NATCAR Final Report

EEC195AB

Weijie Zhang

Weitang Liu

Mason Lee

Brian Jae Kim

I. Overview: Executive Summary

The basic approach that we took with splitting work amongst the group was to have a main person in charge of design while taking in the opinions of others. Our approach to software was to have a main person code while the other people provided support and opinions. We found that this was the most efficient way to work because having too many people code at the same time and not being able to merge different peoples' code together made it too difficult and inefficient. In terms of hardware, the same idea applied. We mainly had one person as the main designer for hardware, while the rest provided support and ideas. Virgil Zhang was our main person for software, while Weitang Liu was our main person for hardware. Mason Lee provided support and worked on car construction, while Brian Jae Kim provided support for various tasks and got supplies.

Percentage of Overall Effort for each member:

Weijie Zhang: 30% **Signature:**

Weitang Liu: 30% **Signature:**

Mason Lee: 20% **Signature:**

Brian Jae Kim: 20% **Signature:**

II. Detailed Technical Reports

A) Mason Lee: Camera Data and Current Feedback Capture/Processing

For our camera data capture, we used an ADC0 interrupt handler, which would be triggered by the camera each time there is a new value that needs to be stored and processed. In the interrupt handler, we implemented the conversion system using two Ping-Pong buffers. While our data is being stored in the Ping buffer, it is simultaneously processing the Pong buffer. The same thing is true for when we are storing data in the Pong buffer, it is simultaneously processing the data from the Ping buffer. We chose to implement Ping Pong buffer instead of just using one buffer because it was much more efficient and allowed us to process our code much faster. We implemented this system twice, once with camera1 data and the other with the data from camera 2.

Specifically in our code, depending on the value of the camFlag variable, the data from the camera would go to different ping/pong buffers. For example, if camFlag = 0, the data would go to either the valPing or valPong buffer depending on whether the value of the Done_Flag is 0 or 1. 0 would indicate that it goes to the valPong[ind] buffer, while 1 would mean that it goes to the ValPing[ind] buffer. Ind is the index of our buffer, which increments after every variable is stored. Since the camera reads at 128 pixels, we determined our parameters for index at 128. This would ensure that all data is stored and incremented to prevent overlapping of data. After storing values from camera 1, the code will set it so the next value is from camera 2 and vice versa.

Next, in the case that we are storing values in the buffers designated to camera 1. After storing those values, we will start ADC conversions on the corresponding buffer value on camera 2 with the code ADC0 -> CFG2 |= ADC_CFG2_MUXSEL_MASK; ADC0->SC1[0] = (AIEN_ON | DIFF_SINGLE | ADC_SC1_ADCH(7)); This would start the conversion of the data

that was received from camera 2. If we were storing values from camera 2, we would trigger the conversion of camera 1 data by using the same code as above, but it would be ADC_SC1_ADCH(7) rather than ADC_SC1_ADCH(6).

This allows us to use ping pong buffers to efficiently process and store data. With this, we were able to store values from camera 1 into one buffer, while also process values from buffer2. The same process happens if we assume that we are storing values in camera 2, it will process data from camera 1 at the same time. Once 128 values have been stored in the ping pong 1 and 2 buffers designated to cameras 1 and 2, indicated by whether or not (ind < 128), the same ADC0 Interrupt Handler will start to process the current feedback from the left and right motors.

In our code, when ind >= 128, and IFB_Done == 0, the code will check the left side current feedback from the motor by doing ADC0 -> CFG2 &=~ADC_CFG2_MUXSEL_MASK; ADC0->SC1[0] = (AIEN_ON | DIFF_SINGLE | ADC_SC1_ADCH(7)); , while the second iteration, when IFB_Done == 1, you will do the same code, except with ADC_SC1_ADCH(6) instead of 7. That code will check the right side current feedback. The values from the conversions are then used to decide the torque on the motor, which would indicate whether the car is going uphill or downhill and etc. which is useful information for speed and other changing variables in our code. After both conversions for current feedback are complete, which means that IFB_Done == 2, the values will be reset because it indicates that either the ping or pong buffer are done, hence values are reset and the process starts over again with the data capture from cameras.

B) Weijie Zhang: Servo/Motor Control and Speed Algorithm

For our servo/motor control algorithm, we started off with initializing and configuring PWM on servo and motors. We made them edge-aligned, high true PWM, and enabled channel interrupts. We set the period and pulse widths of TPM, the servos period is 50 Hz, while the motor's period is 5 KHz. Then, we set up the clock to TPM and PWM and made sure to clear the overflow mask by writing 1 to CHF, while enabling interrupts on TPM. In our TPM interrupt handlers, every time the interrupt is triggered, we clear the pending IRQ as well as the overflow mask by writing 1 to CHF.

In the infinite loop of the main function, we update TPM values to PW1, PW2, and PW3 for the motor and servo every time. We then use these values in the speed algorithm to change the speed of the motors and the angle of the servo. The way our speed algorithm works alongside our servo/motor control, is that our car accelerates to a maxspeed and retains that value when it is going straight. When the camera detects an angle, or an edge that requires a turn to be made, then depending on the edge where the line is detected, the motor on the opposite side slows down, while the servo also inputs a signal to turn. The rate at which the motor slows down is correlated to the angle at which the servo is required to turn. If the angle that the servo needs to turn is quite sharp, then the motor will also slow down at a much higher rate. To prevent our car from slowing down every time it detects an edge, we had conditions where if(PW3 > (4500 + SinMoThreshold)), else if(PW3<(4500 - SinMoThreshold)), and if they both are not applicable, which means that the car is in a dead zone that we established, then the car maintains its current speed of max_speed. This allows the car to run at a fast pace without slowing down every time you detect an edge. We also added more safety checks to our car by having additional code to check the values of PW1 and PW2 to make sure that they are within the speeds established at 0 to

max_speed.

We also used ADC to get the current feedback from the motors, which we used to detect the hills that our car would face around the track. If the current feedback on the left and right sides experience current feedback, we compare those values to a threshold current feedback value that would allow us to determine if the car is going on a hill or not. If the car is actually on a hill, the motors would decelerate. We decided to decelerate the motors when high feedback is detected because at the rising edge of the hill, our car would still have the inertia from the straight edge to keep it heading towards the top. When high current feedback is detected, the car is already at the peak and starting to head downhill, which is why we decelerate the motor.

C) Weitang Liu: Control Algorithm

We used the PID controller theory to design our control algorithm. The term PID refers to the use of proportional, integral, and derivative terms to calculate the movement (correction factor). The algorithm of the PID controller theory is as follows:

$$\mathrm{u}(t) = \mathrm{MV}(t) = K_p e(t) + K_i \int_0^t e(\tau)\,d\tau + K_d \frac{d}{dt} e(t)$$

where

K_p: Proportional gain, a tuning parameter
K_i: Integral gain, a tuning parameter
K_d: Derivative gain, a tuning parameter
e: Error $= SP - PV$
t: Time or instantaneous time (the present)
τ: Variable of integration; takes on values from time 0 to the present t.

First, we found the error associated with our values by doing the error = (index of our left camera) - (index of our right camera). Then, we used that value to get the center_average term, which was determined by taking the average of our last 10 error terms. We also used the error array to find the Kd term, which was found by doing 0.5 * (error[0] - error[3])/3 + 0.5 * (error[1] - error[2]). Now that we know the Kd term, to determine the movement value, depending on if error[0] > 0, we do either 4500 + (center_average*center_average/PgLeft + PWConsLeft) *center_average + average[0]*Dg….or we do….500 + (center_average*center_average/PgRight + PWConsRight)*center_average + average[0]*Dg if error[0] > 0. The values of PgRight, PgLeft, Dg, PWConsLeft, PWConsRight are preset at 950, 700, 520, 10, and 15 respectively. After getting the result, we inputted that into the PW3 term, which is used to determine the PW1 and PW2 terms, which are the speeds of each motors. This allows us to implement turning, by speeding up or slowing down one motor/wheel at a time, which allows the car to turn in the desired direction depending on the input from the camera.

The way we determined our PID coefficients was through trial and error. We first implemented our calculations with P, but we saw that our car was not stable and wiggling, so we tried doing P+I, but it turns out that it made our car's reaction time too long (too slow). Then, we tried doing just P+D, without including the "I" term in our calculation for turning and path recognition. This allowed our car to move much more smoothly without the jagged and unstable path that we saw with just P. The P term is determined by the angle of the turn and error, regardless of how long it has been in the turn, for example, if the error is between 0 and 10, then we would have one P, but once we reach a range of 10 to 20, we would require a higher P. The D term, however, is determined by the rate of change in the turn angle. Hence, when the car just enters a turn from a straight line, the D term will be large, however when the car has been in a turn for a long time, the rate of change in the turn angle is basically 0, hence D term becomes insignificant.

D) Brian Jae Kim: Car's Physical Design

For our car's physical design, we designed the car to have the lowest center of gravity possible to make sure that the car will not flip when making sharp turns at a high speed. To do this, we would have to keep everything as compact as possible. We kept this in mind when we decide to put the battery at the back and lowest part of the car. This would keep the car from being flipped because of the heavy back end of the car. Also, a car with a heavy tail will be able to make turns more swiftly when under-steered, hence drifting. Batteries also tend to give off heat after being used for a long time, so we made sure to elevate our control boards and processor so that they are not touching the battery and risk damaging the board.

In order to elevate the board, we found out that the most hassle-free way was using wooden boards and drilling holes which we later used along with metal gauges and screws to secure to the car itself. This allowed a type of platform that was not touching the battery or motor, yet still compact enough to keep the car's center of gravity low to prevent flipping when over-steering or sudden turns. To secure the motor control boards and processor to the platform, we used Velcro tape and Velcro band to secure them solidly. This allowed the board to be very accessible for testing and flashing new code.

Wire management was also a big deal because since all parts of the car needs to be connected to each other in order for it to function. We needed to prevent the wires from hitting the sides of wheel or getting twisted into the wheel while doing test runs. To do this, we used tape or curled it around the wooden stick that we used to secure the cameras.

Camera placement is one of the most design aspects of the entire car physical design. We need the camera to be as high as possible so that we can angle it to see a good 45 degrees in front

of the car. This angle and height will allow the car to see far enough so it does not force our code to react too quickly to turns and sudden changes in the route. If the camera is too low, it would not be able to see the data clearly due to it being too low at an awkward angle. We also set our cameras as far apart from each other as possible so it makes the dead-band or the overlapping pixels from causing our data to become the same. We achieved this by using a wooden stick and attaching it to the metal poles that we fastened the wooden platform with, and extending them to make it as high, yet as stable as possible. We then attached one camera to each of the ends of the pole, while setting it horizontally across the car. By doing this, the overlapping sights of the cameras are as minimal as possible. This allows us to get clear and distinct data for our code to process.

III. Design and Performance Summary

In terms of accomplishments, our NATCAR design was able to meet all checkpoint requirements and complete the track at faster than average times at the competitions. One of the things that we would change if we could, was instead of using PID, we should have just used PD. This is because in our design, we use PD values instead of PID. Also, for our design, we went with using two cameras instead of one. There were some groups that only used one camera and had an easier time because they did not have to deal with certain problems that comes along with two camera detection such as: difference in photo-detector threshold, calibration issues, and dead-band detection/calibration. So if we could do it differently, we may try to use only one camera instead of one. Finally, we would also like to try using 2-D cameras because the resolution is higher and you can see much more and clearer than the cameras that we use right now.

IV. Safety

The steps that we took to ensure that our car would never be a hazard was that if one camera sees black, it would preserve the errors and send it to the servo for it to turn. If both cameras see black, it would signal the motor to stop running, which would prevent the car from running off the track at full speed across the room.

Another safety precaution that we had was that, if the car accidentally hit something without first realizing that it was off the track, the current feedback to the board would be really large, and our program would be able to recognize that and tell the motor to stop, hence keeping it from constantly running against the wall. These functionalities would allow our car to safely run without any hazards. These functions would allow our car to safely run without any hazards.

V. Appendix

```
//Mason Lee
//Weijie Zhang
//Weitang Liu
//Jae Kim


#include "MKL25Z4.h"                    // Device header




volatile unsigned char valIFB_A, valIFB_B;
volatile unsigned char IFB_Done = 0;

volatile char valPing[128], valPong[128];
volatile char valPing2[128], valPong2[128];
volatile unsigned int ind=0;
volatile int counter_1=0;                       /*introduce a global variable of
counter   */
volatile int counter_2=0;                       /*introduce a global variable of
counter   */
volatile int counter_3=0;                       /*introduce a global variable of
counter   */
volatile int counter_print=0;                       /*introduce a global variable
of counter   */
volatile unsigned int counter_CLK=0;
volatile char Done_Flag=0;
// if Done_Flag == 1 : Ping buffer store, Pong buffter anaylze

// if Done_Flag == 0 : Pong buffer store, Ping buffter anaylze
volatile char Done_Ping=0;
volatile char Done_Pong=0;
volatile char camFlag = 0;
// if camFlag == 0 : Read value from Camera #1

// if camFlag == 1 : Read value from Camera #2
volatile signed int sum = 0,center_average = 0,curve=0;  //integration error
volatile signed int error[10] = {0,0,0,0,0,0,0,0,0,0};
//volatile int error2[10] = {0,0,0,0,0,0,0,0,0,0};
volatile signed int average[3] = {0,0,0};
//volatile int average2[3] = {0,0,0};
volatile int counter1=0;                        /*introduce a global variable of
counter   */
volatile int counter2=0;
volatile int badframe = 0;
volatile int left = 0;
volatile int right = 0;
volatile int x = 0;
volatile int index1,index2;


#define voltageThresholdLeft  0x28 //0x39
#define voltageThresholdRight  0x32 //0x39
#define discardLeft  0x05
```

```
#define discardRight  0x03
#define leftRightRatio  4/3

// Weitang's control parameters
#define MAX_SPEED   260
#define SPEED_DIV  0.45
//turning control
#define ERROR  13
#define ERROR2  13

#define Dg  220
#define PgLeft  650 //400
#define PgRight  800 //500
#define PWConsLeft 15 //10
#define PWConsRight 18 //15

#define ErrorNum 2
#define IThreshold 100
#define SinMoThreshold 15


//#define turning_location 25

//#define Pg_turning   16
//#define Turng  0.65
//#define Pg_inside_track 14

//speed control

#define SPg  1
#define P_strait_gain 0.7

volatile unsigned short PW1 = MAX_SPEED;
volatile unsigned short PW2 = MAX_SPEED;
volatile unsigned short PW3 = 4500;

/*-----------------------------------------------------------------------------
  ADC Definations
 *-----------------------------------------------------------------------------*/
#define A                 0x0
#define B                 0x1


/////// NOTE: the following defines relate to the ADC register definitions
/////// and the content follows the reference manual, using the same symbols.


//// ADCSC1 (register)

// Conversion Complete (COCO) mask
#define COCO_COMPLETE     ADC_SC1_COCO_MASK
#define COCO_NOT          0x00

// ADC interrupts: enabled, or disabled.
#define AIEN_ON           ADC_SC1_AIEN_MASK
#define AIEN_OFF          0x00

// Differential or Single ended ADC input
```

```
#define DIFF_SINGLE       0x00
#define DIFF_DIFFERENTIAL ADC_SC1_DIFF_MASK

//// ADCCFG1

// Power setting of ADC
#define ADLPC_LOW         ADC_CFG1_ADLPC_MASK
#define ADLPC_NORMAL      0x00

// Clock divisor
#define ADIV_1            0x00
#define ADIV_2            0x01
#define ADIV_4            0x02
#define ADIV_8            0x03

// Long samle time, or Short sample time
#define ADLSMP_LONG       ADC_CFG1_ADLSMP_MASK
#define ADLSMP_SHORT      0x00

// How many bits for the conversion?  8, 12, 10, or 16 (single ended).
#define MODE_8            0x00
#define MODE_12           0x01
#define MODE_10           0x02
#define MODE_16           0x03

// ADC Input Clock Source choice? Bus clock, Bus clock/2, "altclk", or the
//                                ADC's own asynchronous clock for less noise
#define ADICLK_BUS        0x00
#define ADICLK_BUS_2      0x01
#define ADICLK_ALTCLK     0x02
#define ADICLK_ADACK      0x03

//// ADCCFG2

// Select between B or A channels
#define MUXSEL_ADCB       ADC_CFG2_MUXSEL_MASK
#define MUXSEL_ADCA       0x00

// Ansync clock output enable: enable, or disable the output of it
#define ADACKEN_ENABLED   ADC_CFG2_ADACKEN_MASK
#define ADACKEN_DISABLED  0x00

// High speed or low speed conversion mode
#define ADHSC_HISPEED     ADC_CFG2_ADHSC_MASK
#define ADHSC_NORMAL      0x00

// Long Sample Time selector: 20, 12, 6, or 2 extra clocks for a longer sample
time
#define ADLSTS_20          0x00
#define ADLSTS_12          0x01
#define ADLSTS_6           0x02
#define ADLSTS_2           0x03

////ADCSC2

// Read-only status bit indicating conversion status
#define ADACT_ACTIVE       ADC_SC2_ADACT_MASK
#define ADACT_INACTIVE     0x00
```

```
// Trigger for starting conversion: Hardware trigger, or software trigger.
// For using PDB, the Hardware trigger option is selected.
#define ADTRG_HW           ADC_SC2_ADTRG_MASK
#define ADTRG_SW           0x00

// ADC Compare Function Enable: Disabled, or Enabled.
#define ACFE_DISABLED      0x00
#define ACFE_ENABLED       ADC_SC2_ACFE_MASK

// Compare Function Greater Than Enable: Greater, or Less.
#define ACFGT_GREATER      ADC_SC2_ACFGT_MASK
#define ACFGT_LESS         0x00

// Compare Function Range Enable: Enabled or Disabled.
#define ACREN_ENABLED      ADC_SC2_ACREN_MASK
#define ACREN_DISABLED     0x00

// DMA enable: enabled or disabled.
#define DMAEN_ENABLED      ADC_SC2_DMAEN_MASK
#define DMAEN_DISABLED     0x00

// Voltage Reference selection for the ADC conversions
// (***not*** the PGA which uses VREFO only).
// VREFH and VREFL (0) , or VREFO (1).

#define REFSEL_EXT         0x00
#define REFSEL_ALT         0x01
#define REFSEL_RES         0x02     /* reserved */
#define REFSEL_RES_EXT     0x03     /* reserved but defaults to Vref */

////ADCSC3

// Calibration begin or off
#define CAL_BEGIN          ADC_SC3_CAL_MASK
#define CAL_OFF            0x00

// Status indicating Calibration failed, or normal success
#define CALF_FAIL          ADC_SC3_CALF_MASK
#define CALF_NORMAL        0x00

// ADC to continously convert, or do a sinle conversion
#define ADCO_CONTINUOUS    ADC_SC3_ADCO_MASK
#define ADCO_SINGLE        0x00

// Averaging enabled in the ADC, or not.
#define AVGE_ENABLED       ADC_SC3_AVGE_MASK
#define AVGE_DISABLED      0x00

// How many to average prior to "interrupting" the MCU?  4, 8, 16, or 32
#define AVGS_4             0x00
#define AVGS_8             0x01
#define AVGS_16            0x02
#define AVGS_32            0x03

////PGA

// PGA enabled or not?
```

```
#define PGAEN_ENABLED      ADC_PGA_PGAEN_MASK
#define PGAEN_DISABLED     0x00

// Chopper stabilization of the amplifier, or not.
#define PGACHP_CHOP        ADC_PGA_PGACHP_MASK
#define PGACHP_NOCHOP      0x00

// PGA in low power mode, or normal mode.
#define PGALP_LOW          ADC_PGA_PGALP_MASK
#define PGALP_NORMAL       0x00

// Gain of PGA.  Selectable from 1 to 64.
#define PGAG_1             0x00
#define PGAG_2             0x01
#define PGAG_4             0x02
#define PGAG_8             0x03
#define PGAG_16            0x04
#define PGAG_32            0x05
#define PGAG_64            0x06

/* Uses UART0 for both Open SDA and TWR-SER Tower card */
#define TERM_PORT_NUM       0

#define TERMINAL_BAUD       115200
#undef  HW_FLOW_CONTROL

#define UART_MODE POLLING
#define POLLING                         0
#define INTERRUPT                       1

/* Misc. Defines */
#ifdef  FALSE
#undef  FALSE
#endif
#define FALSE   (0)

#ifdef  TRUE
#undef  TRUE
#endif
#define TRUE    (1)

#ifdef  NULL
#undef  NULL
#endif
#define NULL    (0)

#ifdef  ON
#undef  ON
#endif
#define ON      (1)

#ifdef  OFF
#undef  OFF
#endif
#define OFF     (0)

////////////// The above values fit into the structure below to select ADC/PGA
////////////// configuration desired:
```

```
#include "stdint.h"

typedef struct adc_cfg {
  uint8_t  CONFIG1;
  uint8_t  CONFIG2;
  uint16_t COMPARE1;
  uint16_t COMPARE2;
  uint8_t  STATUS2;
  uint8_t  STATUS3;
  uint8_t  STATUS1A;
  uint8_t  STATUS1B;
  uint32_t PGA;
  } *tADC_ConfigPtr, tADC_Config ;


#define CAL_BLK_NUMREC 18

typedef struct adc_cal {

uint16_t  OFS;
uint16_t  PG;
uint16_t  MG;
uint8_t   CLPD;
uint8_t   CLPS;
uint16_t  CLP4;
uint16_t  CLP3;
uint8_t   CLP2;
uint8_t   CLP1;
uint8_t   CLP0;
uint8_t   dummy;
uint8_t   CLMD;
uint8_t   CLMS;
uint16_t  CLM4;
uint16_t  CLM3;
uint8_t   CLM2;
uint8_t   CLM1;
uint8_t   CLM0;
} tADC_Cal_Blk ;


/*-------------------------------------------------------------------------
  Function that initializes GPIOs
 *-------------------------------------------------------------------------*/
void GPIO_Initialize(void) {

  PORTB->PCR[18] = (1UL <<  8);                    /* Pin PTB18 is GPIO */
  PORTB->PCR[19] = (1UL <<  8);                    /* Pin PTB19 is GPIO */
  PORTD->PCR[1]  = (1UL <<  8);                    /* Pin PTD1  is GPIO */

        PORTB->PCR[0] = (1UL << 8);
/* Pin PTB0 is GPIO */
        PORTB->PCR[1] = (1UL << 8);
/* Pin PTB1 is GPIO */
        PORTB->PCR[3] = (1UL << 8);
/* Pin PTB3 is GPIO */

        PORTD->PCR[7] = (1UL << 8);
/* Pin PTD7 - SI is GPIO*/
```

```
        PORTE->PCR[1] = (1UL << 8);
/* Pin PTE1 - CLK is GPIO*/

        PORTC->PCR[13] = (1UL <<  8);                       /* Pin PTC13 is GPIO */
        PORTC->PCR[17] = (1UL <<  8);                       /* Pin PTC17 is GPIO */
        PORTE->PCR[21] = (1UL <<  8);                       /* Pin PTE21 is GPIO */
        PORTC->PCR[2]  = (1UL <<  8);                       /* Pin PTC2  is GPIO */
        PORTC->PCR[4]  = (1UL <<  8);                       /* Pin PTC4  is GPIO */

        FPTE->PDDR |= (1UL << 21);
/* enable PTE21 as Output */
        FPTC->PDDR |= (1UL << 2 | 1UL << 4);
/* enable PTC2/4 as Output */

        FPTE->PCOR &= ~(1UL << 21);
/* disable H-Bridge */

        FPTB->PDDR |= (1UL << 0);
/* enable PTB0 as Output */
        FPTB->PDDR |= (1UL << 1);
/* enable PTB1 as Output */
        FPTB->PDDR |= (1UL << 3);
/* enable PTB3 as Output */

        FPTD->PDDR |= (1UL << 7);
/* enable PTD7 - SI as Output */
        FPTE->PDDR |= (1UL << 1);
/* enable PTE1 - CLK as Output */

        FPTB->PCOR |= ((1UL << 0) | (1UL << 1) | (1UL << 3));
        FPTD->PCOR |= (1UL << 7);
        FPTE->PCOR |= (1UL << 1);
}


/*-------------------------------------------------------------------------
  ADC Initialization
 *-------------------------------------------------------------------------*/
unsigned char ADC_Cal()
{

  unsigned short cal_var;

  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(AVGS_32) );  // Turn averaging
ON and set at max value ( 32 )


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

  if ((ADC0->SC3& ADC_SC3_CALF_MASK) == CALF_FAIL )
  {
   return(1);    // Check for Calibration fail error and return
```

```
  }
  // 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 */

  return(0);
}

void ADC_Config_Alt(tADC_ConfigPtr ADC_CfgPtr)
{
 ADC0->CFG1 = ADC_CfgPtr->CONFIG1;
 ADC0->CFG2 = ADC_CfgPtr->CONFIG2;
 ADC0->CV1  = ADC_CfgPtr->COMPARE1;
 ADC0->CV2  = ADC_CfgPtr->COMPARE2;
 ADC0->SC2  = ADC_CfgPtr->STATUS2;
 ADC0->SC3  = ADC_CfgPtr->STATUS3;
 //ADC0->PGA  = ADC_CfgPtr->PGA;  pbd
 ADC0->SC1[0]= ADC_CfgPtr->STATUS1A;
 ADC0->SC1[1]= ADC_CfgPtr->STATUS1B;
}


void ADC_Read_Cal(tADC_Cal_Blk *blk)
{
  blk->OFS  = ADC0->OFS;
  blk->PG   = ADC0->PG;
  blk->MG   = ADC0->MG;
  blk->CLPD = ADC0->CLPD;
```

```
  blk->CLPS = ADC0->CLPS;
  blk->CLP4 = ADC0->CLP4;
  blk->CLP3 = ADC0->CLP3;
  blk->CLP2 = ADC0->CLP2;
  blk->CLP1 = ADC0->CLP1;
  blk->CLP0 = ADC0->CLP0;
  blk->CLMD = ADC0->CLMD;
  blk->CLMS = ADC0->CLMS;
  blk->CLM4 = ADC0->CLM4;
  blk->CLM3 = ADC0->CLM3;
  blk->CLM2 = ADC0->CLM2;
  blk->CLM1 = ADC0->CLM1;
  blk->CLM0 = ADC0->CLM0;

}


void init_ADC0(void){

        tADC_Config Master_Adc_Config;

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


// setup the initial ADC default configuration

        Master_Adc_Config.CONFIG1  = ADLPC_NORMAL
              | ADC_CFG1_ADIV(ADIV_8)
              | ADLSMP_LONG
              | ADC_CFG1_MODE(MODE_8)
              | ADC_CFG1_ADICLK(ADICLK_BUS);

        Master_Adc_Config.CONFIG2  = MUXSEL_ADCA
              | ADACKEN_DISABLED
              | ADHSC_HISPEED
              | ADC_CFG2_ADLSTS(ADLSTS_2) ;

        Master_Adc_Config.COMPARE1 = 0x1234u ;  // can be anything
        Master_Adc_Config.COMPARE2 = 0x5678u ;  // can be anything

        Master_Adc_Config.STATUS2  = ADTRG_SW
              | ACFE_DISABLED
              | ACFGT_GREATER
              | ACREN_DISABLED
              | DMAEN_DISABLED
              | ADC_SC2_REFSEL(REFSEL_EXT);

        Master_Adc_Config.STATUS3  = CAL_OFF
              | ADCO_SINGLE
              | AVGE_ENABLED
              | ADC_SC3_AVGS(AVGS_32);

        Master_Adc_Config.STATUS1A = AIEN_ON | DIFF_SINGLE | ADC_SC1_ADCH(31);

// Configure ADC as it will be used, but becuase ADC_SC1_ADCH is 31,
// the ADC will be inactive.  Channel 31 is just disable function.
// There really is no channel 31.
```

```
        ADC_Config_Alt(&Master_Adc_Config);  // config ADC


// Calibrate the ADC in the configuration in which it will be used:

        ADC_Cal();                    // do the calibration

// The structure still has the desired configuration.  So restore it.
// Why restore it?  The calibration makes some adjustments to the
// configuration of the ADC.  The are now undone:

// config the ADC again to desired conditions

        Master_Adc_Config.CONFIG1  = ADLPC_NORMAL
              | ADC_CFG1_ADIV(ADIV_2)
              | ADLSMP_LONG
              | ADC_CFG1_MODE(MODE_8)
              | ADC_CFG1_ADICLK(ADICLK_BUS);

        Master_Adc_Config.STATUS3  = CAL_OFF    // no hardware averaging
              | ADCO_SINGLE;

        ADC_Config_Alt(&Master_Adc_Config);



        ADC0->CFG1 = (ADLPC_LOW | ADIV_1 | ADLSMP_LONG | MODE_8 | ADICLK_BUS_2);
// 8 bit, Bus clock/2 = 12 MHz
        ADC0->SC2 = 0;          // ADTRG=0 (software trigger mode)

}

/*-------------------------------------------------------------------------
  ADC_Handler
 *-------------------------------------------------------------------------*/

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

                if(ind < 128)
                {
                        if(!camFlag)
// Store in Camera #1
                        {
                                FPTE->PSOR = (1UL << 1);
// Set PTE1 - CLK

                                if(Done_Flag)
                                        valPing[ind] = ADC0->R[0];
                                else
                                        valPong[ind] = ADC0->R[0];

                                camFlag = 1;

                                ADC0 -> CFG2 |= ADC_CFG2_MUXSEL_MASK; // select b
channel
```

```
                                ADC0->SC1[0] = (AIEN_ON | DIFF_SINGLE |
ADC_SC1_ADCH(7)); // start conversion on channel SE7b (PTD6)
                        }
                        else
// Store in Camera #2
                        {
                                if(Done_Flag)
                                        valPing2[ind] = ADC0->R[0];
                                else
                                        valPong2[ind] = ADC0->R[0];

                                camFlag = 0;
                                ind++;
// Index Update

                                ADC0 -> CFG2 |= ADC_CFG2_MUXSEL_MASK; // select b
channel
                                ADC0->SC1[0] = (AIEN_ON | DIFF_SINGLE |
ADC_SC1_ADCH(6)); // start conversion on channel SE6b (PTD5)
                        }
                }
                else
                {
                        if(IFB_Done == 2)
                        {
                                valIFB_B = ADC0->R[0];
                                ind = 0;
                                IFB_Done = 0;

                                if(Done_Flag)
// Ping Buffer Done
                                {
                                        Done_Flag = 0;
                                        Done_Ping = 1;
                                }
                                else
// Pong Buffer Done
                                {
                                        Done_Flag = 1;
                                        Done_Pong = 1;
                                }

                                FPTB->PCOR = (1UL << 3);
// Clear GPIO - PTB3
                        }
                        else if(IFB_Done == 0)
                        {
                                counter_CLK = ADC0->R[0];

                                IFB_Done++;

                                ADC0 -> CFG2 &= ~ADC_CFG2_MUXSEL_MASK; // select a
channel
                                ADC0->SC1[0] = (AIEN_ON | DIFF_SINGLE |
ADC_SC1_ADCH(7)); // start conversion on channel SE7a (PTE23)
                        }
                        else if(IFB_Done == 1)
                        {
```

```
                                valIFB_A = ADC0->R[0];

                                IFB_Done++;

                                ADC0->SC1[0] = (AIEN_ON | DIFF_SINGLE |
ADC_SC1_ADCH(3)); // start conversion on channel SE3  (PTE22)
                        }
                }


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

}



/*-------------------------------------------------------------------------
  PIT_Handler
 *-------------------------------------------------------------------------*/
volatile unsigned PIT_interrupt_counter = 0;

void Init_PIT(unsigned period_us) {
        // 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 Start_PIT(void) {
// Enable counter
        PIT->CHANNEL[0].TCTRL |= PIT_TCTRL_TEN_MASK;
}

void Stop_PIT(void) {
// Disable counter
        PIT->CHANNEL[0].TCTRL &= ~PIT_TCTRL_TEN_MASK;
}

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) {

                // Do ISR work - move next sample from buffer to DAC
                FPTD->PSOR = (1UL << 7);
// Set PTD7 - SI

                // 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
                FPTB->PSOR = (1UL << 3);
// Set PTB3 - GPIO
                FPTE->PSOR = (1UL << 1);
// Set PTE1 - CLK

                counter_CLK = counter_CLK + 1;
                counter_CLK = counter_CLK - 1;
                counter_CLK = counter_CLK + 1;
                counter_CLK = counter_CLK - 1;                          // Delay


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

         // Start ADC Conversion
                ADC0 -> CFG2 |= ADC_CFG2_MUXSEL_MASK; // select b channel
                ADC0->SC1[0] = (AIEN_ON | DIFF_SINGLE | ADC_SC1_ADCH(6)); // start
conversion on channel SE6b (PTD5)


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

        } 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;
        }
}

/*****************************************************************************
 * Begin UART0 functions
 *****************************************************************************/
void uart0_init (int sysclk, int baud)
{
    uint8_t i;
    uint32_t calculated_baud = 0;
    uint32_t baud_diff = 0;
    uint32_t osr_val = 0;
    uint32_t sbr_val, uart0clk;
    uint32_t baud_rate;
    uint32_t reg_temp = 0;
    uint32_t temp = 0;
```

```
SIM->SCGC4 |= SIM_SCGC4_UART0_MASK;

// Disable UART0 before changing registers
UART0->C2 &= ~(UART0_C2_TE_MASK | UART0_C2_RE_MASK);

// Verify that a valid clock value has been passed to the function
if ((sysclk > 50000) || (sysclk < 32))
{
    sysclk = 0;
    reg_temp = SIM->SOPT2;
    reg_temp &= ~SIM_SOPT2_UART0SRC_MASK;
    reg_temp |= SIM_SOPT2_UART0SRC(0);
    SIM->SOPT2 = reg_temp;

                      // Enter inifinite loop because the
                      // the desired system clock value is
                      // invalid!!
                      while(1)
                            {}
}
// Verify that a valid value has been passed to TERM_PORT_NUM and update
// uart0_clk_hz accordingly.  Write 0 to TERM_PORT_NUM if an invalid
// value has been passed.
if (TERM_PORT_NUM != 0)
{
    reg_temp = SIM->SOPT2;
    reg_temp &= ~SIM_SOPT2_UART0SRC_MASK;
    reg_temp |= SIM_SOPT2_UART0SRC(0);
    SIM->SOPT2 = reg_temp;

                      // Enter inifinite loop because the
                      // the desired terminal port number
                      // invalid!!
                      while(1)
                            {}
}
// Initialize baud rate
baud_rate = baud;

// Change units to Hz
uart0clk = sysclk * 1000;
// Calculate the first baud rate using the lowest OSR value possible.
i = 4;
sbr_val = (uint32_t)(uart0clk/(baud_rate * i));
calculated_baud = (uart0clk / (i * sbr_val));

if (calculated_baud > baud_rate)
    baud_diff = calculated_baud - baud_rate;
else
    baud_diff = baud_rate - calculated_baud;

osr_val = i;

// Select the best OSR value
for (i = 5; i <= 32; i++)
{
    sbr_val = (uint32_t)(uart0clk/(baud_rate * i));
    calculated_baud = (uart0clk / (i * sbr_val));
```

```
        if (calculated_baud > baud_rate)
            temp = calculated_baud - baud_rate;
        else
            temp = baud_rate - calculated_baud;

        if (temp <= baud_diff)
        {
            baud_diff = temp;
            osr_val = i;
        }
    }

    if (baud_diff < ((baud_rate / 100) * 3))
    {
        // If the OSR is between 4x and 8x then both
        // edge sampling MUST be turned on.
        if ((osr_val >3) && (osr_val < 9))
            UART0->C5|= UART0_C5_BOTHEDGE_MASK;

        // Setup OSR value
        reg_temp = UART0->C4;
        reg_temp &= ~UART0_C4_OSR_MASK;
        reg_temp |= UART0_C4_OSR(osr_val-1);

        // Write reg_temp to C4 register
        UART0->C4 = reg_temp;

        reg_temp = (reg_temp & UART0_C4_OSR_MASK) + 1;
        sbr_val = (uint32_t)((uart0clk)/(baud_rate * (reg_temp)));

         /* Save off the current value of the uartx_BDH except for the SBR field
*/
        reg_temp = UART0->BDH & ~(UART0_BDH_SBR(0x1F));

        UART0->BDH = reg_temp |  UART0_BDH_SBR(((sbr_val & 0x1F00) >> 8));
        UART0->BDL = (uint8_t)(sbr_val & UART0_BDL_SBR_MASK);

#if UART_MODE == INTERRUPT
        UART0->C2 |= UART_C2_RIE_MASK;
#endif

        /* Enable receiver and transmitter */
        UART0->C2 |= (UART0_C2_TE_MASK | UART0_C2_RE_MASK );
    }
    else
                {
        // Unacceptable baud rate difference
        // More than 3% difference!!
        // Enter infinite loop!
        while(1)
                                {}
                }

}

char uart0_getchar()
{
```

```
      /* Wait until character has been received */
      while (!(UART0->S1 & UART0_S1_RDRF_MASK));

      /* Return the 8-bit data from the receiver */
      return UART0->D;
}

void uart0_putchar (char ch)
{
      /* Wait until space is available in the FIFO */
      while(!(UART0->S1 & UART0_S1_TDRE_MASK));

      /* Send the character */
      UART0->D = (uint8_t)ch;

 }
/*********************************************************************/
/*
 * Check to see if a character has been received
 *
 * Parameters:
 *  channel      uart channel to check for a character
 *
 * Return values:
 *  0       No character received
 *  1       Character has been received
 */
int uart0_getchar_present ()
{
    return (UART0->S1 & UART0_S1_RDRF_MASK);
}
/*********************************************************************/
#if UART_MODE == INTERRUPT
void UART0_IRQHandler (void)
{
  char c = 0;
  if (UART0->S1&UART_S1_RDRF_MASK)
  {
    c = UART0->D;

    if ((UART0->S1&UART_S1_TDRE_MASK)||(UART0->S1&UART_S1_TC_MASK))
    {
        UART0->D  = c;
    }
  }
}
#endif


/*********************************************************************
*****                      TPM Handler                          *****
*********************************************************************/
void TPM0_IRQHandler(void) {
//clear pending IRQ
        NVIC_ClearPendingIRQ(TPM0_IRQn);

// clear the overflow mask by writing 1 to CHF
```

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

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

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

// clear the overflow mask by writing 1 to CHF

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

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(1) | SIM_SOPT2_PLLFLLSEL_MASK);

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

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

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

        PORTC->PCR[1] = PORT_PCR_MUX(4); // Configure PTC1 as TPM0_CH0
        PORTC->PCR[3] = PORT_PCR_MUX(4); // Configure PTC3 as TPM0_CH2
        PORTB->PCR[0] = PORT_PCR_MUX(3); // Configure PTB1 as TPM1_CH0

// Set channel TPM0_CH0 to CHIE, edge-aligned, high-true PWM

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

// Set period and pulse widths

        TPM0->MOD = 600-1;              // Freq. = (48 MHz / 16) / 600 = 5000 Hz
        TPM0->CONTROLS[0].CnV = PW1;
        TPM0->CONTROLS[2].CnV = PW2;

        TPM1->MOD = 60000-1;            // Freq. = (48 MHz / 16) / 60000 = 50 Hz
        TPM1->CONTROLS[0].CnV = PW3;

// set TPM0 to up-counter, divide by 16 prescaler and clock mode
```

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

// clear the overflow mask by writing 1 to CHF

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

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

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

// Enable Interrupts

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

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


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

void printHexAscii(char value)
{
        char val1, val2;

        val1 = value & (0xF0);
        val1 = val1 >> 4;
        val2 = value & (0x0F);

        if(val1 < 10)
                uart0_putchar('0' + val1);
        else
                uart0_putchar('A' + val1 - 0x0A);

        if(val2 < 10)
                uart0_putchar('0' + val2);
        else
                uart0_putchar('A' + val2 - 0x0A);

}
void control(void)
{
        int i;

//      sum = sum + error[0];
```

```
                                        // d control
                                        center_average =
(error[1]+error[2]+error[3]+error[4]+error[5]+error[6]+error[7]+error[8]+error[9]+
error[0])/10;
                                                for(i = 9; i > 0; i--)
                {
                        error[i] = error[i - 1];
                }


/*

//
if((center_average> ERROR2) || (center_average < -ERROR2))
                                                                        {
//
PW1 = MAX_SPEED;
//
PW2 = MAX_SPEED;
                                                                        }
//                                                                      else
                                                                        {
//
if(center_average > 0)
                                                                                {
//
PW1 = MAX_SPEED - (128-center_average)*(128-center_average) / 130;
//
PW2 = MAX_SPEED - (128-center_average)*(128-center_average) / 130;
                                                                                }
//
else
                                                                                {
//
PW1 = MAX_SPEED - center_average*center_average / SPEED_DIV;
//
PW2 = MAX_SPEED - center_average*center_average / SPEED_DIV;
                                                                                }
                                                                        }
*/                                                      ////////////////////////////////
//////////////////////////


                                                                if((index1 ==
0)&&(index2 == 0))
                                                                {

                                                                        PW3 = 4500
+ (x * x / PgLeft) * x;

                                                                        if(PW3 >
6000 && error[0] < 0)
                                                                        {

PW3 = 3200;
                                                                        }
                                                                        else
if(PW3 < 3000)
```

```
PW3 = 3200;
                                                                        else
if(PW3 > 6000)
PW3 = 5800;
                                                                    }
                                                                    else if((index1 ==
0)&&(index2 == (0x7F*leftRightRatio)))
                                                                    {
                                                                    PW3 = 4500 + (x *
x / PgRight) * x;
                                                                        if(PW3 <
3000)
PW3 = 3200;
                                                                        else
if(PW3 > 6000)
PW3 = 5800;
                                                                    }
                                                                    else if((index1 !=
0) || (index2 != 0) || (index2 != (0x7F*leftRightRatio)))
                                                                    {

if((center_average > ERROR) || (center_average < -ERROR2))
                                                                        {
                                                                            x
= error[ErrorNum];
                                                                            //
d term
average[0] = 0.5 * (error[0] - error[3])/3 + 0.5 * (error[1] - error[2]);

if(error[0]>0)
PW3 = 4500 + (center_average*center_average/PgLeft + PWConsLeft)*center_average +
average[0]*Dg;
else
PW3 = 4500 + (center_average*center_average/PgRight + PWConsRight)*center_average
+ average[0]*Dg;

if(PW3 > 6000 && error[0] < 0)
                                                                            {
PW3 = 3200;
                                                                            }
else if(PW3 < 3000)
PW3 = 3200;
```

```
else if(PW3 > 6000)

PW3 = 5800;

                                                                        }
                                                                        else


PW3 = 4500;
                                                                }


//                                                              if(PW1 > MAX_SPEED
|| PW2 > MAX_SPEED)
//                                                              if((index1 ==
0)&&(index2 == 127))
//                                                                      PW1 = 0;
//                                                                      PW2 = 0;
//                                                              else
                                                                {

//
PW1 = MAX_SPEED;// - center_average*center_average / SPEED_DIV;
//
PW2 = MAX_SPEED;// - center_average*center_average / SPEED_DIV;
                                                                }


                                                              if(PW3 > (4500 +
SinMoThreshold))
                                                                {

if(MAX_SPEED > x/SPEED_DIV)

PW1 = MAX_SPEED - x / SPEED_DIV;
                                                                        PW2 =
MAX_SPEED;
                                                                }
                                                              else if(PW3 <
(4500 - SinMoThreshold))
                                                                {

if(MAX_SPEED > (- x/SPEED_DIV))

PW2 = MAX_SPEED + x / SPEED_DIV;
                                                                        PW1 =
MAX_SPEED;
                                                                }
                                                                else
                                                                {
                                                                        PW1 =
MAX_SPEED;
                                                                        PW2 =
MAX_SPEED;
                                                                }


                                                              if(PW1 >
MAX_SPEED)
                                                                        PW1 =
```

```
MAX_SPEED;
                                                                        if(PW2 >
MAX_SPEED)
                                                                                PW2 =
MAX_SPEED;


                                                                        if((valIFB_A >
IThreshold) && (valIFB_B > IThreshold))
                                                                        {
                                                                                PW1=0;
                                                                                PW2=0;
                                                                        }


                                                        printHexAscii(index1);
                                                        uart0_putchar('/');
                                                        printHexAscii(index2);
                                                        uart0_putchar('/');
                                                        printHexAscii(center_average);
                                                        uart0_putchar('/');
                                                        printHexAscii(x);
                                                        uart0_putchar(' ');

}
/*-------------------------------------------------------------------------
 *      Main: Initialize
 *-------------------------------------------------------------------------*/
int main (void) {

        char valPingBinaryVT[128], valPongBinaryVT[128];   // Voltage Threshold
Buffer of 1 (light) and 0 (dark)
        char valPingBinaryVT2[128], valPongBinaryVT2[128];   // Voltage Threshold
Buffer of 1 (light) and 0 (dark)
        char firstOne, lowZeroIndex, highZeroIndex;

        unsigned int i=0;
        char key;
        int uart0_clk_khz;
        char str[] = "\r\nEnter 'p' to print buffer\r\n";
  char str2[] = "\r\nEnter 'c' to continue or 'q' to quit\r\n";

        SystemCoreClockUpdate();

        /* 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);

  GPIO_Initialize();           /* Initialize the LEDs           */

        init_ADC0();                    // initialize and calibrate ADC0
        Init_PWM();

        Init_PIT(10000);
// count-down period = 100 us - 10KHz

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

        put("Please turn on the power supply and press SW2...\r\n");
        while(!(FPTC->PDIR & (1UL << 17)))
// wait for SW2 pressed
        {
        }

        while(!(FPTC->PDIR & (1UL << 13)))
// while SW1 not pressed
        {
        }

        Start_PIT();

        /* Enable Interrupts */
        NVIC_SetPriority(ADC0_IRQn, 64); // 0, 64, 128 or 192
        NVIC_ClearPendingIRQ(ADC0_IRQn);
        NVIC_EnableIRQ(ADC0_IRQn);

        __enable_irq();

        TPM1->CONTROLS[0].CnV = PW3;

        put(str);


        // Main Loop
        while(1)
        {
                // Parameter Initialization

                firstOne = 0;
                lowZeroIndex = 0;
                highZeroIndex = 0;


                // If Keyboard Input
```

```
                if(uart0_getchar_present())
                {                                                       //here is
very dangerous pay attention
                        key = uart0_getchar();

                        while(key != 'p')
                        {
                                put("\r\nWrong Command!\r\n");
                                key = uart0_getchar();
                        }

                        Stop_PIT();

                        if(Done_Flag)
// Print Pong Buffers
                        {
                                put("\r\nIFB Values:   ");
                                printHexAscii(valIFB_A);
                                put("|");
                                printHexAscii(valIFB_B);

                                put("\r\n\r\n#1 Camera Print Pong Buffer:\r\n");
                                for(i=0; i<128; i++)
                                {
                                        printHexAscii(valPong[i]);
                                        uart0_putchar(' ');
                                }

                                put("\r\n\r\nVoltage Threshold Buffer:\r\n");
                                for(i=0; i<128; i++)
                                {
                                        printHexAscii(valPongBinaryVT[i]);
                                        uart0_putchar(' ');
                                }


                                put("\r\n\r\n#2 Camera Print Pong Buffer:\r\n");
                                for(i=0; i<128; i++)
                                {
                                        printHexAscii(valPong2[i]);
                                        uart0_putchar(' ');
                                }
                                put("\r\n\r\nVoltage Threshold Buffer:\r\n");
                                for(i=0; i<128; i++)
                                {
                                        printHexAscii(valPongBinaryVT2[i]);
                                        uart0_putchar(' ');
                                }

                                                                        //here is very
dangerous pay attention
                        }
                        else
// Print Ping Buffers
                        {
                                put("\r\nIFB Values:   ");
                                printHexAscii(valIFB_A);
                                put("|");
```

```
                printHexAscii(valIFB_B);

                put("\r\n\r\n#1 Camera Print Ping Buffer:\r\n");
                for(i=0; i<128; i++)
                {
                        printHexAscii(valPing[i]);
                        uart0_putchar(' ');
                }

                put("\r\n\r\nVoltage Threshold Buffer:\r\n");
                for(i=0; i<128; i++)
                {
                        printHexAscii(valPingBinaryVT[i]);
                        uart0_putchar(' ');
                }


                put("\r\n\r\n#2 Camera Print Ping Buffer:\r\n");
                for(i=0; i<128; i++)
                {
                        printHexAscii(valPing2[i]);
                        uart0_putchar(' ');
                }
                put("\r\n\r\nVoltage Threshold Buffer:\r\n");
                for(i=0; i<128; i++)
                {
                        printHexAscii(valPingBinaryVT2[i]);
                        uart0_putchar(' ');
                }

        }

        put(str2);

        while(!uart0_getchar_present());
        key = uart0_getchar();

        while(key != 'c' && key != 'q')
        {
                put("\r\nWrong Command!\r\n");
                key = uart0_getchar();
        }
        if(key == 'c')
        {
                put("\r\nContinue.\r\n");
                Start_PIT();
        }
        else if(key == 'q')
        {
                put("\r\nQuit.\r\n");
        }

        }

/*************************************************************************
 *************************************************************************/

        // Main Analysis
```

```
                else
                {
                        if(Done_Ping || Done_Pong)
                        {
                                if(Done_Flag && Done_Pong)
                                {
//                                      Camera #1 Pong Buffer Analysis
                                        for(i=0;i<128;i++)
                                        {
                                                if(valPong[i] <
voltageThresholdLeft)
                                                        valPongBinaryVT[i] = 0;
                                                else
                                                {
                                                        valPongBinaryVT[i] = 1;
                                                        if(!firstOne)
                                                                lowZeroIndex = i;
                                                        firstOne = 1;
                                                }
// Calculate Cam 1 Voltage Binary Buffer
                                        }

                                        index1 = lowZeroIndex;

                                        // Parameter Initialization
                                        firstOne = 0;
                                        lowZeroIndex = 0;

//                                      Camera #2 Pong Buffer Analysis
                                        for(i=0;i<128;i++)
                                        {
                                                if(valPong2[i] <
voltageThresholdRight)
                                                        valPongBinaryVT2[i] = 0;
                                                else
                                                {
                                                        valPongBinaryVT2[i] = 1;
                                                        if(!firstOne)
                                                                lowZeroIndex = i;
                                                        firstOne = 1;
                                                }
// Calculate Cam 1 Voltage Binary Buffer

                                                if(firstOne && valPongBinaryVT2[i]
== 0)
                                                {
                                                                highZeroIndex =
(i-1);
                                                  firstOne = 0;
                                                }

                                        }

                                        index2 = 127 - highZeroIndex;
                                        index2 = index2 * leftRightRatio;


//                                      if(index1 > 0 && index1 <= discardLeft)
```

```
//                                      index1 = 0;
                                    if(index2 > 0 && index2 <= discardRight)
                                      index2 = 0;

                                    error[0] = (index1 - index2);
                                    control();

                                    TPM0->CONTROLS[0].CnV = PW1;
                                    TPM0->CONTROLS[2].CnV = PW2;
                                    TPM1->CONTROLS[0].CnV = PW3;


                                    //Print Values onto terminal
//                                  printHexAscii(index1);
//                                  uart0_putchar('/');
//                                  printHexAscii(index2);
//                                  uart0_putchar(' ');

                                    Done_Pong = 0;
                            }

                          /***********************************************
                           ***********************************************/

                            else
                                    if(Done_Ping)
                                    {
//                                  Camera #1 Ping Buffer Analysis
                                      for(i=0;i<128;i++)
                                      {
                                        if(valPing[i] <
voltageThresholdLeft)
                                          valPingBinaryVT[i]
= 0;
                                        else
                                        {
                                          valPingBinaryVT[i]
= 1;
                                          if(!firstOne)

lowZeroIndex = i;
                                          firstOne = 1;
                                        }
// Calculate Cam 1 Voltage Binary Buffer

                                        if(firstOne &&
valPingBinaryVT[i] == 0)
                                        {

highZeroIndex = (i-1);
                                                  firstOne =
0;
                                        }

                                      }

                                      index1 = lowZeroIndex;
```

```
                                                        // Parameter Initializatio
                                                        firstOne = 0;
                                                        lowZeroIndex = 0;

//                                                  Camera #2 Ping Buffer Analysis
                                                        for(i=0;i<128;i++)
                                                        {
                                                                if(valPing2[i] <
voltageThresholdRight)

valPingBinaryVT2[i] = 0;
                                                                else
                                                                {

valPingBinaryVT2[i] = 1;
                                                                        if(!firstOne)

lowZeroIndex = i;
                                                                        firstOne = 1;
                                                                }
// Calculate Cam 1 Voltage Binary Buffer

                                                                if(firstOne &&
valPingBinaryVT2[i] == 0)
                                                                {
                                                                        highZeroIndex =
(i-1);
                                                                  firstOne = 0;
                                                                }
                                                        }

                                                        index2 = 127 - highZeroIndex;
                                                        index2 = index2 * leftRightRatio;


//                                                      if(index1 > 0 && index1 <=
discardLeft)
//                                                              index1 = 0;
                                                        if(index2 > 0 && index2 <=
discardRight)
                                                                index2 = 0;

                                                        error[0] = (index1 - index2);

                                                        control();

                                                        TPM0->CONTROLS[0].CnV = PW1;
                                                        TPM0->CONTROLS[2].CnV = PW2;
                                                        TPM1->CONTROLS[0].CnV = PW3;

                                                        //Print Values onto terminal
//                                                      printHexAscii(index1);
//                                                      uart0_putchar('/');
//                                                      printHexAscii(index2);
//                                                      uart0_putchar(' ');

                                                        Done_Ping = 0;
                                                }
```

```
                        }
                    }
                }
}
```

Displaying K1065 3-13_Speed-260.c.