2); //
        FPTC->PSOR    = (1UL << 4); //Forward

        if (TerrorR>-200)
        {
          if ((PW1 + fabs(TerrorR)/5)<500) //Make sure not to set to fast of a speed
          {
          TPM0->CONTROLS[0].CnV = (PW1)+fabs(TerrorR/5);//set speed change Kd=1/5
          TPM0->CONTROLS[2].CnV = 0;
          }
          else
          {
          TPM0->CONTROLS[0].CnV = 500;
          TPM0->CONTROLS[2].CnV = 0;
          }
        }
        else if (TerrorR<=-200)
        {
          if ((PW1 + fabs(TerrorR)/5)<500) //Make sure not to set to fast of a speed
          {
          TPM0->CONTROLS[0].CnV = (PW1)+fabs(TerrorR/5);//set speed change Kd=1/5
          TPM0->CONTROLS[2].CnV = 0;
          }
          else
          {
          TPM0->CONTROLS[0].CnV = 500;
          TPM0->CONTROLS[2].CnV = 0;
          }
```

Example of steering control code:



```
// Move Servo straight
  if (ErrorL>=0 && ErrorL<=5 && ErrorR>=0 && ErrorR<=8)
  {
   W1 = (4600);//4600=Straight
    TPM1->CONTROLS[0].CnV = W1;
  }
// Move Servo right
  else if ((ErrorL>=33 && ErrorL<128) && (AveR<=0x0F) && !(ErrorL<(AveL-0x1A)) && !(ErrorL>(AveL+0x1A)))
  {
  W1 = (4600+(((ErrorL1)*1600)/(65)));
    if ( W1<5850)
    TPM1->CONTROLS[0].CnV = W1;
    else
    TPM1->CONTROLS[0].CnV = 5850;
  }
  // Move Servo Left
else if ((ErrorR>=33 && ErrorR < 128) && (AveL<=0x0F) && !(ErrorR<(AveR-0x1A)) && !(ErrorR>(AveR+0x1A)))
  {
   W1 = (4600-(((ErrorR1)*1600)/(65)));
  if ( W1 > 3250)
    TPM1->CONTROLS[0].CnV = W1;
  else
    TPM1->CONTROLS[0].CnV = 3250;
  }
  // Move Servo Left, minimal corrections for straigh lines/highs speeds
else if ((ErrorR>=8 && ErrorR < 33)) //was 35 and 120 below
  {
   W1 = (4600-(((ErrorR)*1600)/(150)));
  if ( W1 > 3250)
  TPM1->CONTROLS[0].CnV = W1;
  else
    TPM1->CONTROLS[0].CnV = 3250;
  }
  //Move Servo right, minimal corrections for straigh lines/highs speeds
```

<u>Acknowledged and Approved by:</u>

Brian Kong: ____________________

Bruce Krietzer: ____________________

Earnest Sayles III: ____________________

Justin Scaccianoce: ____________________