EEC 195A/B Final Report.

Alisson Li
Kevin Su
Siyuan "Peter" Hu
Thien Nguyen

2014 - 2015

Alisson Li
Kevin Su
Siyuan "Peter" Hu
Thien Nguyen

EEC 195B Report

I. Overview (Executive Summary)

The basic approach to designing our car is to make the car run as fast as possible. We tried our best to get the best camera data we could get by changing the angle, lens and length it was protruding from the chassis. We integrated an aerodynamic and light design in order to reduce any air resistance when the car was running at full speed. We focused on making tight turns at high speeds. Once we have reached the speed limit of our car, where it would go off the track, we would make small changes to our car to make it go faster. We spent a lot of time adjusting our camera angle and gain, so that our car will be able to make tight turns.

Task Breakdown

Servo Control Loop	Alisson Li
Camera Mount/ Configuration	Kevin Su
DC Motor Control Board	Siyuan “Peter” Hu
ADC/ Ping-Pong Buffer	Thien Nguyen

II. Technical Reports

Servo Control Loop Algorithm

by Alisson Li

The car runs through a while statement checking to see if the ping or pong buffer is filled at the same time. The ADC is set to interrupt whenever the data conversion was complete, so that the processor can be analyzing the data of the other buffer. The servo control depends on two elements: the proportional component and the differential component.

The proportional component is the main element in controlling the servo. The equation is as follows:

```
trackloc = center + pingloc1  * turnstat - pingloc2  * turnstat;  //P
trackloc = center + pongloc1  * turnstat - pongloc2  * turnstat;  //P
```

The variables pingloc1 and pongloc1 represent where the left black line is from camera 1 and the variables pongloc1 and pongloc2 represent where the right black line is from camera 2. The trackloc is the value sent to the servo motor, and depends on whether the ping buffer or pong buffer is being analyzed at that time. The variable turnstat is the proportional gain. The car first starts out in the center value, with pingloc1 and pingloc2 negating eachother. To make a left turn, camera 2 would see an increase in its value, meaning that the right edge line is getting closer to the car. Camera 1 would see a decrease in its value, meaning that the left edge line is getting farther from the car. Therefore, the value of trackloc would decrease to a value that depends on how high the value is for camera 2 and how low the value is for camera 1. To make a right turn, the cameras operate in the opposite direction than the previous case.

The differential component helps the servo turn faster at high speeds. The equation is as follows:

```
df = ((trackloc - pastloc) * turning_d) / ig;    //D
```

The previous value of trackloc is stored into the variable pastloc. The variable turning_d is the differential gain. The variable ig (which is set to 100) is used to remove small noise by dividing the differential gain to 0 if the change is small. When the difference between the current value and the previous value is big, then it will be incorporated into the equation that determines the position of the servo motor.

The equations that determines the location of the servo motor is as follows:

```
turn = trackloc + df;    //P + D
 if(turn >5000)
         turn = 5000;
 else if (turn < 3200)
         turn = 3200;
```

The variable turn is the value that is sent to the servo motor. The proportional component dominates on the straight track, while the differential component will have a large part in high speed turns. When the sum of these two components surpasses the limit of how hard to turn the servo, then it saturates to the highest value that the servo motor can turn.

We tried to use an integral term in the control algorithm, but it gave us too many errors. The errors would stack up, and would cause the car to go off the track.

The proportional gain is adjusted so that when the car veers off from the center of the track, it comes back to the center of the track. We calibrated the gain by having the car see the maximum left and right edges of the track, and have the wheels pointing straight along the track. At slow speeds, the proportional gain would be enough. At higher speeds, the car would not be able to turn fast enough. As a result, we added a differential component to handle fast speeds.

The differential gain is adjusted so that the car does not randomly go off the track. The differential component is very sensitive to noise, because it takes small changes that the car sees and multiplies it by the differential gain factor.

We had trouble make the car fast and responsive. The cameras were set to sample at 100 Hz, which is double the recommended sampling rate. The servo motor was set to receive inputs at 50 Hz, which is the recommended rate. There is a notable delay when the car sees a change in the track, and when it turns. To cope with this, I tried to make the ADC data acquisition fast. I eliminated as many if statements as I could, so that the processor can move along its control loop faster. 50 Hz is slow at the processor level, but if we could make the update to the servo motor faster, we would make the rate much higher.

Since the car was a bit heavy, the car had a very hard time making tight turns as high speeds. The car would always go off the track, and then return back to the track and complete the turn. This results in lost time, since the car has to deaccelerate more to complete the turn, and was not taking the fastest route. To alleviate this problem, the car had to turn earlier. I tried to make the car be sensitive to small, consistent changes on the track, which meant raising the proportional gain. As a result, the car would oscillate, and then would make the turns. I had to balance how much time the car would waste on oscillating on the straight track, or on making the turn.

We kept using two cameras to detect the edges of the track, instead of switching to one camera. We had to sample twice as much data, but we would be more accurate in determining where in the track the car was. We decided to use the threshold voltage method for analysis because it was faster than the slope method and was more reliable.

<u>Camera Mount</u>

by Kevin Su

The initial intended goal of the design is to create a design that has a low center of gravity and mimic current automotive aerodynamics to reduce camera shake in order to get the best data possible. Starting up with a clear styrene platform with steel poles sticking out worked great, but the car was slowed down by the weight. The car also needed to be faster and more stable, so I decided to use a thermoplastic to mold the individual supports by hand that hold the car together. This not only made the car stable, it made retrieving the car a much easier process. Knowing that the triangular shape is the strongest geometrical shape, I choose triangles to be the support shape. The design of camera support is then inspired by a bridge structure called a "truss bridge". The triangular spaces on the car also allowed air to pass through so it wouldn't stop the car from reaching its full potential. Through trial and error we realized that to make the sharp turns faster we had to receive the camera data earlier so we can send the signal to the servo in time for the turn. Therefore we extended the camera as far as possible in order to make the turn and have the car stay in the track. There was also a slight delay in the response of the servo, so in pushing the camera far out it also made up for the delay. The maximum camera sampling rate was too low but we tried to make it up with good camera data. Although the camera would look out far and make the turns and intersections on the flat track, when it got up hills it would lose sight of the track and giving it the false sense of a turn. We then attached small angle brackets so we can adjust the angle at which the camera is looking at. The angle is also very important due to the camera's limitations. The camera was viewing too far and since all cameras had a drop off at the end of the set of image data we had to angle it down so it wouldn't mess with our derivative values and disturb our turns. By taking out the first couple pixels of the image data array we fixed that problem. Due to the camera gathering data algorithm, with one camera looking from right to left we had to flip the other camera to basically inverting what it gathered. Therefore one side of the camera mount is higher than the other to make up for the fact that one set of data is for right turns and the other for left turns. A couple weeks into testing the car we realized the performance of the car would increase and decrease without changing the algorithm or the mount. After shining a bright light on the track during some of our test runs we then realized that lighting was a really big issue. Since the amount of charge that was gathered at the pixel through a capacitor was directly related to the light intensity and the integration time and therefore the data; we had to reduce that. I built a camera housing for the back of the camera to reduce the light noise, we also sealed it with electrical tape for extra support. We also had to make sure that we had evenly lighting of the track every time we test the car. In the future we plan on adding a set of LEDs to make sure that the lighting is the same every time we tested it. At one point one of our camera lenses fell out and that is how we figured out that we can further refine our image data from focusing the camera lens. We used an oscilloscope and connected the SI and CO signals and rotated the lens until the black line was clearly seen by the camera and we taped it down using electrical tape. The chassis of the car were giving us some problem due to the fact that our turns were so sharp that our wheel was rubbing the chassis and stopping one of the wheels from turning. We couldn't make any changes to the chassis of the car so we had to limit the max angle of the servo as a trade-off; it wasn't the most ideal, but it made the turns during the class competitions. Although the camera data was good enough the rest depended on the algorithm applied to it. We tested our car with a PID controller and we ended up with using a PD controller the Integral term didn't make much of a difference. Our derivative term took some tweaking to get right since we always ended up with too much noise. To solve that we divided the term by the inverted gain to reduce the noise when the derivative term changes.

Motor Control Board

by Siyuan "Peter" Hu

The objective of the self-made motor control board is to allow us to make custom modifications in addition to the Freescale TFC Shield functionalities and to satisfy the NATcar competition requirement. The motor board itself contains parts like discrete transistors, diodes, capacitors, terminal blocks, 18-pins socket, MAX620 chip. We finished building the motor control board following the lab manual description and the related motor control board schematic posted on SmartSite. Something to keep in mind is to have all the proper capacitors properly connected to the given PCB board. Capacitors in the wrong places might cause shorting the Vcc to GND.

In addition to the motor control board, we found it necessary and convenient to add a separate breadboard to the design. Having a breadboard allows us to have all the Ground(GND) pins properly grounded, give proper voltage(5V) for the processor board, MAX chip and the servo motor. Another key element of having this separate breadboard is to include a 1-ohm resistor to limit the current goes to the DC motor when the switch goes from OFF to ON. On this breadboard we also built connection for both camera pins and servo pins so we can easily identify and test the data received/sent from/to the cameras and servo.

Moreover, we also built a simple circuit using the +5V regulator(LM2940) with a setup of decoupling capacitors to regulate maximum current of our circuit. This is a necessary step because 5V is the recommended voltage for the processor board, MAX chip, and servo motor. Having this +5V regulator circuit allows us to ensure the current going to each key component of our design.

Lastly, in order to have the full functionalities of the Freescale TFC shield, we also built a little circuit board contains two potentiometers and two switches to turn on the motor and servo. To ensure the potentiometers and the switches are working properly, we first tested the behaviors of them using the combination of Oscilloscope, Power supply, Voltmeter, and Function generator.

Tradeoffs & Improvements

There are quite a bit of tradeoffs in our current self-made PCB motor control board. First tradeoff is that we have multiple boards "lying" within our car. We need to have multiple wires to connect the boards together to make our design complete. Although having multiple boards allow us to test out the parts before connecting, multiple boards add weights to the car and could possibly slow down our speed.

The quick bread-boarding prototype saved us quite some time building up the board, but left with lots of wires hanging around, which can be easily come out, create unwanted shorts. Also, the loose wire connections of the breadboard give us lots of noise for the camera data.

Our little circuit board with potentiometers and switches also can be improved. Our current design keep them right next to each other. It allows easy access to them; however, we used the parts from we found in lab. The potentiometers require we use a small screwdriver to turn the knots and the switches are quite small to turn them off and on.

In order to solve these issue, we are looking to design and build our own PCB during Spring break and have it ready prior to the Freescale Cup competition. On this PCB we are going to build, we will have all the functionalities that all the circuit boards had listed above. Having one PCB can have us to decrease the weights of the car and decrease the numbers of loose wires hanging around. On this PCB board, we will purchase big enough potentiometers, which we can turn the knots by our fingers; simple button ON/OFF switches allow us to quickly turn on the motors and ready for the race.

<u>LineScan Camera Configuration:</u>Ping Pong buffer

Specs: About the camera function and how we implemented our program

by Thien Nguyen

Our camera interface has two signals, which are SI and clock. Those two act as output from our processor board and input signal to camera. The SI and Clock signal take in between 0 to 3 volts. The SI single is used to control the integration time of the pixel. The integration time determines how long each pixel is exposed to light. In the dim lighting environment, we would need a relatively long integration time so that the pixels corresponding to white objects reach a sufficient voltage to distinguish from dark objects. Also, control the integration time is extremely useful for varying lighting conditions. On the other hand, we would need short integration time in the bright environment. Since we need to maintain the contrast between light and dark objects, we only require having short integration time. With intense light and long integration time, we risk to saturating all the pixels and losing the contrast between light and dark pixels. Both of the cameras receive the same SI and CLK signals, so their integration time is the same. The minimum integration time is calculated as follow: (128-18)*clock period + 20μs. The 128 is the number of pixels in series and the 18 is the needed logic setup clocks. The 20μs is the pixel charge transfer time. Our clock signal is 7.4μs. The calculation is (128-18) * 7.4 μs + 20 μs = 0.01628s. Basically the minimum integration time is the time it is required to clock out the pixels in addition to the time to discharge the pixels. An increase in minimum integration time would mean a slower clock speed and the maximum light level is reduced. The clock signal is used to shift out each of 128 analog pixel voltages sequentially. For this project, all we have to do is implement the code to generate SI and Clock signal by using a Periodic Interrupt Timer (PIT) interrupt. This PIT interrupt handler will also generate the first CLK cycle and start an A/D conversion for pixel 1. The ADC will generate an interrupt when the conversion is complete. The ADC interrupt handler will read the A/D value and store it in a ping and pong buffer, generate the next clock signal and start the next A/D conversion for pixel 2. Ping-pong buffer is a way to speed up a computer that can overlap I/O processing. Data in one buffer is processed while the next set of data is read into the other one. We implemented ping-pong buffers by first creating 2 buffers, and filling one up at a time, while the other one is being analyzed. The ADC interrupt handler stores the converted values into either an unfilled ping buffer or an unfilled pong buffer. Then it starts another conversion, and stops once the buffer counter (also the clock counter) reaches 128. Once the count reaches 128, it sets the global variables "pingdone" and "readping" to 1, or “pongdone” and “readpong” to 1. The global variables “pingdone” and “pongdone” are 1 when their buffers are full, and 0 when it is ready to be filled, while “readping” and “readpong” are 1 when their buffers are ready to be operated on, and 0 when the average and difference are already calculated. Then, inside the main function, if a buffer is filled, the buffer is operated on, and then clears the “read ping” and “readpong” variables to 0. During this time, the ADC can be collecting data on the other buffer. First, we set up the UART function so that we can control the microcontroller using serial input, and see the serial output on a terminal. Then, we created a ping-pong buffer so that we could read in new values into one buffer and evaluate the values in the other buffer. Once we filled a buffer, the average and the difference between the maximum and minimum was calculated. We calculated the average by summing the values and dividing by 128. You can also divide by 128 by shifting to the right 7 bits. The maximum and minimum were obtained by sweeping through the buffer during the summing process, and using a comparison function. Then, we converted the ADC values into hexadecimal characters, so that the data can be easily read. The lower 4 bits of the 8-bit ADC value were extracted using a mask (0x0F), and the upper 4 bits were extracted by dividing the value by 32.The high pulse signal is on PTB3. The GPIO signal used to measure the conversion time of the 128 pixels goes high for 400 us (microseconds). The period for the PIT interrupts is 600 us when the program is running at its maximum frequency. Thus, the duty cycle of this GPIO signal is 67%. The limiting factor in determining the maximum between the rate of PIT interrupts is length of time the ADC spends taking in values and converting them, and the length of time the main function spends processing the data. One of the most important part is that we have to use TFC shield, which has an interface (J6 header) to the linescan camera. We need to connect the ribbon cable from the J6 header on the TFC shield to the linescan camera. One of the most important things when we set up for the TFC shield is to consider the light noise. To prevent as much noise, we follow the lab instruction by put a piece of black electrical tape across the back of the linescan camera pcb. The length of the ADC conversion for the 2 cameras is 1 ms. The optimal configuration is to have the 2 cameras about 6 inches apart from each other, and 5 inches off the ground. We used the simulated line scan camera program to communicate with the line scan camera. Once we confirmed that the camera works, we tested on the track paper with different SI frequencies and at different heights. For the voltage threshold method, we converted the values into 1 or 0 depending on whether the value was higher or lower than 80% of the maximum value. We calculated the index of the buffer that correlates to the center of the black line by finding the point where the buffer goes from 1 -> 0 and the point where the buffer goes from 0 -> 1, and then calculating the average of the two points. If the camera does not detect a black line, the default index value is set to either 0 or 128, depending on whether the camera is used for the left black line or the right black line of the track. For the slope method, we used the difference equation to calculate the slope, and found the index of the buffer that correlates to the center of the black line by finding the point with the maximum slope. This point describes where the track goes from black to white, or white to black. One of the good things using the slope method is because it is easier the find the center of the black line by finding the maximum slope.

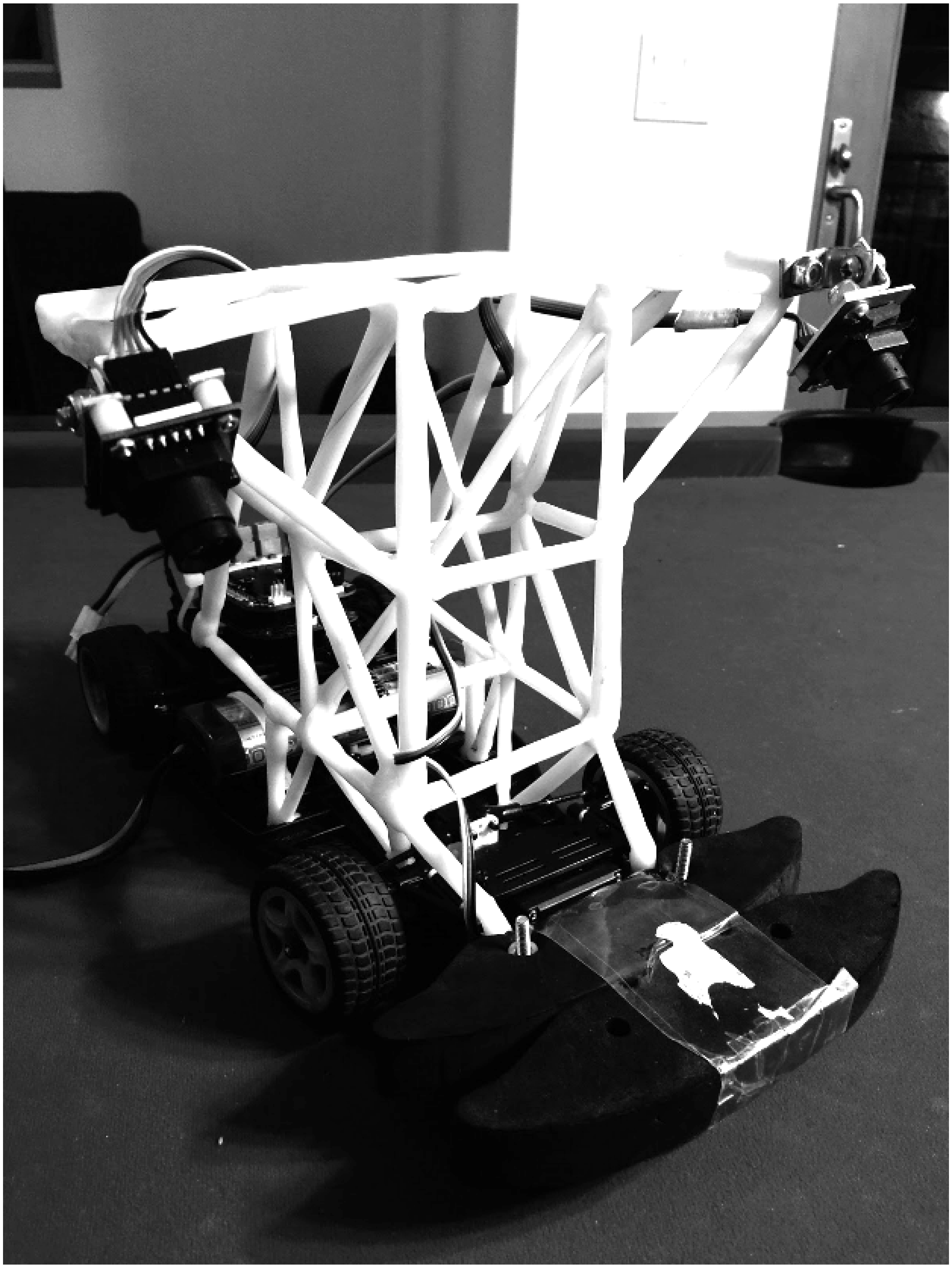
III. Design and Performance Summary

For the first competition, our car clocked in at 23.7 s, which was 4.8 ft/s. For the second competition, our car clocked in at 31.2 s, which was 5.1 ft/s. Our car was heavy due to the amount of plastic on the chassis. We wanted to distribute the weight evenly on the car, but we ended up making it heavier. We may build a lighter mount so that the car can speed up and turn faster. We used wires and a breadboard to hook up our motor control board to the processor. For the next competitions, we are planning on building a PCB board to house all the connections. The algorithm was designed so that while one buffer is being filled by the camera data, the processor would be analyzing the data from the other buffer. At 50 Hz though, it seems that having one buffer is enough, since analyzing the data would not take too long. We might move to one buffer before the Freescale Cup.

IV. Safety

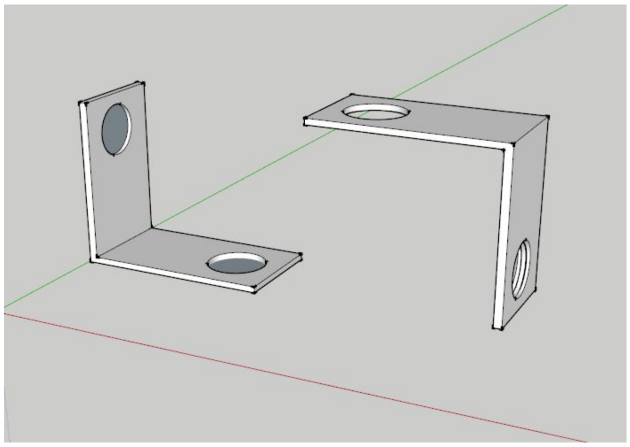
The bumper consists of a bumper that was the length of the camera mount therefore to not damage the camera or anything that it hits. We wanted an easy way to grab our car in case it loses track and runs into people. By using the plastic material on top of the chassis it made it sturdy enough to grab simply by one support. The design of the car created a “handle” shape for easy retrieval of the car in case it loses track of its target. The overall car design was meant to stay low and out of the way but sturdy enough to survive a collision with other cars. We also tapped the wires down and hid it in a cardboard housing so that it wouldn’t tangle with any other car wires or short out any part of our circuit. In the algorithm we tried to put in a stop function for our car when it would run off the track. However, it was unreliable, since it would not stop when the car ran out of the track, and would stop when the car was still on the track. We are planning to add a stop function by the time the Freescale Cup starts.

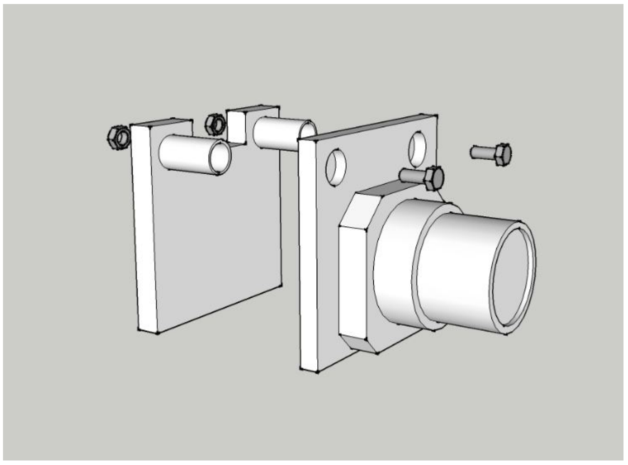
V. Appendixes



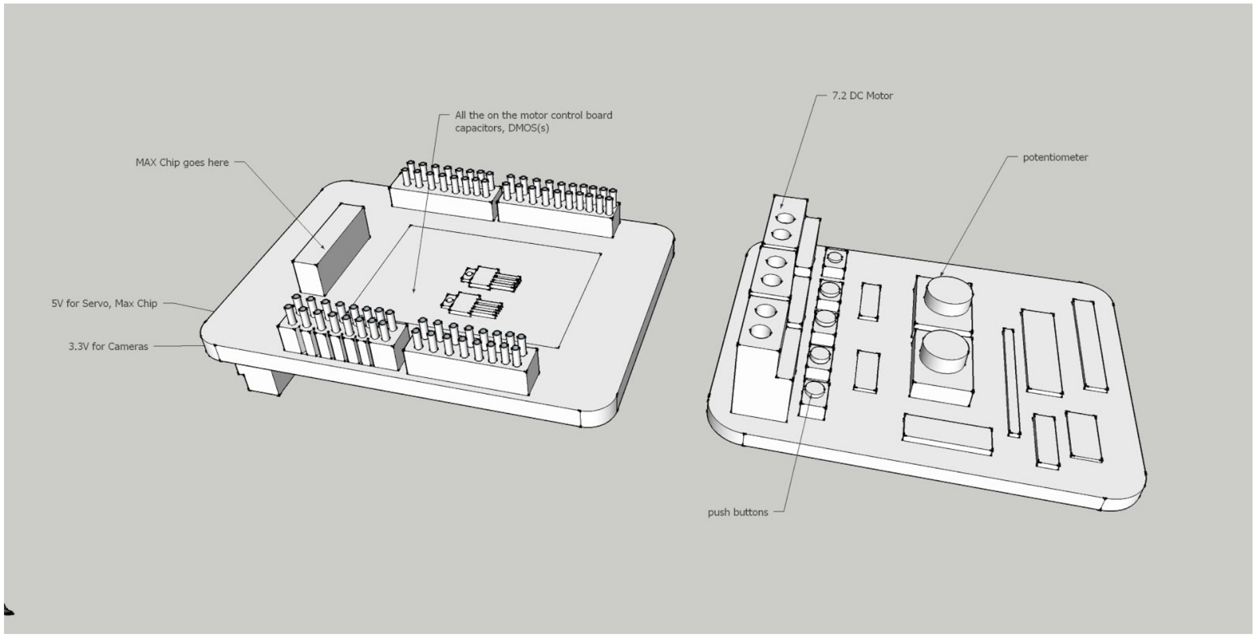
Current camera

- allows us to quickly adjust the camera angles





Design for the new PCB board (To-Be-Finished)



- In order to eliminate the loose wires, we decide to have a similar design to the TFC shield to have the plug & play functionality
- We will have 3 different voltage port(7.2V, 5V, 3.3V) on the PCB board for testing purposes
- In order to protect each component from possible shorting, we will have four different push button switches. This allows the current to only go to the component we are testing and keep everything else safe

Advantages

- light weight
- No loose wires
- Plug and play
- easy to debug each component on the board

Code

```
/* Alisson Li
 Kevin Su
 Siyuan "Peter" Hu
 Thien Nguyen
 */



/*
//PTD5:		ADC channel C1
//PTB3: POT1
//PTB0: PWM output signal for servo motor
//PTB8: Pit interrupt square wave toggle (x2 sampling rate)
PTD7 ->GPIO: SI
PTE1 ->GPIO: CLK
*/

#include "main.h" /* include peripheral declarations */
#include "stdio.h"
#include "stdlib.h"

volatile int period_us = 10000; //100 Hz			//controls pit frequency
volatile char data[128];		//not relevant
volatile unsigned int i = 0;	//for a loop
volatile unsigned int clkcounter = 0;  //counts clock cycle

//1ms = 3000		2ms = 6000
volatile unsigned short PW1;//4300;
volatile unsigned short PW2 = 0;
volatile unsigned short PW3 = 0;
volatile char TPMflag = 0;

volatile char pingc1[128];			// holds pingc1 buffer
volatile int pingdone = 0;	//holds flag if buffer is filled
volatile char pongc1[128];
volatile int pongdone = 0;

volatile int readping = 0;	//holds flag to show if pingc1 buffer is read
volatile int readpong = 0;

volatile int b = 0;
volatile char ping10c1[128];   //holds 1/0 values (voltage threshold)
volatile char pong10c1[128];

volatile char ping10c2[128];   //holds 1/0 values (voltage threshold)
volatile char pong10c2[128];

volatile char pingc1slope[128];         //holds slope value
volatile char pongc1slope[128];         //

volatile char pingc2slope[128];
volatile char pongc2slope[128];

volatile int pingpslope;                //holds positive slope
volatile int pongpslope;
volatile int pingc1maxslope;   //holds maximum slope value
volatile int pongc1maxslope;

volatile int pingc2maxslope;
volatile int pongc2maxslope;


volatile char pingloc1 = 0;
volatile char pinghex[2];
volatile char pongloc1 = 0;
volatile char ponghex[2];

volatile char pingloc2 = 0;
volatile char pongloc2 = 0;

volatile char pingthres[128];
volatile char pongthres[128];


volatile int ping0flagc1 = 0;           //voltage threshold 10 -> 01
volatile int pong0flagc1 = 0;

volatile int ping0flagc2 = 0;
volatile int pong0flagc2 = 0;

volatile int adcamflag = 0;             //0 -> C1; 1 -> C2

volatile char pingc2[128];
volatile char pongc2[128];

volatile char pot[2];
volatile char start = 0;
volatile char end = 0;
volatile unsigned int sw1,sw2;

volatile char A_IFBflag =0;
volatile char B_IFBflag =0;

volatile char IFB[2];
volatile char readIFB = 0;
volatile char IFBdone=1;

volatile int capmode = 1;        //0:slope 1:voltage
volatile int center;             //4300
volatile float turning = 25; //18 how hard to turn                    //48
volatile float turning_d = 60; //90; //how hard to d-turn       //90
volatile int turnstat;
volatile int print = 0; //1:print to terminal 0:don't print
volatile int turn;

volatile int left;
volatile int right;

volatile int trackloc;
volatile int ig = 100;

volatile int df = 0;
//volatile int pastloc[3];
volatile int pastloc;

volatile int pc1read = 0;
volatile int pc2read = 0;

volatile int maxc1 = 0;
volatile int maxc2 = 0;

volatile int driftfunct = 0;

volatile int drift1 = 0;         //increase speed of left wheel
volatile int drift2 = 0; //increase speed of right wheel

volatile int driftgain = 10;
volatile int idrift = 10000;

volatile int sumterm = 0;
volatile int turning_i = 1;
volatile int iig = 100;

volatile int p1stall = 0;
volatile int p2stall = 0;

void LedInitialize(){



  SIM->SCGC5    |= (1UL <<  10) | (1UL <<11) |(1UL <<  12) | (1UL <<13);      /* Enable Clock
to Port B, C, D & E*/
  PORTB->PCR[18] = (1UL <<  8);                    /* Pin PTB18 is GPIO */
  PORTB->PCR[19] = (1UL <<  8);                    /* Pin PTB19 is GPIO */

        PORTB->PCR[8]  = (1UL <<  8);
                        //Pin PTB8 is GPIO
        PORTD->PCR[1]  = (1UL <<  8);                    /* Pin PTD1  is GPIO */
        PORTD->PCR[7]  = (1UL <<  8);
                //Pin PTD7 is GPIO ->SI
        PORTE->PCR[1]  = (1UL <<  8);
                //Pin PTE1 is GPIO ->CLK
        PORTE->PCR[21]  = (1UL << 8);
                        //Pin PTE21 is H-bridge enable

        PORTC->PCR[4] = (1UL << 8);
                                //Pin PTC4 is GPIO = 0
        PORTC->PCR[2] = (1UL << 8);
                                //Pin PTC2 is GPIO = 0
        PORTC->PCR[13] = PORT_PCR_MUX(1);
                //Pin PTC13 is set to input
        PORTC->PCR[17] = PORT_PCR_MUX(1);


  FPTB->PDOR = ((1UL <<19) | (1UL <<18) | (1UL <<3)| (1UL <<8) );          /* switch
Red/Green LED off  */
  FPTB->PDDR = ((1UL <<19) | (1UL <<18) | (1UL <<3)| (1UL <<8) );          /* enable
PTB18/19 as Output */

        FPTD->PDOR = (1UL <<1) | (1UL <<7);            /* switch Blue LED off  */
  FPTD->PDDR = (1UL <<1) | (1UL <<7);            /* enable PTD1 and PTD7as Output */

        FPTE->PDOR = (1UL <<1) | (1UL << 21);
        FPTE->PDDR = (1UL <<1) | (1UL << 21);
```

```
	FPTC->PDDR &= ~(1UL << 13);
	FPTC->PDDR &= ~(1UL << 17);	//Configure as input~(1UL << 17); //Configure as
input

	FPTC->PDOR |= (1UL <<4) | (1UL <<2);
	FPTC->PDDR |= ((1UL <<4) | (1UL <<2));
	FPTC->PCOR |= (1UL <<4) | (1UL <<2);




}

void InitPIT(){
	// PIT module enable

	// Enable clock to PIT module
	SIM->SCGC6 |= SIM_SCGC6_PIT_MASK;

	// Enable module, freeze timers in debug mode
	PIT->MCR &= ~PIT_MCR_MDIS_MASK;
	PIT->MCR |= PIT_MCR_FRZ_MASK;

	// Initialize PIT0 to count down from argument
	PIT->CHANNEL[0].LDVAL = PIT_LDVAL_TSV(period_us*24); // 24 MHz clock
frequency

	// No chaining
	PIT->CHANNEL[0].TCTRL &= PIT_TCTRL_CHN_MASK;

	// Generate interrupts
	PIT->CHANNEL[0].TCTRL |= PIT_TCTRL_TIE_MASK;

	/* Enable Interrupts */
	NVIC_SetPriority(PIT_IRQn, 128); // 0, 64, 128 or 192
	NVIC_ClearPendingIRQ(PIT_IRQn);
	NVIC_EnableIRQ(PIT_IRQn);
}

void PIT_IRQHandler() {
	//clear pending IRQ
	NVIC_ClearPendingIRQ(PIT_IRQn);

	// check to see which channel triggered interrupt
	if (PIT->CHANNEL[0].TFLG & PIT_TFLG_TIF_MASK) {
		// clear status flag for timer channel 0
		PIT->CHANNEL[0].TFLG &= PIT_TFLG_TIF_MASK;

		// Do ISR work - move next sample from buffer to DAC

		FPTD->PSOR = (1UL << 7);						//set SI
PTD7 hi			SI

		clkcounter = 0;
			//delay 20 ns

		FPTE->PSOR = (1UL << 1);						//set
CLK PTE1 hi		CLK

		FPTB->PTOR = (1UL << 8);						//
toggle PTB8


						//delay 120 ns

		FPTD->PCOR = (1UL << 7);						//set SI
PTD7 low

		ADC0->CFG2 = ADC_CFG2_MUXSEL_MASK;
		ADC0->SC1[0] = ADC_SC1_AIEN_MASK  | 0x6;				// start
conversion (software trigger) on AD12 i.e. ADC0_SE12 (PTB2)

									//ADC
interrupt enabled

		//FPTE->PCOR = (1UL << 1);
//set CLK PTE1 low


	} else if (PIT->CHANNEL[1].TFLG & PIT_TFLG_TIF_MASK) {
		// clear status flag for timer channel 1
		PIT->CHANNEL[1].TFLG &= PIT_TFLG_TIF_MASK;
	}
}

void Init_ADC(void) {

	// Calibrate ADC
```

```
        unsigned short cal_var;

        SIM->SCGC6 |= (SIM_SCGC6_ADC0_MASK );    // Enable ADC0 clock

  ADC0->SC2 &=  ~ADC_SC2_ADTRG_MASK ; // Enable Software Conversion Trigger for
Calibration Process    - ADC0_SC2 = ADC0_SC2 | ADC_SC2_ADTRGW(0);
  ADC0->SC3 &= ( ~ADC_SC3_ADCO_MASK & ~ADC_SC3_AVGS_MASK ); // set single
conversion, clear avgs bitfield for next writing
  ADC0->SC3 |= ( ADC_SC3_AVGE_MASK | ADC_SC3_AVGS(0x03) );  // Turn averaging
ON and set at max1 value ( 16 )

  ADC0->SC3 |= ADC_SC3_CAL_MASK ;      // Start CAL
  while ( (ADC0->SC1[0] & ADC_SC1_COCO_MASK ) == 0x00 ); // Wait calibration end


  // Calculate plus-side calibration
  cal_var = 0x00;

  cal_var =  ADC0->CLP0;
  cal_var += ADC0->CLP1;
  cal_var += ADC0->CLP2;
  cal_var += ADC0->CLP3;
  cal_var += ADC0->CLP4;
  cal_var += ADC0->CLPS;

  cal_var = cal_var/2;
  cal_var |= 0x8000; // Set MSB

  ADC0->PG = ADC_PG_PG(cal_var);


  // Calculate minus-side calibration
  cal_var = 0x00;

  cal_var =  ADC0->CLM0;
  cal_var += ADC0->CLM1;
  cal_var += ADC0->CLM2;
  cal_var += ADC0->CLM3;
  cal_var += ADC0->CLM4;
  cal_var += ADC0->CLMS;

  cal_var = cal_var/2;

  cal_var |= 0x8000; // Set MSB
  ADC0->MG = ADC_MG_MG(cal_var);

  ADC0->SC3 &= ~ADC_SC3_CAL_MASK ; /* Clear CAL bit */


          ADC0->CFG1 = (ADC_CFG1_ADLPC_MASK |   0x01);   // 8 bit, Bus clock/2 = 12
MHz
          ADC0->SC2 = 0;                    // ADTRG=0 (software trigger mode)

          //intialize ADC

 ADC0->CFG1 =  ADC_CFG1_MODE(0x00)
            | ADC_CFG1_ADICLK(0x00);       //CFG1
 ADC0->CFG2 = ADC_CFG2_MUXSEL_MASK;

 ADC0->SC2  = ADC_SC2_REFSEL(0x00);

 ADC0->SC3  = 0;

 NVIC_EnableIRQ(ADC0_IRQn);                          //enable interrupt


}

void Init_PWM(void) {

// Set up the clock source for MCGPLLCLK/2.
// See p. 124 and 195-196 of the KL25 Sub-Family Reference Manual, Rev. 3, Sept 2012
// TPM clock will be 48.0 MHz if CLOCK_SETUP is 1 in system_MKL25Z4.c.

          SIM-> SOPT2 |=  (SIM_SOPT2_TPMSRC(0) |SIM_SOPT2_TPMSRC(1) |
SIM_SOPT2_PLLFLLSEL_MASK);

// See p. 207 of the KL25 Sub-Family Reference Manual, Rev. 3, Sept 2012

          SIM->SCGC6 |= SIM_SCGC6_TPM1_MASK | SIM_SCGC6_TPM0_MASK ; // Turn
on clock to TPM1 and TPM0

// See p. 163 and p. 183-184 of the KL25 Sub-Family Reference Manual, Rev. 3, Sept 2012

          PORTB->PCR[0] = PORT_PCR_MUX(3); // Configure PTB0 as TPM1_CH0
          PORTC->PCR[3] = PORT_PCR_MUX(4); // Configure PTC3 as TPM0_CH2
          PORTC->PCR[1] = PORT_PCR_MUX(4); // Configure PTC1 as TPM0_CH2

// Set channel TPM1_CH1 to edge-aligned, high-true PWM and the CHF interrupt
```

```
	TPM1->CONTROLS[0].CnSC = TPM_CnSC_MSB_MASK |
TPM_CnSC_ELSB_MASK | TPM_CnSC_CHF_MASK | TPM_CnSC_CHIE_MASK;
	TPM0->CONTROLS[2].CnSC = TPM_CnSC_MSB_MASK |
TPM_CnSC_ELSB_MASK | TPM_CnSC_CHF_MASK | TPM_CnSC_CHIE_MASK;
	TPM0->CONTROLS[0].CnSC = TPM_CnSC_MSB_MASK |
TPM_CnSC_ELSB_MASK | TPM_CnSC_CHF_MASK | TPM_CnSC_CHIE_MASK;;

// Set period and pulse widths

	TPM1->MOD = 60000-1;		// Freq. = (48 MHz / 16) / 60000 = 50 Hz
	TPM1->CONTROLS[0].CnV = PW1;

	TPM0->MOD = 600-1;		// Freq. = (48 MHz / 16) / 600 = 5000 Hz
	TPM0->CONTROLS[2].CnV = PW2;

	TPM0->MOD = 600-1;		// Freq. = (48 MHz / 16) / 600 = 5000 Hz
	TPM0->CONTROLS[0].CnV = PW3;

// set TPM1 to up-counter, divide by 16 prescaler and clock mode

	//TPM1->SC = (TPM_SC_TOF_MASK | TPM_SC_CMOD(1) | TPM_SC_PS(4));
	TPM1->SC = ( TPM_SC_CMOD(1) | TPM_SC_PS(4));
	TPM0->SC = ( TPM_SC_CMOD(1) | TPM_SC_PS(4));

// clear the overflow mask by writing 1 to TOF

//if (TPM1->SC & TPM_SC_TOF_MASK) TPM1->SC |= TPM_SC_TOF_MASK;
	if (TPM1->CONTROLS[0].CnSC & TPM_CnSC_CHIE_MASK)
		TPM1->CONTROLS[0].CnSC |= TPM_CnSC_CHIE_MASK;

	if (TPM0->CONTROLS[2].CnSC & TPM_CnSC_CHIE_MASK)
		TPM0->CONTROLS[2].CnSC |= TPM_CnSC_CHIE_MASK;

		if (TPM0->CONTROLS[0].CnSC & TPM_CnSC_CHIE_MASK)
		TPM0->CONTROLS[0].CnSC |= TPM_CnSC_CHIE_MASK;


// Enable Interrupts

	NVIC_SetPriority(TPM1_IRQn, 192); // 0, 64, 128 or 192
	NVIC_ClearPendingIRQ(TPM1_IRQn);
	NVIC_EnableIRQ(TPM1_IRQn);

	NVIC_SetPriority(TPM0_IRQn, 192); // 0, 64, 128 or 192
```

```
          NVIC_ClearPendingIRQ(TPM0_IRQn);
          //NVIC_EnableIRQ(TPM0_IRQn);


}

void TPM1_IRQHandler(void) {
//clear pending IRQ
          NVIC_ClearPendingIRQ(TPM1_IRQn);

// clear the overflow mask by writing 1 to TOF

          if (TPM1->CONTROLS[0].CnSC & TPM_CnSC_CHIE_MASK)
                    TPM1->CONTROLS[0].CnSC |= TPM_CnSC_CHIE_MASK;

// modify pulse width for TPM1_CH1
          TPM1->CONTROLS[0].CnV = PW1;


}


void TPM0_IRQHandler(void) {

//clear pending IRQ
          NVIC_ClearPendingIRQ(TPM0_IRQn);

// clear the overflow mask by writing 1 to TOF

          if (TPM0->CONTROLS[2].CnSC & TPM_CnSC_CHIE_MASK)
                    TPM0->CONTROLS[2].CnSC |= TPM_CnSC_CHIE_MASK;

                    if (TPM0->CONTROLS[0].CnSC & TPM_CnSC_CHIE_MASK)
                    TPM0->CONTROLS[0].CnSC |= TPM_CnSC_CHIE_MASK;

          TPM0->CONTROLS[2].CnV = PW2;
          TPM0->CONTROLS[0].CnV = PW3;
}


void ADC0_IRQHandler(void){
		//clear pending IRQ
	NVIC_ClearPendingIRQ(ADC0_IRQn);


		if(pingdone ==0){

		if(adcamflag ==0){
		pingc1[clkcounter]= ADC0->R[0];				// read result register
to pingc1
			adcamflag =1;

			//if(clkcounter<128){
			FPTE->PSOR = (1UL << 1);
//set CLK PTE1 hi
			ADC0->SC1[0] = ADC_SC1_AIEN_MASK  | 7;			// start
conversion (software trigger)
			/*}
			else if(clkcounter==128){
			pingdone = 1;				//pingc1 buffer finished
			readping = 1;				//analyze pingc1 buffer
			readpong = 0;
			pongdone = 0;
			//FPTE->PCOR = (1UL << 1);
	//set CLK PTE1 low
			}
			*/

		}
		else{
			pingc2[clkcounter]= ADC0->R[0];			// read result register
to pingc2
			adcamflag = 0;
			FPTE->PCOR = (1UL << 1);
//set CLK PTE1 low
			if(clkcounter<128){
				clkcounter++;
				ADC0->SC1[0] = ADC_SC1_AIEN_MASK  | 6;
// start conversion (software trigger)
			}
			else if(clkcounter==128){
			pingdone = 1;				//pingc1 buffer finished
			readping = 1;				//analyze pingc1 buffer
			readpong = 0;
			pongdone = 0;
```

```
                    //FPTE->PCOR = (1UL << 1);
        //set CLK PTE1 low
                    }
                    }

        }

        else if (pongdone == 0){

                if(adcamflag ==0){
                pongc1[clkcounter] = ADC0->R[0];                  // read result register to
pongc1
                    adcamflag = 1;
                    //if(clkcounter<128){
                    FPTE->PSOR = (1UL << 1);
//set CLK PTE1 hi
                    ADC0->SC1[0] = ADC_SC1_AIEN_MASK  | 7;                  // start
conversion (software trigger)
                    /*
                    }
                    else if(clkcounter==128){
                    pingdone = 0;                   //pingc1 buffer finished
                    readping = 0;                   //analyze pingc1 buffer
                    readpong = 1;
                    pongdone = 1;
                    //FPTE->PCOR = (1UL << 1);
        //set CLK PTE1 low
                    }
                    */
                }

                else{
                    pongc2[clkcounter] = ADC0->R[0];  //read result register to pongc2
                 adcamflag = 0;
                    FPTE->PCOR = (1UL << 1);
//set CLK PTE1 low
                            if(clkcounter<128){
                            clkcounter++;
                            ADC0->SC1[0] = ADC_SC1_AIEN_MASK  | 6;
// start conversion (software trigger)
                    }
                    else if(clkcounter==128){
                    pingdone = 0;                   //pingc1 buffer finished
                    readping = 0;                   //analyze pingc1 buffer
                    readpong = 1;
```

```
                        pongdone = 1;
                        //FPTE->PCOR = (1UL << 1);
            //set CLK PTE1 low
                        }
                  }

            }

            }


void put(char *ptr_str)
{
            while(*ptr_str)
                  uart0_putchar(*ptr_str++);
}




int main(void){
            int uart0_clk_khz;
            int a = 0;

            //Read buffer variables
            int max1 = 0;
            int max2 = 0;
            char ad;

            int onethc1 = 0;
            int onethc2 = 0;

            center = 4100; //4300


            PW1 = center;
            trackloc = center;
            pastloc = center;
            turnstat = turning;

            pot[0] = 0;
            pot[1] = 0;

            /* Enable the pins for the selected UART */
            /* Enable the UART_TXD function on PTA1 */
```

```
        SIM->SCGC5 |= (SIM_SCGC5_PORTA_MASK
                                                        | SIM_SCGC5_PORTB_MASK
                                                        | SIM_SCGC5_PORTC_MASK
                                                        | SIM_SCGC5_PORTD_MASK
                                                        | SIM_SCGC5_PORTE_MASK );
        SIM->SOPT2 |= SIM_SOPT2_PLLFLLSEL_MASK; // set PLLFLLSEL to select the
PLL for this clock source
        SIM->SOPT2 |= SIM_SOPT2_UART0SRC(1); // select the PLLFLLCLK as UART0
clock source

        PORTA->PCR[1] = PORT_PCR_MUX(0x2);              // Enable the UART0_RX
function on PTA1
        PORTA->PCR[2] = PORT_PCR_MUX(0x2);              // Enable the UART0_TX
function on PTA2

        uart0_clk_khz = (48000000 / 1000); // UART0 clock frequency will equal half the PLL
frequency
        uart0_init (uart0_clk_khz, TERMINAL_BAUD);

        LedInitialize();
        InitPIT();

        PIT->CHANNEL[0].TCTRL |= PIT_TCTRL_TEN_MASK;        //start PIT

        Init_ADC();
        Init_PWM();


        PORTC->PCR[13] = PORT_PCR_MUX(1);
                //Pin PTC13 is set to input
        PORTC->PCR[17] = PORT_PCR_MUX(1);

        FPTC->PDDR &= ~(1UL << 13);
        FPTC->PDDR &= ~(1UL << 17);    //Configure as input~(1UL << 17); //Configure as
input


        while(1) {


                if(start ==0){
                sw1 = FPTC->PDIR&(1UL<<13);
                        if(sw1){
                                start = 1;
```

```
                    //FPTE->PSOR |= (1UL <<21);
                    NVIC_EnableIRQ(TPM0_IRQn);
            }
    }



    else if(start ==1){
                    sw2 = FPTC-> PDIR & (1<<17);
            if(sw2){
            start = 2;
                    NVIC_DisableIRQ(TPM0_IRQn);
                    break;
            }




    ADC0->SC1[0] = 0xD;                     // start conversion (software trigger)
on AD12 i.e. ADC0_SE12 (PTB2)
   while (!(ADC0->SC1[0] & ADC_SC1_COCO_MASK)) {        ; }             // wait for
conversion to complete (polling)
    pot[0] = ADC0->R[0];

                    //hex: diff

    ADC0->SC1[0] = 0xC;
   while (!(ADC0->SC1[0] & ADC_SC1_COCO_MASK)) {        ; }             // wait for
conversion to complete (polling)
    pot[1] = ADC0->R[0];



                    //Potentiometer control

                            PW2 = (pot[0] * 399)/ 0xFF;
                            PW3 = PW2;


                            turnstat = turning/2 + (pot[1] * turning/ 0xFF);


                    }
            }
```

```
          while(1){

                    if(readping == 1){
                              readping = 0;


                    max1 = pingc1[0];
                    max2 = pingc2[0];




                    for(b = 1; b < 128; b++){
                              if(max1 < pingc1[b])
                                                  max1 = pingc1[b];

                              if(max2 < pingc2[b])
                                                  max2 = pingc2[b];

                              else;
          }
                    /*
          if(capmode == 0){

                              pingc1maxslope = 0;
                              pingc2maxslope = 0;
                              pingloc1 = 0;
                              pingloc2 = 0;

                    for(b = 1; b < 128; b++){
                              pingc1slope[b] = (pingc1[b+1] - pingc1[b-1])/2;
// divide by 2 not required
                              pingc2slope[b] = (pingc2[b+1] - pingc2[b-1])/2;
// divide by 2 not required
                              //check for center of black line



                              if ((b>0) && (b<127)){

                                        // ping 1
                                        if((pingc1slope[b] & 0x80) == 0x80)
                                        pingpslope = pingc1slope[b] * -1;

                                                  else
                                        pingpslope = pingc1slope[b];

                                                  if(pingpslope > pingc1maxslope){
                                                            pingc1maxslope = pingpslope;
                                                            pingloc1 = b;
                                                  }

                              //pong c2
                                        if((pingc2slope[b] & 0x80) == 0x80)
                                        pingpslope = pingc2slope[b] * -1;

                                                  else
                                        pingpslope = pingc2slope[b];
                                                  if(pingpslope > pingc2maxslope){
                                                            pingc2maxslope = pingpslope;
                                                            pingloc2 = b;
                                                  }
                              }
                    }
          }

                //voltage threshold

                    if(capmode == 1){
                              */

                              onethc1 = max1 * .8;
                onethc2 = max2 * .8;


                              if((maxc1 < onethc1) && (max1 < 255)){
                              maxc1 = onethc1;
                              }
                              if((maxc2 < onethc2)&& (max2 < 255)){
                maxc2 = onethc2;
                              }

                    //ping0flagc1 =0;
                //ping0flagc2 =0;
                              //pingloc1 = pc1read;
                              //pingloc2 = pc2read;
```

```
                                        for(a = 127; a >= 0; a--){

                                                //calculate voltage threshold of c1
                                                if(pingc1[a] > onethc1){
                                                        ping10c1[a] = 1;

                                         }

                                                else{
                                                        ping10c1[a] = 0;
                                                        if((ping10c1[a+1] == 1) && (ping10c1[a+2] ==
1)&& (ping10c1[a+3] == 1) && (a <123)){
                                                                pingloc1 = a;
                                                                //break;
                                                        }
                                                }
                                        }

                                        for(a = 127; a >= 0; a--){

                                                //calculate voltage threshold of c2

                                                if(pingc2[a] > onethc2){
                                                        ping10c2[a] = 1;


                                         }

                                                else{
                                                        ping10c2[a] = 0;
                                                        if((ping10c2[a+1] == 1) && (ping10c2[a+2] ==
1)&& (ping10c2[a+3] == 1) && (a <123)){
                                                                pingloc2 = a;
                                                                break;
                                                        }

                                                }
                                        }

                                        if(max1 < maxc1 * .35)
                                                pingloc1 = pc1read;


                                        if(max2 < maxc2 * .35)
                                                pingloc2 = pc2read;
```

```
                        pc1read = pingloc1;
                        pc2read = pingloc2;
                    //}

                        if(print == 1){

                        //ping c1
                            ad = pingloc1 & 0x0f;
        //convert index to hex
                        if(ad < 10)
                            pinghex[1] = ad + '0';
                        else
                            pinghex[1] = ad - 10 + 'A';

                        ad = pingloc1 / 16;
                        if(ad < 10)
                            pinghex[0] = ad + '0';
                        else
                            pinghex[0] = ad - 10 + 'A';


                        uart0_putchar(10);
                        uart0_putchar(13);


                        for (a=0; a <2; a++){
                        uart0_putchar(pinghex[a]);                          //print
    out index of black line
                        }
                        uart0_putchar(' ');


                        //ping c2
                            ad = pingloc2 & 0x0f;
        //convert index to hex
                        if(ad < 10)
                            pinghex[1] = ad + '0';
                        else
                            pinghex[1] = ad - 10 + 'A';

                        ad = pingloc2 / 16;
                        if(ad < 10)
                            pinghex[0] = ad + '0';
                        else
```

```
                                pinghex[0] = ad - 10 + 'A';

                        for (a=0; a <2; a++){
                        uart0_putchar(pinghex[a]);                                    //print
out index of black line
                        }
                        uart0_putchar(' ');
        }
                trackloc = center + pingloc1 * turnstat- pingloc2 * turnstat; //P

                df = ((trackloc - pastloc) * turning_d)/ ig;      //D
                turn = trackloc + df;    //P + D
                if(turn >5000)
                        turn = 5000;
                else if (turn < 3200)
                        turn = 3200;

                PW1 = turn;
                pastloc = turn;

                        }

                //Calculate and display avg/diff

                if(readpong == 1){


                readpong = 0;


                max1 = pongc1[0];
                max2 = pongc2[0];

                for(b = 1; b < 128; b++){
                        if(max1 < pongc1[b])
                                        max1 = pongc1[b];

                        if(max2 < pongc2[b])
                                        max2 = pongc2[b];

                }
                /*
                if(capmode == 0){
                        pongc1maxslope = 0;
                        pongc2maxslope = 0;
```

```
                                        pongloc1 = 0;
                                        pongloc2 = 0;

                                        for(b = 1; b < 128; b++){
                                        pongc1slope[b] = (pongc1[b+1] - pongc1[b-1])/2;
// divide by 2 not required
                                        pongc2slope[b] = (pongc2[b+1] - pongc2[b-1])/2;
// divide by 2 not required

                                        if ((b>0) && (b<128)){
                                                      //find index of center of black line

                                               //pong c1
                                               if((pongc1slope[b] & 0x80) == 0x80)
                                               pongpslope = pongc1slope[b] * -1;

                                                      else
                                               pongpslope = pongc1slope[b];


                                                      if(pongpslope > pongc1maxslope){
                                                             pongc1maxslope = pongpslope;
                                                             pongloc1 = b;
                                                      }

                                               //pong c2
                                               if((pongc2slope[b] & 0x80) == 0x80)
                                               pongpslope = pongc2slope[b] * -1;

                                                      else
                                               pongpslope = pongc2slope[b];

                                                      if(pongpslope > pongc2maxslope){
                                                             pongc2maxslope = pongpslope;
                                                             pongloc2 = b;
                                                      }

                                        }

                                 }

                                 }

                                                             //voltage threshold
```

```
                    if(capmode == 1){
                    */


                            onethc1 = max1 * .8;
                      onethc2 = max2 * .8;

                            if((maxc1 < onethc1) && (max1 < 255)){
                            maxc1 = onethc1;
                            }
                            if((maxc2 < onethc2)&& (max2 < 255)){
                      maxc2 = onethc2;
                            }

                    //pong0flagc1 =0;
                      //pong0flagc2 =0;
                            //pongloc1 = pc1read;
                            //pongloc2 = pc2read;



                            for(a = 127; a >=0; a--){



                                    //calculate voltage threshold of c1
                                    if(pongc1[a] > onethc1){
                                            pong10c1[a] = 1;


                             }


                                    else{
                                            pong10c1[a] = 0;
                                            if((pong10c1[a+1] == 1) && (pong10c1[a+2] ==
1)&& (pong10c1[a+3] == 1) && (a <123)){
                                                    pongloc1 = a;
                                                    //break;
                                            }
                                    }
                            }
                            for(a = 127; a >= 0; a--){
                                    //calculate voltage threshold of c2

                                    if(pongc2[a] > onethc2){
                                            pong10c2[a] = 1;
                             }
                                    else{
                                            pong10c2[a] = 0;
                                            if((pong10c2[a+1] == 1) && (pong10c2[a+2] ==
1)&& (pong10c2[a+3] == 1) && (a <123)){
                                                    pongloc2 = a;
                                                    break;
                                            }
                                    }

                            }

                            if(max1 < maxc1 * .35)
                                    pongloc1 = pc1read;

                            if(max2 < maxc2 * .35)
                                    pongloc2 = pc2read;


                            pc1read = pongloc1;
                            pc2read = pongloc2;
                    //}


                            if(print == 1){



                            //pong c1
                            ad = pongloc1 & 0x0f;
            //convert index to hex
                            if(ad < 10)
                                    ponghex[1] = ad + '0';
                            else
                                    ponghex[1] = ad - 10 + 'A';

                            ad = pongloc1 / 16;
                            if(ad < 10)
                                    ponghex[0] = ad + '0';
                            else
                                    ponghex[0] = ad - 10 + 'A';


                            uart0_putchar(10);
                            uart0_putchar(13);
```

```
				for (a=0; a <2; a++){
				uart0_putchar(ponghex[a]);					//print
out index of black line
				}
				uart0_putchar(' ');


				//pong c2
				ad = pongloc2 & 0x0f;
	//convert index to hex
				if(ad < 10)
					ponghex[1] = ad + '0';
				else
					ponghex[1] = ad - 10 + 'A';

				ad = pongloc2 / 16;
				if(ad < 10)
					ponghex[0] = ad + '0';
				else
					ponghex[0] = ad - 10 + 'A';


				for (a=0; a <2; a++){
				uart0_putchar(ponghex[a]);					//print
out index of black line
				}
				uart0_putchar(' ');

	}

			trackloc = center + pongloc1  * turnstat - pongloc2  * turnstat;	//P

			df = ((trackloc - pastloc) * turning_d) / ig;	//D
			turn = trackloc + df;	//P + D
			if(turn >5000)
				turn = 5000;
			else if (turn < 3200)
				turn = 3200;

			PW1 = turn;
			pastloc = turn;

				}
}//MAIN(
```