Team Finesse
Code Listings

C32 – 4 modules

encoders.h/.c
main.h/.c
motors.h/.c
slave_com.h/.c

encoders.h

```
#ifndef _encoders_h
#define _encoders_h

/** C32 HARDWARE MAPPINGS **/
/*
T7 - Left Encoder
T6 - Right Encoder
*/


/** CONSTANTS **/
#define DEGREES_PER_TICK 19.5

/** PUBLIC PROTOTYPES **/

void Init_Encoders(void);
unsigned char LEncoderDone(void);
unsigned char REncoderDone(void);
void SetLDegrees(unsigned char num_degrees);
void SetRDegrees(unsigned char num_degrees);
unsigned long GetLTicks(void);
unsigned long GetRTicks(void);

//Not to be accessed!
void interrupt _Vec_TIMER6 SignalTimer1(void);
void interrupt _Vec_TIMER7 SignalTimer2(void);

#endif
```

encoders.c

```
#include <stdio.h>
#include <ME218_C32.h>
#include <S12Vec.h>
//#include "ADS12.H"
#include "encoders.h"

/** DATA STRUCTURES **/
static unsigned char L_num_degrees_goal = 0;
static unsigned char R_num_degrees_goal = 0;
//the actual number of degrees must be scaled, thus an int
static unsigned long cur_L_ticks = 0;
static unsigned long cur_R_ticks = 0;

static unsigned int uPeriodL;
static unsigned int uLastEdgeL;
static unsigned int uPeriodR;
static unsigned int uLastEdgeR;

/** MODULE PRIVATE PROTOTYPES **/
static void InitTimers(void);


/** PUBLIC FUNCTIONS **/


/*****
** Function: void Init_Encoders(void)
** Just initializes the timers
*****/
void Init_Encoders(void)
{
    InitTimers();
}

/*****
** Function: void interrupt _Vec_TIMER6 SignalTimer1(void)
** Encoder interrupt.
*****/
void interrupt _Vec_TIMER6 SignalTimer1(void)
{
  cur_R_ticks++;
  uPeriodR = (TC6) - uLastEdgeR;
  uLastEdgeR = TC6;

  //Clear IC6 flag
  TFLG1 = _S12_C6F;
}

/*****
** Function: void interrupt _Vec_TIMER7 SignalTimer2(void)
** Encoder interrupt.
*****/
void interrupt _Vec_TIMER7 SignalTimer2(void)
{
  cur_L_ticks++;
  uPeriodL = (TC7) - uLastEdgeL;
  uLastEdgeL = TC7;

  //Clear IC7 flag
  TFLG1 = _S12_C7F;
}

/*****
** Function: unsigned char LEncoderDone(void)
** Checks to see if the left encoder has ticked more times
** than was specified.  A goal of 0 means run forever.
*****/
unsigned char LEncoderDone(void)
{
    //if no degrees it means run forever
    printf("\r\nCurrent L ticks = %lu", cur_L_ticks);
    if(L_num_degrees_goal == 0) return FALSE;
    if(L_num_degrees_goal <= cur_L_ticks)
    {
        return TRUE;
    } else return FALSE;
}

/*****
** Function: unsigned char REncoderDone(void)
** Checks to see if the right encoder has ticked more times
** than was specified.  A goal of 0 means run forever.
*****/
unsigned char REncoderDone(void)
{
    //if no degrees it means run forever
    printf("\r\nCurrent R ticks = %lu", cur_R_ticks);

    if(R_num_degrees_goal == 0) return FALSE;
    if(R_num_degrees_goal <= cur_R_ticks)
    {
        return TRUE;
    } else return FALSE;
}

/*****
** Function: void SetLDegrees(unsigned char num_degrees)
** Set the goal for number of encoder ticks.  NOT DEGREES
** ANYMORE!
*****/
void SetLDegrees(unsigned char num_degrees)
{
    cur_L_ticks = 0;
    L_num_degrees_goal = num_degrees;
}

/*****
** Function: void SetRDegrees(unsigned char num_degrees)
** Set the goal for number of encoder ticks.  NOT DEGREES
** ANYMORE!
*****/
void SetRDegrees(unsigned char num_degrees)
{
    cur_R_ticks = 0;
    R_num_degrees_goal = num_degrees;
}

/*****
** Function: unsigned long GetLTicks(void)
** Return the number of ticks currently counted by
** the left encoder.
*****/
unsigned long GetLTicks(void)
{
    return cur_L_ticks;
}
```

```
/*****
** Function: unsigned long GetRTicks(void)
** Return the number of ticks currently counted by
** the right encoder.
*****/
unsigned long GetRTicks(void)
{
    return cur_R_ticks;
}

/** PRIVATE FUNCTIONS **/

/*****
** Function: static void InitTimers(void)
** Initialization function for the input capture timers.
** Run at 187,500Hz
*****/
static void InitTimers(void)
{
    //Turn timer system on
    TSCR1 = _S12_TEN;

    //Set pre-scale to /128
    TSCR2 = (_S12_PR2 | _S12_PR0 | _S12_PR1);

    //Set capture/compare 6&7 to input capture
    TIOS &= ~(_S12_IOS6|_S12_IOS7);

    //Set input capture to look at rising edges
    TCTL3 |= _S12_EDG6A;
    TCTL3 &= ~_S12_EDG6B;
    TCTL3 |= _S12_EDG7A;
    TCTL3 &= ~_S12_EDG7B;

    //Clear IC6&7 flag
    TFLG1 = _S12_C6F;
    TFLG1 = _S12_C7F;

    //Enable IC6&7 interrupt
    TIE   |= (_S12_C6I | _S12_C7I);
}
```

main.h

```
#ifndef _main_h
#define _main_h

/** C32 HARDWARE MAPPINGS **/
/*
See individual .h's
*/


/** CONSTANTS **/
#define TRUE 1
#define FALSE 0
#define MOTOR_OFF_BIT BIT0HI
```

```
#define MOTOR_STRAIGHT_BIT BIT1HI
#define MOTOR_FWD_BIT BIT2HI
#define MOTOR_CW_BIT BIT2HI //same bit
#define R_STRAIGHT_DUTY 90
#define L_STRAIGHT_DUTY 90
#define R_TURN_DUTY 90
#define L_TURN_DUTY 90
#define DUTY_MIN 85
#define DUTY_MAX 95
#define SPIN_DUTY_MIN 30
#define SPIN_DUTY_MAX 40
#define L_START_ADJUST 1
#define TICKS_MULTIPLIER 2 //just because the left is a bit slower than
right


#define MASTER_QUERY_COMMAND 0xFF
/** PUBLIC PROTOTYPES **/
/*****
** Function: void main(void)
** Runs the whole shebang on the C32.
*****/
void main(void);


#endif
```

main.c

```
#include <stdio.h>
#include <ME218_C32.h>
#include <S12Vec.h>
#include "ADS12.H"
#include "encoders.h"
//#include "led.h"
#include "motors.h"
#include "slave_com.h"
#include "main.h"

//#define TEST

/** DATA STRUCTURES **/
static char pGain = 30;
static char iGain = 0;
static char dGain = 0;
static char left_done = FALSE;

/** MODULE PRIVATE PROTOTYPES **/
static void Handle_Data(unsigned char data_byte);
static unsigned char GetDegrees(unsigned char data_byte);
static void Handle_Data(unsigned char data_byte);
static void DrivePID(unsigned char num_degrees);
static void SpinPID(unsigned char num_degrees);
static void Init_C32(void);
static void Test_C32(void);

/** PUBLIC FUNCTIONS **/
```

```
/*****
** Function: void main(void)
** After init, it runs the cycle of:
** (1) checking for new data
** (2) if the data is not the query by the E128 for encoders being
done:
** (3) run the motors based on the new command
*****/
void main(void)
{
    unsigned char command;

    Init_C32();

    EnableInterrupts;

    #ifdef TEST

    Test_C32();

    #endif

    printf("\r\nAt start of loop after init on C32");
    while(TRUE)
    {
        //printf("\r\nIn overriding while loop.");
        //send CheckFlag true so the interrupt flag resets to 0

        if(CheckFlag(TRUE))
        {
            command = GetData();

            printf("\r\nCommand received is %u", command);
            //if(CheckDummy()) printf("\r\nlatent interrupt had");
            if(command != MASTER_QUERY_COMMAND)
            {
                Handle_Data(command);
            }
        }
    }
}


/** PRIVATE FUNCTIONS **/

/*****
** Function: static void Test_C32(void)
** Tests drive forward for 3 encoder ticks
*****/
static void Test_C32(void)
{
    int i = 0;

    Handle_Data(0x9C);
}
```

```
/*****
** Function: static unsigned char GetDegrees(unsigned char data_byte)
** Decodes the number of ticks in the command byte from Massuh
*****/
static unsigned char GetDegrees(unsigned char data_byte)
{
    unsigned char num_degrees = 0;
    if(data_byte & BIT3HI)
    {
        num_degrees += 1;
    }
    if(data_byte & BIT4HI)
    {
        num_degrees += 2;
    }
    if(data_byte & BIT5HI)
    {
        num_degrees += 4;
    }
    if(data_byte & BIT6HI)
    {
        num_degrees += 8;
    }
    if(data_byte & BIT7HI)
    {
        num_degrees += 16;
    }
    return num_degrees*TICKS_MULTIPLIER;
}

/*****
** Function: static void Handle_Data(unsigned char data_byte)
** Decodes the command byte from Massuh and runs Drive/Turn_PID
*****/
static void Handle_Data(unsigned char data_byte)
{
    unsigned char num_degrees;
    //check for stop motors first
    if(data_byte & MOTOR_OFF_BIT)
    {
        printf("\r\nTold to Stop Motors by E128");
        StopMotors();
        return;
    }


    num_degrees = GetDegrees(data_byte);
    //printf("\r\nNum Degrees Sent is %u", num_degrees);

    SetLDegrees(num_degrees);
    SetRDegrees(num_degrees);

    if(data_byte & MOTOR_STRAIGHT_BIT)
    {
        //do we need to turn duty to 0 before initializing?
        if(data_byte & MOTOR_FWD_BIT)
        {
            printf("\r\nTold to go FWD by E128 %u degrees",
num_degrees);
            ForwardR();
            ForwardL();
        } else
        {
            printf("\r\nTold to go RRV by E128 %u degrees",
num_degrees);
            ReverseR();
            ReverseL();
        }

        //drive her straight
        DrivePID(num_degrees);
        //SetRDuty(R_STRAIGHT_DUTY);
        //SetLDuty(L_STRAIGHT_DUTY);
    } else
    {
        if(data_byte & MOTOR_CW_BIT)
        {
            printf("\r\nTold to spin CW by E128 %u degrees",
num_degrees);
            ForwardL();
            ReverseR();
        } else
        {
            printf("\r\nTold to spin CCW by E128 %u degrees",
num_degrees);
            ForwardR();
            ReverseL();
        }

        //turn on a dime
        SpinPID(num_degrees);
        //SetRDuty(R_TURN_DUTY);
        //SetLDuty(L_TURN_DUTY);
    }
}

/*****
** Function: static void DrivePID(unsigned char num_degrees)
** PID control based on encoder ticks for straight
** drive of the motors.  NOTE WE ENDED UP DOING JUST
** PROPORTIONAL CONTROL, NO NEED FOR I OR D!!
*****/
static void DrivePID(unsigned char num_degrees)
{
    static long ticks_dif;
    //NO derivative control - dGain set to 0
    static long last_dif;
    static long sum_dif;

    //start the left one a bit faster
    signed char r_duty = (DUTY_MAX + DUTY_MIN)/2;
    signed char l_duty = (DUTY_MAX + DUTY_MIN)/2 + L_START_ADJUST;
    //if sent 0 degrees, run forever until next interrupt
    //otherwise signal when done
    //printf("\r\nJust before while loop of drive PID");
    while((num_degrees>0) || !CheckFlag(FALSE))
    {
            if(!left_done)
            {
                //printf("\r\nL Ticks: %lu, R Ticks:
%lu",GetLTicks(),GetRTicks());

            ticks_dif = GetLTicks() - GetRTicks();

            //printf("\r\nticks difference = %ld", ticks_dif);

            sum_dif += ticks_dif;

            //negative ticks dif means R is faster, so L must speed up
            l_duty += ((-ticks_dif)*pGain)/100 + ((-sum_dif)*iGain)/100
+ ((-last_dif)*dGain)/100;
            //negative ticks dif means R should slow down
            r_duty += (ticks_dif*pGain)/100 + (sum_dif*iGain)/100 +
(last_dif*dGain)/100;

            last_dif = ticks_dif;

            //NO anti-windup FOR NOW
            if(l_duty > DUTY_MAX)
            {
                l_duty = DUTY_MAX;
            }else if (l_duty < DUTY_MIN)
            {
                l_duty = DUTY_MIN;
            }
            if(r_duty > DUTY_MAX)
            {
                r_duty = DUTY_MAX;
            }else if (r_duty < DUTY_MIN)
            {
                r_duty = DUTY_MIN;
            }

            //printf("\r\nLeft Duty = %u, Right Duty = %u", l_duty,
r_duty);

            SetRDuty((unsigned char)r_duty);
            SetLDuty((unsigned char)l_duty);
            }

        //test if encoders are done
        if(LEncoderDone())
        {
            //printf("\r\nLeft encoder done");
            StopLMotor();
            left_done = TRUE;
            if(REncoderDone())
            {
                //printf("\r\nRight encoder done");
                StopRMotor();
                left_done = FALSE;
                SendDone();
                return;
            }
        //NOTE I foresee a problem here with stopping one motor before
another...
        } else if(REncoderDone())
        {
            StopRMotor();
        }
    }

    //handle case where you're supposed to run forever
    StopRMotor();
    StopLMotor();
}

/*****
** Function: static void SpinPID(unsigned char num_degrees)
** PID control based on encoder ticks for turning
** of the motors.  NOTE WE ENDED UP DOING JUST
** PROPORTIONAL CONTROL, NO NEED FOR I OR D!!
*****/
static void SpinPID(unsigned char num_degrees)
{
    static long ticks_dif;
    //NO derivative control - dGain set to 0
    static long last_dif;
    static long sum_dif;

    //start the left one a bit faster
    signed char r_duty = (SPIN_DUTY_MAX + SPIN_DUTY_MIN)/2;
    signed char l_duty = (SPIN_DUTY_MAX + SPIN_DUTY_MIN)/2 +
L_START_ADJUST;

    //if sent 0 degrees, run forever until next interrupt
    //otherwise signal when done
    //printf("\r\nJust before while loop of drive PID");
    while((num_degrees>0) || !CheckFlag(FALSE))
    {
            if(!left_done)
            {
                //printf("\r\nL Ticks: %lu, R Ticks:
%lu",GetLTicks(),GetRTicks());

            ticks_dif = GetLTicks() - GetRTicks();

            printf("\r\nticks difference = %ld", ticks_dif);

            sum_dif += ticks_dif;

            //negative ticks dif means R is faster, so L must speed up
            l_duty += ((-ticks_dif)*pGain)/100 + ((-sum_dif)*iGain)/100
+ ((-last_dif)*dGain)/100;
            //negative ticks dif means R should slow down
            r_duty += (ticks_dif*pGain)/100 + (sum_dif*iGain)/100 +
(last_dif*dGain)/100;

            last_dif = ticks_dif;

            //NO anti-windup FOR NOW
            if(l_duty > SPIN_DUTY_MAX)
            {
                l_duty = SPIN_DUTY_MAX;
            }else if (l_duty < DUTY_MIN)
            {
                l_duty = SPIN_DUTY_MIN;
            }
            if(r_duty > SPIN_DUTY_MAX)
            {
                r_duty = SPIN_DUTY_MAX;
            }else if (r_duty < SPIN_DUTY_MIN)
            {
                r_duty = SPIN_DUTY_MIN;
            }

            printf("\r\nLeft Duty = %u, Right Duty = %u", l_duty,
r_duty);
            SetRDuty((unsigned char)r_duty);
            SetLDuty((unsigned char)l_duty);
            }

        //test if encoders are done
        if(LEncoderDone())
        {
            printf("\r\nLeft encoder done");
            StopLMotor();
            left_done = TRUE;
            if(REncoderDone())
            {
                printf("\r\nRight encoder done");
                StopRMotor();
                left_done = FALSE;
                SendDone();
                return;
            }
        //NOTE I foresee a problem here with stopping one motor before
another...
        } else if(REncoderDone())
        {
            StopRMotor();
        }
    }

    //handle case where you're supposed to run forever
    StopRMotor();
    StopLMotor();
}

/*****
** Function: static void Init_C32(void)
** Duh
```

```
*****/
static void Init_C32(void)
{
    Init_Encoders();
    //Init_LED();
    Init_Motors();
    Init_Slave();

    /*TODO own init code*/
    //Interrupts enabled in slave_com.c
}
```

motors.h

```
#ifndef _motors_h
#define _motors_h

/** C32 HARDWARE MAPPINGS **/
/*
T4 - Motor Driver L293A, Direction Left
T3 - Motor Driver L293A, PWM Left
T1 - Motor Driver L293A, Direction Right
T0 - Motor Driver L293A, PWM Right
*/


/** CONSTANTS **/
#define R_MOTOR_DIR_BIT BIT1HI
#define L_MOTOR_DIR_BIT BIT4HI
//#define R_MOTOR_PWM_BIT BIT0HI
//#define L_MOTOR_PWM_BIT BIT3HI
#define DIVIDE_BY_150 0x4B; //75 = 0100 1011
#define NUM_TICKS_IN_PERIOD 0x64; //100

/** PUBLIC PROTOTYPES **/
void Init_Motors(void);
void SetRDuty(unsigned char duty);
void SetLDuty(unsigned char duty);
void ForwardR(void);
void ReverseR(void);
void ForwardL(void);
void ReverseL(void);
void StopMotors(void);
void StopRMotor(void);
void StopLMotor(void);

#endif
```

motors.c

```
#include <ME218_C32.h>
//#include <S12Vec.h>
//#include "ADS12.H"
#include "motors.h"

/** DATA STRUCTURES **/
/** MODULE PRIVATE PROTOTYPES **/
static void EnablePWM(void);


/** PUBLIC FUNCTIONS **/

/*****
** Function: void StopMotors(void)
** Duh
*****/
void StopMotors(void)
{
    SetRDuty(0);
    SetLDuty(0);
}

/*****
** Function: void StopLMotor(void)
** Duh
*****/
void StopLMotor(void)
{
    SetLDuty(0);
}

/*****
** Function: void StopRMotor(void)
** Duh
*****/
void StopRMotor(void)
{
    SetRDuty(0);
}

/*****
** Function: void ReverseR(void)
** Duh
*****/
void ReverseR(void)
{
    PWMPOL |= _S12_PPOL0;
    PTT &= ~R_MOTOR_DIR_BIT;
}

/*****
** Function: void ForwardR(void)
** Duh
*****/
void ForwardR(void)
{
    PWMPOL &= ~_S12_PPOL0;
    PTT |= R_MOTOR_DIR_BIT;
}

/*****
** Function: void ReverseR(void)
** Duh
```

```
*****/
void ReverseL(void)
{
    PWMPOL |= _S12_PPOL3;
    PTT &= ~L_MOTOR_DIR_BIT;
}

/*****
** Function: void ForwardL(void)
** Duh
*****/
void ForwardL(void)
{
    PWMPOL &= ~_S12_PPOL3;
    PTT |= L_MOTOR_DIR_BIT;
}

/*****
** Function: void SetRDuty(unsigned char duty)
** Must send it a char between 0 and 100
*****/
void SetRDuty(unsigned char duty)
{
    PWMDTY0 = duty;
}

/*****
** Function: void SetRDuty(unsigned char duty)
** Must send it a char between 0 and 100
*****/
void SetLDuty(unsigned char duty)
{
    PWMDTY3 = duty;
}

/*****
** Function: void Init_Motors(void)
** Duh
*****/
void Init_Motors(void)
{

    //Direction ports as output
    DDRT |= (R_MOTOR_DIR_BIT | L_MOTOR_DIR_BIT);

    //PWM ports initialized
    EnablePWM();
}


/** PRIVATE FUNCTIONS **/


/*****
** Function: void EnablePWM(void)
** Runs PWM at a pre-scale of 750kHz
** Runs scale clock A and B at 5kHz and
** 100 ticks per period for 50Hz cycling
** and 1:1 duty cycle control!  Cool!
*****/
static void EnablePWM(void)
{
    //Enable PWM for T0,T3
    PWME     |= (_S12_PWME0 | _S12_PWME3);

    //Run pre-scale at 24MHz/32 = 750kHz
    PWMPRCLK |= (_S12_PCKA0 | _S12_PCKA2);
    PWMPRCLK &= ~(_S12_PCKA1);
    PWMPRCLK |= (_S12_PCKB0 | _S12_PCKB2);
    PWMPRCLK &= ~(_S12_PCKB1);

    //Set polarity to rising edge on start
    PWMPOL   |= (_S12_PPOL0 | _S12_PPOL3);

    //Use clock SA & SB
    PWMCLK   |= (_S12_PCLK0 | _S12_PCLK3);

    //Clock speed is 750kHz/(75*2) = 5kHz
    /*Note* Recall extra factor of two*/
    PWMSCLA   = DIVIDE_BY_150;
    PWMSCLB   = DIVIDE_BY_150;

    //DO NOT CENTER ALIGN
    PWMCAE   &= ~(_S12_CAE0 | _S12_CAE3);

    //Set period to be 100 ticks/period = 50Hz
    PWMPER0   = NUM_TICKS_IN_PERIOD;
    PWMPER3   = NUM_TICKS_IN_PERIOD;

    //Start at 0 duty cycle
    SetRDuty(0);
    SetLDuty(0);

    //Port map T0&T
    MODRR    |= (_S12_MODRR0 | _S12_MODRR3);
}
```

slave_com.h

```
#ifndef _slave_com_h
#define _slave_com_h

/** C32 HARDWARE MAPPINGS **/
/*
M5 - SCK on E128
M4 - MOSI on E128
M3 - Tied LO
M2 - MISO on E128
*/


/** CONSTANTS **/
#define SPTEF_BIT BIT5HI
#define DONE_RESPONSE 2
```

```
#define MASTER_QUERY_COMMAND 0xFF

/** PUBLIC PROTOTYPES **/
void Init_Slave(void);
unsigned char GetData(void);
unsigned char CheckFlag(unsigned char reset_flag);
void interrupt _Vec_SPI SPI_interrupt(void);
void SendDone(void);
unsigned char Is_Done(void);
//unsigned char CheckDummy(void);

#endif
```

slave_com.c

```
#include <stdio.h>
#include <ME218_C32.h>
#include <S12Vec.h>
//#include "ADS12.H"
#include "slave_com.h"

/** DATA STRUCTURES **/
static unsigned char control = 0;
static unsigned char data1 = MASTER_QUERY_COMMAND;
static unsigned char com_flag = FALSE;
static unsigned char done_flag = FALSE;
static unsigned char dummy_flag = FALSE;

/** MODULE PRIVATE PROTOTYPES **/


/** PUBLIC FUNCTIONS **/

/*****
** Function: void SendDone(void)
** In the end we just set the done flag to be true.
** Don't want an interrupt to happen during this
*****/
void SendDone(void)
{
    DisableInterrupts;
    printf("\r\nActivated to send done to the E128");
    //the one time we set the SPIDR!
    //com_flag = FALSE;
    done_flag = TRUE;
    /*if(SPISR & BIT5HI)
    {
        SPIDR = TRUE;
    }*/
    EnableInterrupts;
}

/*****
** Function: unsigned char CheckFlag(unsigned char reset_flag)
** Checks flag with two conditions in mind- reset the flag after you
** return what it was or leave it as is.  reset_flag as TRUE
** resets the flag after checking.
```

```
*****/
unsigned char CheckFlag(unsigned char reset_flag)
{
    if(com_flag == TRUE)
    {
        if(reset_flag)
        {
            com_flag = FALSE;
        }
        return TRUE;
    } else return FALSE;

}

/*unsigned char CheckDummy(void)
{
    if(dummy_flag == TRUE)
    {
        dummy_flag = FALSE;
        return TRUE;
    } else return FALSE;
}*/

/*****
** Function: unsigned char GetData(void)
** Returns the data byte last received by the Massuh.
** Then sets the data byte to a dummy value that
** main() will ignore if another request arrives
** before a new communication.
*****/
unsigned char GetData(void)
{
    unsigned char data = data1;
    data1 = MASTER_QUERY_COMMAND;
    return data;
}

/*****
** Function: void Init_Slave(void)
** Duh
*****/
void Init_Slave(void)
{
    //set the baud rate (SPPR = 2, SPR = 5, Baud Rate 125kHz)
    SPIBR = SPIBR | (_S12_SPPR1 | _S12_SPR2 | _S12_SPR1);

    //SPI Clock control
    SPICR1 = SPICR1 | (_S12_CPOL | _S12_CPHA);

    //enable the master
    SPICR1 = SPICR1; //c32 is not the master

    //set interupts
    SPICR1 |= _S12_SPIE;

     //enable communication
    SPICR1 = SPICR1 | _S12_SPE;
    done_flag = FALSE;
}

/*****
** Function: unsigned char Is_Done(void)
** Returns if the encoders are done with their command.
** Also resets the done_flag to FALSE after checking.
*****/
unsigned char Is_Done(void)
{
    if(done_flag)
    {
        done_flag = FALSE;
        return TRUE;
    }else
    {
        return FALSE;
    }
}


/*****
** Function: void interrupt _Vec_SPI SPI_interrupt(void)
** Complicated- see notes below
*****/
void interrupt _Vec_SPI SPI_interrupt(void)
{
    /*
    the slave sends data in one case:
    when the encoders completed a specific
    number of degrees of rotation.  SPIDR
    is set to true above, and in this case the
    com_flag must be erased- since otherwise
    the junk data the E128 sends will get read by
    the C32 on the next wait for instructions.
    No need to worry about the interim checks
    by the E128 since the C32 is in a loop
    and is not looking for E128 calls
    */
    //printf("E128 wants to talk to us!  Yay! or boooo");
    DisableInterrupts;

    com_flag = TRUE;

    //run SPI routine
    control = SPISR;
    data1 = SPIDR;

    if(SPISR & SPTEF_BIT)
    {
        if(done_flag)
        {
            done_flag = FALSE;
            //printf("\r\nSet the SPIDR bit to 2.");
            SPIDR = DONE_RESPONSE;
        } else
        {
            //printf("\r\nSet the SPIDR bit to FALSE.");
            SPIDR = FALSE;
        }
    }


    //BELOW IS OBSELETE IF E128 SENDS 0xFF ON QUERY
    /*if(done_flag)
    {
        //printf("\r\nAcknowledged done set, should tell E128 so");
        com_flag = FALSE;
    }*/

    //dummy_flag = TRUE;
    //this delay seems to be necessary
    //printf("data1 = %x\r\n",data1);
    //printf("Here's what we got %x", data1);
    //set done_flag to false- only true if set by encoders done

   EnableInterrupts;
}

/** PRIVATE FUNCTIONS **/
```

E128 – 8 modules

ball.h/.c
beacon.h/.c
collision.h/.c
flash.h/.c
main.h/.c
master_com.h/.c
tape.h/.c
timer.h/.c

ball.h

```
#ifndef _ball_h
#define _ball_h

/** E128 HARDWARE MAPPINGS **/
/*
T0 - Ball Delivery L293A, Enable 1 (Pin 1)
T1 - Ball Delivery L293A, Enable 2 (Pin 9)
T2 - Ball Delivery L293A, Direction 1 (Pin 2)
T3 - Ball Delivery L293A, PWM 1 (Pin 7)
T4 - Ball Delivery L293A, PWM 2 (Pin 10)
T5 - Ball Delivery L293A, Direction 2 (Pin 15)

//CURRENTLY NOT IMPLEMENTED
P3 - Ball Received Detection
P4 - Ball Delivered Detection
AD4 - Ball Color Detection
*/


/** CONSTANTS **/
#define GATE_NUM_OVERFLOWS_1 45
#define GATE_NUM_OVERFLOWS_2 25

/** PUBLIC PROTOTYPES **/
void Init_Ball(void);
void CloseGate(void);
void OpenGate(void);
void StartOpen(void);
void StopOpen(void);


#endif
```

ball.c

```
#include <hidef.h>      /* common defines and macros */
#include <mc9s12e128.h>     /* derivative information */
#include "timer.h"
#include "bitdef.h"
#include "ball.h"


/** DATA STRUCTURES **/
/** MODULE PRIVATE PROTOTYPES **/


/** PUBLIC FUNCTIONS **/

/*****
** Function: void Init_Ball(void)
**
*****/
void Init_Ball(void)
{
    //Port AD is initialized once, in main

    //Ball received and delivered as inputs
    DDRP &= ~(BIT3HI | BIT4HI);

    //Motor pins as outputs
    DDRT |= (BIT0HI | BIT1HI | BIT2HI | BIT3HI | BIT4HI | BIT5HI);

    //Disable motors at startup
    PTT &= ~(BIT0HI | BIT1HI);
}

void CloseGate(void)
{
    //disable agitator
    PTT &= ~BIT1HI;

    //set motor direction
    PTT &= ~BIT2HI;
    //turn motor on and enable
    PTT |= (BIT3HI | BIT0HI);

    //wait
    Delay_Overflows(GATE_NUM_OVERFLOWS_1+GATE_NUM_OVERFLOWS_2);

    //disable motor
    PTT &= ~BIT0HI;

}


void OpenGate(void)
{
    //set direction
    PTT &= ~BIT3HI;
    //turn on and enable motor
    PTT |= (BIT2HI | BIT0HI);

    //wait
    Delay_Overflows(GATE_NUM_OVERFLOWS_1);

    //set agitator direction
    PTT &= ~BIT5HI;
    //turn on and enable agitator
    PTT |= (BIT4HI | BIT1HI);
    Delay_Overflows(GATE_NUM_OVERFLOWS_2);

    //disable motor
    PTT &= ~BIT0HI;
}

void StartOpen(void)
{
    //set direction
    PTT &= ~BIT3HI;
    //turn on and enable motor
    PTT |= (BIT2HI | BIT0HI);
}

void StopOpen(void)
{
    //disable motor
    PTT &= ~BIT0HI;
}




/** PRIVATE FUNCTIONS **/
```

beacon.h

```
#ifndef _beacon_h
#define _beacon_h

/** E128 HARDWARE MAPPINGS **/
/*
U0 - Duty Cycle Detection
AD5 - Left Tracking
AD6 - Right Tracking
*/

/** CONSTANTS **/
#define L_BEACON 5
#define R_BEACON 6
#define RISING_EDGE_U0 (PTU&BIT0HI)

#define MIN_BALL_FEED_THRESH 43
#define MAX_BALL_FEED_THRESH 53
#define MIN_GOAL_THRESH 80
#define MAX_GOAL_THRESH 100
#define INITIAL_POS_THRESH 65
#define MAX_EASY_GOAL_THRESH 40
#define MIN_EASY_GOAL_THRESH 20
#define MIN_FINAL_GOAL_THRESH 65
#define MAX_FINAL_GOAL_THRESH 75

#define LEFT_BEACON_AD 5
#define RIGHT_BEACON_AD 6

#define NUM_READINGS 10
```

```
/** PUBLIC PROTOTYPES **/
void Init_Beacon(void);
void interrupt _Vec_tim2ch4  MiddleDutyCycle(void);
void SmartDetect(unsigned char low_thresh, unsigned char hi_thresh);
unsigned int Get_M_Duty(void);
unsigned int Get_R_Beacon (void);
unsigned int Get_L_Beacon (void);
unsigned int Average_L_AD_Beacon(void);
unsigned int Average_R_AD_Beacon(void);
unsigned int* Average_AD_Beacon(void);
void Clear_M_Duty(void);

#endif
```

beacon.c

```
#include <hidef.h>      /* common defines and macros */
#include <mc9s12e128.h>     /* derivative information */
#include <stdio.h>
#include "ME218_E128.h"
#include <S12eVec.h>
#include "ADS12.h"
//#include "bitdef.h"
#include "beacon.h"

/** DATA STRUCTURES **/
static unsigned int M_Duty = 0;
static unsigned char M_Duty_Flag = 0;


/** MODULE PRIVATE PROTOTYPES **/
static void InitTimer(void);

/** PUBLIC FUNCTIONS **/

/*****
** Function: void Init_Beacon(void)
**
*****/
void Init_Beacon(void)
{
    //Port AD is initialized once, in main

    //U0 as input for duty cycle detection
    DDRU &= ~BIT0HI;

    InitTimer();
}

unsigned int Get_M_Duty(void)
{
    return(M_Duty);
}

unsigned int Get_L_Beacon(void)
{
   return(ADS12_ReadADPin(LEFT_BEACON_AD));
}

unsigned int Get_R_Beacon (void)
{
   return(ADS12_ReadADPin(RIGHT_BEACON_AD));
}

unsigned int* Average_AD_Beacon(void)
{
    unsigned char i;
    unsigned int sums[2] =
    {
        0,0
    };
    i=0;
    while(i<NUM_READINGS)
    {
       sums[0] += ADS12_ReadADPin(LEFT_BEACON_AD);
       sums[1] += ADS12_ReadADPin(RIGHT_BEACON_AD);
       i++;
    }

    sums[0] = sums[0]/NUM_READINGS;
    sums[1] = sums[1]/NUM_READINGS;
    return sums;
}

unsigned int Average_L_AD_Beacon(void)
{
    unsigned char i;
    unsigned int sum = 0;
    i=0;
    while(i<NUM_READINGS)
    {
       sum += ADS12_ReadADPin(LEFT_BEACON_AD);
       i++;
    }

    return sum/NUM_READINGS;
}

unsigned int Average_R_AD_Beacon(void)
{
    unsigned char i;
    unsigned int sum = 0;
    i=0;
    while(i<NUM_READINGS)
    {
       sum += ADS12_ReadADPin(RIGHT_BEACON_AD);
       i++;
    }

    return sum/NUM_READINGS;
}

void Clear_M_Duty(void)
```

```
{
    M_Duty = 0;
}

void SmartDetect(unsigned char low_thresh, unsigned char hi_thresh)
{
       unsigned int m_value, average;
       unsigned int count = 0;
       unsigned int last_five_sum = 0;

       while(TRUE)
       {
            m_value = M_Duty;
            //printf("\r\nM_duty is %u", m_value);
            //printf("\r\nLeft AD is %u, Right AD is %u",
Get_L_Beacon(), Get_R_Beacon());

            //sample ten values, check the average, if all ten within
bounds, you've got a hit,
            if(M_Duty_Flag == 1)
            {
                m_value = M_Duty;
                //printf("\r\nM_duty is %u", m_value);
                //printf("\r\nLeft AD is %u, Right AD is %u",
Get_L_Beacon(), Get_R_Beacon());
                if(m_value>hi_thresh || m_value<low_thresh)
                {
                    count = 0;
                    last_five_sum = 0;
                } else
                {
                    last_five_sum += m_value;
                    if (count < 4)
                    {
                        count++;
                    } else //count == 4
                    {
                        printf("\r\nGot 5 values in a row within duty
thresholds of %u and %u", low_thresh, hi_thresh);
                        //we've gotten ten contiguous values, at
count==0 thru count==4
                        average = last_five_sum/5;
                        //printf("\r\nThe average is %u", average);
                        if(average<(hi_thresh-3) &&
average>(low_thresh+2))
                        {
                            return;
                        } else
                        {
                            count = 0;
                            last_five_sum = 0;
                        }
                    }

                }
                M_Duty_Flag = 0;
            }
       }
}


void interrupt _Vec_tim2ch4  MiddleDutyCycle(void)
{
    static unsigned int prevTime = 10, HiTime = 10, LoTime = 10;
    static unsigned int currentTime;


    TIM2_TFLG1 = TIM2_TFLG1_C4F_MASK;      //clear flag

    currentTime = TIM2_TC4;
    //printf("\r\nM interrupt");

      if(RISING_EDGE_U0)
      {
          LoTime = currentTime - prevTime;
          //printf("\r\nLo: %u", LoTime);
          M_Duty = HiTime*100 / (LoTime+HiTime);
          //printf("\r\nM: %u", M_Duty);
          M_Duty_Flag = 1;
      }else {
          HiTime  = currentTime - prevTime;
          //printf("\r\nHi: %u", HiTime);
      }

      prevTime = TIM2_TC4;
}


/** PRIVATE FUNCTIONS **/
static void InitTimer(void)
{
    //Turn the timer system on
      TIM2_TSCR1 |= TIM2_TSCR1_TEN_MASK;

      //Set the pre-scale (TBD with some testing in the lab) for now
128, 187.5 kHz
      TIM2_TSCR2 |= (TIM2_TSCR2_PR1_MASK | TIM2_TSCR2_PR2_MASK |
TIM2_TSCR2_PR0_MASK);

      //Port U0 as input capture
      TIM2_TIOS &= ~TIM2_TIOS_IOS4_MASK;

      //Set Input capture pins to capture both rising and falling edges
      TIM2_TCTL3 |= (TIM2_TCTL3_EDG4A_MASK | TIM2_TCTL3_EDG4B_MASK);

    //Clear the Flags
      TIM2_TFLG1 = TIM2_TFLG1_C4F_MASK;

      //Enable Interrupts
      TIM2_TIE = TIM2_TIE_C4I_MASK;
}
```

bitdef.h

```
#ifndef _bitdef_h
#define _bitdef_h

#define BIT0HI 0x01
#define BIT1HI 0x02
#define BIT2HI 0x04
#define BIT3HI 0x08
#define BIT4HI 0x10
#define BIT5HI 0x20
#define BIT6HI 0x40
#define BIT7HI 0x80

#endif
```

collision.h

```
#ifndef _sense_h
#define _sense_h

/** E128 HARDWARE MAPPINGS **/
/*
U5 - Right Bump Sensor
U4 - Left Bump Sensor
*/

/** CONSTANTS **/


/** PUBLIC PROTOTYPES **/
void Init_Collision(void);
unsigned char Check_R_Collision(void);
unsigned char Check_L_Collision(void);

#endif
```

collision.c

```
#include <hidef.h>      /* common defines and macros */
#include <mc9s12e128.h>     /* derivative information */
#include "bitdef.h"
#include "collision.h"

/** DATA STRUCTURES **/


/** MODULE PRIVATE PROTOTYPES **/


/** PUBLIC FUNCTIONS **/

/*****
** Function: void Init_Collision(void)
**
*****/
void Init_Collision(void)
{
    //Bump sensors as inputs
    DDRU &= ~(BIT4HI | BIT5HI);
}

unsigned char Check_R_Collision(void)
{
    if(PTU & BIT4HI)
    {
        return TRUE;
    }else
    {
        return FALSE;
    }
}

unsigned char Check_L_Collision(void)
{
    if(PTU & BIT5HI)
    {
        return TRUE;
    }else
    {
        return FALSE;
    }
}
/** PRIVATE FUNCTIONS **/
```

flash.h

```
#ifndef _flash_h
#define _flash_h

/** E128 HARDWARE MAPPINGS **/
/*
T7 -  Flash Detection
*/

/** CONSTANTS **/


/** PUBLIC PROTOTYPES **/
void Init_Flash(void);
unsigned char Check_Flash(void);

#endif
```

flash.c

```
#include <hidef.h>      /* common defines and macros */
#include <mc9s12e128.h>     /* derivative information */
#include "bitdef.h"
#include "flash.h"

/** DATA STRUCTURES **/


/** MODULE PRIVATE PROTOTYPES **/


/** PUBLIC FUNCTIONS **/
```

```
/*****
** Function: void Init_Flash(void)
**
*****/
void Init_Flash(void)
{
    //Flash IR detection as input
    DDRT &= ~BIT7HI;
    //Do we need interrupt code for
    //the flash detection?
}

unsigned char Check_Flash(void)
{
    if(PTT & BIT7HI)
    {
        return TRUE;
    }else
    {
        return FALSE;
    }
}



/** PRIVATE FUNCTIONS **/
```

main.h

```
#ifndef _main_h
#define _main_h


/** E128 HARDWARE MAPPINGS **/
/*
See module .h's

Port AD is:
AD0 - Right Tape Sensor
AD1 - Left Tape Sensor
AD2 - End Of Line Tape Sensor
AD4 - Ball Color Detection
AD5 - Left Beacon Tracking
AD6 - Right Beacon Tracking

*/


/** CONSTANTS **/
#define TRUE 1
#define FALSE 0


#define ROTATE_IN_SET_LEFT_BOARD 22
#define FIRST_TURN_CW_DEG 10
#define FIRST_TURN_CCW_DEG 8
```

```
#define DEG_AFTER_LINE 18
#define NINETY_DEGREE_TURN 22
#define RIGHT_BOARD_CORRECTION 2
#define BAD_BEACON_ESCAPE_TICKS 10

#define BALL_REVERSE_DEG 4
#define BALL_FORWARD_DEG 4

#define GOAL_FORWARD_DEG 30
#define GOAL_DROP_OVERFLOWS 10

#define FORWARD_AFTER_FIRST_TAPE 20
#define FORWARD_BEFORE_BALL_DEPRESS_AT_START 6
#define FORWARD_BEFORE_DEPRESS_IN_LOOP 16
//#define DRIVE_PAST_LINE_AMOUNT 2
#define REVERSE_AFTER_BALLS_IN_BASKET 6
#define ROTATE_BEFORE_SENSING_GOAL_BEACON 20
#define ROTATE_AFTER_SENSING_GOAL_BEACON 2
#define ROTATE_BEFORE_SENSE_BALL_BEACON 16
#define DRIVE_FORWARD_TO_GATE_OPEN 4
#define DRIVE_HALFWAY_TO_GATE 56
#define REVERSE_HALFWAY_TO_BALL_DISPENSER 38
#define TIME_BEFORE_GATE_CLOSE 10
#define MINIMUM_DELAY 3
#define ONE_SECOND 27

#define FINAL_REVERSE 8
#define FINAL_ROTATE 20
#define REVERSE_TO_FINAL_GOAL 32
#define FORWARD_TO_FINAL_GOAL 58
#define FOWARD_AGAIN_TO_FINAL_GOAL 20



//#define TEST
/** PUBLIC PROTOTYPES **/

/*****
** Function: void main(void)
** Runs the whole shebang on the E128.
*****/
void main(void);
void Set_Board(unsigned char is_left);
unsigned char Is_Board_Set(void);
#endif
```

main.c

```
#include <hidef.h>      /* common defines and macros */
#include <mc9s12e128.h>     /* derivative information */
#include "ME218_E128.h"
#include <stdio.h>
#include <S12eVec.h>
#include "ADS12.H"
#include "ball.h"
#include "beacon.h"
//#include "bitdef.h"
```

```
#include "collision.h"
#include "timer.h"
#include "flash.h"
#include "led.h"
#include "master_com.h"
#include "tape.h"
#include "main.h"

#pragma LINK_INFO DERIVATIVE "SampleS12"

/** DATA STRUCTURES **/
static unsigned char is_left_board = TRUE;
static unsigned char set_board_yet = FALSE;


/** MODULE PRIVATE PROTOTYPES **/
static void Init_E128(void);
static void WaitForFlash(void);
static void FindBallFeedBeacon(void);
static void DriveToTape(unsigned char tape_color);
static void FirstRotate(void);
//static void FollowTapeL(void); OBSELETE
static void DriveToCollision(void);
static void PickupBalls(void);
static void FindGoalBeacon(void);
static void ZeroInOnGoal(void);
static void Gola(void);
static void Test_E128(void);
static void Test_Set_Left_Board(void);
static void Center_On_Line(void);
static void Check_Left_Board(void);
static void Collect_Balls(void);
static void Drive_To_Goal(void);
static void Get_Back_To_Ball_Feed(void);
static void Stage_1(void);
static void Stage_2(void);
static void Stage_3(void);

/** PUBLIC FUNCTIONS **/

void main(void)
{
    unsigned char count;

    printf("\r\n Code begins");
    Init_E128();
    //printf("\r\n Init Done");

    /*
    #ifdef TEST

    printf("\r\n Test begins");
    Test_E128();

    #endif
    */
    Stage_1();
    for(count=0; count<3; count++)
    {
        Stage_2();
    }


    printf("\r\n Stage 3 begins");
    Stage_3();
}

void Set_Board(unsigned char is_left)
{
    set_board_yet = TRUE;
    is_left_board = is_left;
}

unsigned char Is_Board_Set(void)
{
    return set_board_yet;

}

/** PRIVATE FUNCTIONS **/

static void Stage_1(void)
{
    WaitForFlash();
    //printf("Flash found");

    FindBallFeedBeacon();
    //printf("\r\nFound beacon");

    Delay_Overflows(MINIMUM_DELAY);

    DriveToTape(BLACK);
    //printf("\r\nFound tape");

    Delay_Overflows(MINIMUM_DELAY);

    Drive_Forward(FORWARD_AFTER_FIRST_TAPE);
    Wait_For_Encoders();
    //printf("\r\nDrove forward a bit");

    Delay_Overflows(MINIMUM_DELAY);

    FirstRotate();
    //printf("\r\nFirst rotate done");

    Delay_Overflows(MINIMUM_DELAY);

    DriveToTape(BLACK);
    //printf("\r\nDrove to tape again");

    Delay_Overflows(MINIMUM_DELAY);

    Center_On_Line();
    //printf("\r\nCentered on line");

    Delay_Overflows(MINIMUM_DELAY);

    Drive_Forward(FORWARD_BEFORE_BALL_DEPRESS_AT_START);
    Wait_For_Encoders();
    //printf("\r\nDrove forward a bit");

    Delay_Overflows(MINIMUM_DELAY);

    DriveToCollision();
    //printf("\r\nCollision!");

    Delay_Overflows(ONE_SECOND);
}

static void Stage_2(void)
{
    unsigned char i;

    for(i=0; i<4; i++)
    {
        Collect_Balls();
    }

    Delay_Overflows(MINIMUM_DELAY);

    Drive_To_Goal();

    OpenGate();
    Delay_Overflows(TIME_BEFORE_GATE_CLOSE);
    CloseGate();

    Get_Back_To_Ball_Feed();
}

static void Stage_3(void)
{
    unsigned char i;

    for(i=0; i<4; i++)
    {
        Collect_Balls();
    }

    Delay_Overflows(MINIMUM_DELAY);

    Drive_Reverse(FINAL_REVERSE);
    Wait_For_Encoders();

    Delay_Overflows(MINIMUM_DELAY);

    if(is_left_board)
    {
        Turn_CCW(FINAL_ROTATE);
        Wait_For_Encoders();
    } else
    {
        Turn_CW(FINAL_ROTATE);
        Wait_For_Encoders();

    }
    Delay_Overflows(MINIMUM_DELAY);

    if(is_left_board)
    {
        Turn_CCW1();
    } else
    {
        Turn_CW1();
    }
    SmartDetect(MIN_EASY_GOAL_THRESH, MAX_EASY_GOAL_THRESH);
    StopMotors();

    Delay_Overflows(MINIMUM_DELAY);

    //printf("\r\nFound final beacon!!");

    Drive_Forward(FORWARD_TO_FINAL_GOAL);
    Wait_For_Encoders();

    Delay_Overflows(MINIMUM_DELAY);

    Drive_Forward(FOWARD_AGAIN_TO_FINAL_GOAL);
    Wait_For_Encoders();
    Delay_Overflows(MINIMUM_DELAY);

    if(is_left_board)
    {
        Turn_CW(4);
        Wait_For_Encoders();

    } else
    {
        Turn_CCW(4);
        Wait_For_Encoders();

    }

    OpenGate();
    Delay_Overflows(TIME_BEFORE_GATE_CLOSE);
    //CloseGate();
}

static void Get_Back_To_Ball_Feed(void)
{
    Drive_Reverse(REVERSE_HALFWAY_TO_BALL_DISPENSER);
    Wait_For_Encoders();

    Delay_Overflows(MINIMUM_DELAY);

    Drive_Reverse(REVERSE_HALFWAY_TO_BALL_DISPENSER-2);
    Wait_For_Encoders();
    Delay_Overflows(MINIMUM_DELAY);

    //rotate back to beacon
    if(is_left_board)
    {
        Turn_CCW(ROTATE_BEFORE_SENSE_BALL_BEACON);
        Wait_For_Encoders();
        Delay_Overflows(MINIMUM_DELAY);
        Clear_M_Duty();
        Turn_CCW1();
    } else
    {
        Turn_CW(ROTATE_BEFORE_SENSE_BALL_BEACON);
        Wait_For_Encoders();
        Delay_Overflows(MINIMUM_DELAY);
        Clear_M_Duty();
        Turn_CW1();

    }
    SmartDetect(MIN_BALL_FEED_THRESH, MAX_BALL_FEED_THRESH);
    StopMotors();

    Delay_Overflows(MINIMUM_DELAY);

    Drive_Forward(FORWARD_BEFORE_DEPRESS_IN_LOOP);
    Wait_For_Encoders();
    printf("\r\nDrove forward a bit");

    Delay_Overflows(MINIMUM_DELAY);

    DriveToCollision();
    printf("\r\nCollision!");

    Delay_Overflows(ONE_SECOND);
}

static void Drive_To_Goal(void)
{
    Drive_Reverse(REVERSE_AFTER_BALLS_IN_BASKET);
    Wait_For_Encoders();

    Delay_Overflows(MINIMUM_DELAY);

    if(is_left_board)
    {
        Turn_CW(ROTATE_BEFORE_SENSING_GOAL_BEACON);
        Wait_For_Encoders();
        Delay_Overflows(MINIMUM_DELAY);
        Clear_M_Duty();
        Turn_CW1();
    } else
    {
        Turn_CCW(ROTATE_BEFORE_SENSING_GOAL_BEACON);
        Wait_For_Encoders();
        Delay_Overflows(MINIMUM_DELAY);
        Clear_M_Duty();
        Turn_CCW1();
    }
    SmartDetect(MIN_GOAL_THRESH, MAX_GOAL_THRESH);
    StopMotors();

    Delay_Overflows(MINIMUM_DELAY);

    //correct for over-turning
    /*if(is_left_board)
    {
        Turn_CCW(ROTATE_AFTER_SENSING_GOAL_BEACON);
        Wait_For_Encoders();
    } else
    {
        Turn_CW(ROTATE_AFTER_SENSING_GOAL_BEACON);
        Wait_For_Encoders();

    }
    Delay_Overflows(MINIMUM_DELAY);
    */
    Drive_Forward(DRIVE_HALFWAY_TO_GATE);
    Wait_For_Encoders();

    Delay_Overflows(MINIMUM_DELAY);

    //correct for over-turning
    if(is_left_board)
    {
        Turn_CCW1();
    } else
    {
        Turn_CW1();
    }
    SmartDetect(MIN_GOAL_THRESH, MAX_GOAL_THRESH);
    StopMotors();
    Delay_Overflows(MINIMUM_DELAY);

    //more correct for over-turning
    if(is_left_board)
    {
        Turn_CCW(ROTATE_AFTER_SENSING_GOAL_BEACON);
        Wait_For_Encoders();
    } else
    {
        Turn_CW(ROTATE_AFTER_SENSING_GOAL_BEACON);
        Wait_For_Encoders();

    }
    Delay_Overflows(MINIMUM_DELAY);

    DriveToTape(GREEN);
    printf("\r\nDrove to green tape");

    Delay_Overflows(MINIMUM_DELAY);

    Drive_Forward(DRIVE_FORWARD_TO_GATE_OPEN);
    Wait_For_Encoders();
```

```
}

static void Collect_Balls(void)
{
    Drive_Reverse(BALL_REVERSE_DEG);
    Wait_For_Encoders();
    printf("\r\nBack off");

    Delay_Overflows(MINIMUM_DELAY);

    Drive_Forward(BALL_FORWARD_DEG);
    Wait_For_Encoders();
    printf("\r\nBack off");

    Delay_Overflows(MINIMUM_DELAY);

    DriveToCollision();
    printf("\r\nDrove to depress");

    Delay_Overflows(ONE_SECOND);
}

static void Check_Left_Board(void)
{
    unsigned int center, left, right, i;
    for(i = 0; i<5; i++)
    {
        center = Read_Center_Tape();
        left= Read_Left_Tape();
        right = Read_Right_Tape();

        printf("\r\nCenter = %u, Right = %u, Left = %u", center, right,
left);
    }
    getchar();
}

static void Test_Set_Left_Board(void)
{
    while(TRUE)
    {
        if(Check_R_Collision())
        {
            printf("Not left board.");
            is_left_board = FALSE;
            break;
        }else if(Check_L_Collision())
        {
            printf("Is left board.");
            is_left_board = TRUE;
            break;
        }
    }
}

static void FindGoalBeacon(void)
{
    Drive_Reverse(BALL_REVERSE_DEG);
    Wait_For_Encoders();

    if(is_left_board)
    {
        Turn_CW1();
    } else
    {
        Turn_CCW1();
    }

    SmartDetect(MIN_GOAL_THRESH, MAX_GOAL_THRESH);
    StopMotors();
}

static void DriveToCollision(void)
{

    //if our encoders already brought us there...
    if(Check_R_Collision())
    {
        return;
    }

    Drive_Forward1();

    while(TRUE)
    {
        if(Check_R_Collision())
        {
            break;
        }/*else if(Check_L_Collision())
        {
            break;
        }*/
    }
    StopMotors();
}

static void Center_On_Line(void)
{
    //center the robot
    /*printf("\r\nIn center on line driving foward a bitsy");
    Drive_Forward(DRIVE_PAST_LINE_AMOUNT);
    Wait_For_Encoders();

    Delay_Overflows(MINIMUM_DELAY);*/
    //turn CW specified amount
    if(is_left_board)
    {
        printf("Rotating to center");
        //Turn_CCW(ROTATE_TO_CENTER_DEG);
        Turn_CCW1();
        Check_Both_Sensors(BLACK);
        StopMotors();

    } else
    {
        Drive_Forward(RIGHT_BOARD_CORRECTION);
        Wait_For_Encoders();
        Delay_Overflows(MINIMUM_DELAY);

        printf("Rotating to center");
        //Turn_CW(ROTATE_TO_CENTER_DEG);
        Turn_CW1();
        Check_Both_Sensors(BLACK);
        StopMotors();
    }

    //Wait_For_Encoders();

    //NOTE we are doing no checking of the tape sensors...
    //are we cheating?
}

static void FirstRotate(void)
{
    printf("\r\nIn first rotate");
    if(is_left_board)
    {

        Turn_CW(FIRST_TURN_CW_DEG);
    }
    else
    {
        //printf("\r\nSpinning CCW an amount");
        Turn_CCW(FIRST_TURN_CCW_DEG);
    }
    //printf("\r\nWaiting for encoders");
    Wait_For_Encoders();
}

static void DriveToTape(unsigned char tape_color)
{

    Drive_Forward1();
    printf("\r\nSent drive forward in DriveToTape");
    Check_Both_Sensors(tape_color);

        /*if(is_left_board)
        {
            Check_Sensor(RIGHT_TAPE_AD, BLACK_LOWER_LIM,
BLACK_UPPER_LIM);
        }
        else
        {
            Check_Sensor(LEFT_TAPE_AD, BLACK_LOWER_LIM,
BLACK_UPPER_LIM);
        }*/

    StopMotors();
}

static void FindBallFeedBeacon(void)
```

```
{
    unsigned int* sums;
    unsigned int count;
    unsigned char failed;
    while(TRUE)
    {
        //Clear_M_Duty();
        failed = FALSE;
        //this should find out if we're running on left board or not
        if(is_left_board)
        {
            //DO Opposite at start
            Turn_CCW1();
        } else
        {
            Turn_CW1();
        }

        SmartDetect(MIN_BALL_FEED_THRESH, MAX_BALL_FEED_THRESH);
        StopMotors();
        for(count=0; count<10; count++)
        {
            sums = Average_AD_Beacon();
            if(sums[0]>100 || sums[1]>100)
            {
                printf("Bad beacon detected!");
                failed = TRUE;
                break;
            }
        }

        if(failed == FALSE) break;
        else
        {
            if(is_left_board)
            {
                //make it rotate the other way to start!
                Delay_Overflows(MINIMUM_DELAY);
                is_left_board = FALSE;
                Turn_CW(BAD_BEACON_ESCAPE_TICKS);
                Wait_For_Encoders();
                Delay_Overflows(MINIMUM_DELAY);
            } else
            {
                Delay_Overflows(MINIMUM_DELAY);
                is_left_board = TRUE;
                Turn_CCW(BAD_BEACON_ESCAPE_TICKS);
                Wait_For_Encoders();
                Delay_Overflows(MINIMUM_DELAY);
            }
        }
    }
}

static void WaitForFlash(void)
{
    while(TRUE)
    {
        if(Check_Flash())
        {
            return;
        }
    }
}

static void Init_E128(void)
{
    //Initialize Port AD Once:
    if(!(ADS12_Init("OAAAAAAA") == ADS12_OK))
    {
        printf("Error Initializing A/D. \r\n");
    }

    Init_Ball();
    Init_Beacon();
    Init_Collision();
    Init_Flash();
    Init_LED();
    Init_Master();
    Init_Tape();
    Init_Timer();

    EnableInterrupts;
}
```

master_com.h

```
#ifndef _master_com_h
#define _master_com_h

/** E128 HARDWARE MAPPINGS **/
/*
SS - Set HI
SCK - C32 M5
MOSI - C32 M4
MISO C32 M2
*/


/** CONSTANTS **/
#define MOTOR_OFF_BIT BIT0HI
#define MOTOR_STRAIGHT_BIT BIT1HI
#define MOTOR_FWD_BIT BIT2HI
#define MOTOR_CW_BIT BIT2HI //same bit

#define TICKS_DIVISOR 2
#define SLAVE_DONE_COMMAND 2
#define MASTER_QUERY_COMMAND 0xFF

/** PUBLIC PROTOTYPES **/
void Init_Master(void);
void Wait_For_Encoders(void);
unsigned char Encoders_Done(void);
void StopMotors(void);
void Drive_Forward(unsigned char degrees);
void Drive_Reverse(unsigned char degrees);
void Turn_CCW(unsigned char degrees);
void Turn_CW(unsigned char degrees);
//overloaded functions:
//default to 0 degrees = drive forever
void Drive_Forward1(void);
void Drive_Reverse1(void);
void Turn_CCW1(void);
void Turn_CW1(void);

#endif
```

master_com.c

```
#include <stdio.h>
#include <hidef.h>      /* common defines and macros */
#include <mc9s12e128.h>     /* derivative information */
#include "bitdef.h"
#include "master_com.h"

/** DATA STRUCTURES **/


/** MODULE PRIVATE PROTOTYPES **/
static unsigned char Set_Degrees(unsigned char degrees);

/** PUBLIC FUNCTIONS **/

/*****
** Function: void Init_Master(void)
**
*****/
void Init_Master(void)
{
    //set the baud rate (SPPR = 2, SPR = 5, Baud Rate 125kHz)
    SPIBR = SPIBR | (SPIBR_SPPR1_MASK | SPIBR_SPR2_MASK |
SPIBR_SPR1_MASK);

    //SPI Clock control
    SPICR1 = SPICR1 | (SPICR1_CPOL_MASK | SPICR1_CPHA_MASK);

    //enable the master
    SPICR1 = SPICR1 | SPICR1_MSTR_MASK; //E128 is the master

    //enable communication
    SPICR1 = SPICR1 | SPICR1_SPE_MASK;
}

void Wait_For_Encoders(void)
{
    while(TRUE)
    {
        if(Encoders_Done() == SLAVE_DONE_COMMAND)
        {
            return;
        }
    }
}

unsigned char Encoders_Done(void)
{
    unsigned char is_done, query = MASTER_QUERY_COMMAND;

    //printf("\r\nIn Encoders_Done, setting it all up");
    is_done = SPISR; //junk value- is_done will be re-set
    SPIDR = query;

    // While flag not set...
    while(!(SPISR & SPISR_SPTEF_MASK))
    {
        //Wait...
    }

    //printf("\r\nSent your query");
    is_done = SPIDR;
    //printf("\r\nIs done?... %u", is_done);
    return is_done;
}

void StopMotors(void)
{
    unsigned char junk, com_bit = 0;
    com_bit |= MOTOR_OFF_BIT;

    junk = SPISR;
    SPIDR = com_bit;

    // While flag not set...
    while(!(SPISR & SPISR_SPTEF_MASK))
    {
        //Wait...
    }

    junk = SPIDR;
}

void Drive_Forward(unsigned char degrees)
{
    unsigned char junk, com_bit = Set_Degrees(degrees);
    com_bit &= ~MOTOR_OFF_BIT;
    com_bit |= MOTOR_STRAIGHT_BIT;
    com_bit |= MOTOR_FWD_BIT;

    junk = SPISR;
    SPIDR = com_bit;

    // While flag not set...
    while(!(SPISR & SPISR_SPTEF_MASK))
    {
        //Wait...
    }

    junk = SPIDR;
```

```
}

void Drive_Forward1(void)
{
    unsigned char junk, com_bit = 0;
    com_bit &= ~MOTOR_OFF_BIT;
    com_bit |= MOTOR_STRAIGHT_BIT;
    com_bit |= MOTOR_FWD_BIT;

    junk = SPISR;
    SPIDR = com_bit;

    // While flag not set...
    while(!(SPISR & SPISR_SPTEF_MASK))
    {
        //Wait...
    }

    junk = SPIDR;
}

void Drive_Reverse(unsigned char degrees)
{
    unsigned char junk, com_bit = Set_Degrees(degrees);
    com_bit &= ~MOTOR_OFF_BIT;
    com_bit |= MOTOR_STRAIGHT_BIT;
    com_bit &= ~MOTOR_FWD_BIT;

    junk = SPISR;
    SPIDR = com_bit;

    // While flag not set...
    while(!(SPISR & SPISR_SPTEF_MASK))
    {
        //Wait...
    }

    junk = SPIDR;
}

void Drive_Reverse1(void)
{
    unsigned char junk, com_bit = 0;
    com_bit &= ~MOTOR_OFF_BIT;
    com_bit |= MOTOR_STRAIGHT_BIT;
    com_bit &= ~MOTOR_FWD_BIT;

    junk = SPISR;
    SPIDR = com_bit;

    // While flag not set...
    while(!(SPISR & SPISR_SPTEF_MASK))
    {
        //Wait...
    }

    junk = SPIDR;
```

```
}

void Turn_CCW(unsigned char degrees)
{
    unsigned char junk, com_bit = Set_Degrees(degrees);
    com_bit &= ~MOTOR_OFF_BIT;
    com_bit &= ~MOTOR_STRAIGHT_BIT;
    com_bit &= ~MOTOR_CW_BIT;

    junk = SPISR;
    SPIDR = com_bit;

    // While flag not set...
    while(!(SPISR & SPISR_SPTEF_MASK))
    {
        //Wait...
    }

    junk = SPIDR;
}

void Turn_CCW1(void)
{
    unsigned char junk, com_bit = 0;
    com_bit &= ~MOTOR_OFF_BIT;
    com_bit &= ~MOTOR_STRAIGHT_BIT;
    com_bit &= ~MOTOR_CW_BIT;

    junk = SPISR;
    SPIDR = com_bit;

    // While flag not set...
    while(!(SPISR & SPISR_SPTEF_MASK))
    {
        //Wait...
    }

    junk = SPIDR;
}

void Turn_CW(unsigned char degrees)
{
    unsigned char junk, com_bit = Set_Degrees(degrees);
    com_bit &= ~MOTOR_OFF_BIT;
    com_bit &= ~MOTOR_STRAIGHT_BIT;
    com_bit |= MOTOR_CW_BIT;

    junk = SPISR;
    SPIDR = com_bit;

    // While flag not set...
    while(!(SPISR & SPISR_SPTEF_MASK))
    {
        //Wait...
    }

    junk = SPIDR;
```

```
}

void Turn_CW1(void)
{
    unsigned char junk, com_bit = 0;
    com_bit &= ~MOTOR_OFF_BIT;
    com_bit &= ~MOTOR_STRAIGHT_BIT;
    com_bit |= MOTOR_CW_BIT;

    junk = SPISR;
    SPIDR = com_bit;

    // While flag not set...
    while(!(SPISR & SPISR_SPTEF_MASK))
    {
        //Wait...
    }

    junk = SPIDR;
}

/** PRIVATE FUNCTIONS **/

/**
**NOTE* only accepts up to 31*5=155 ticks
*/
static unsigned char Set_Degrees(unsigned char degrees)
{
    unsigned char bit = 0;

    degrees = degrees/TICKS_DIVISOR;

    if(degrees & BIT0HI)
    {
        bit |= BIT3HI;
    }
    if(degrees & BIT1HI)
    {
        bit |= BIT4HI;
    }
    if(degrees & BIT2HI)
    {
        bit |= BIT5HI;
    }
    if(degrees & BIT3HI)
    {
        bit |= BIT6HI;
    }
    if(degrees & BIT4HI)
    {
        bit |= BIT7HI;
    }
    return bit;
}
```

tape.h

```
#ifndef _tape_h
#define _tape_h

/** E128 HARDWARE MAPPINGS **/
/*
AD0 - Right Tape Sensor
AD1 - Left Tape Sensor
AD2 - End Of Line Tape Sensor
*/


/** CONSTANTS **/
#define RED_LOWER_LIM_L 511
#define RED_UPPER_LIM_L 515
#define GREEN_LOWER_LIM_L 399
#define GREEN_UPPER_LIM_L 404
#define BLACK_LOWER_LIM_L 578
#define BLACK_UPPER_LIM_L 710
#define WHITE_LOWER_LIM_L 138
#define WHITE_UPPER_LIM_L 143

#define RED_LOWER_LIM_R 549
#define RED_UPPER_LIM_R 553
#define GREEN_LOWER_LIM_R 419
#define GREEN_UPPER_LIM_R 423
#define BLACK_LOWER_LIM_R 608
#define BLACK_UPPER_LIM_R 710
#define WHITE_LOWER_LIM_R 136
#define WHITE_UPPER_LIM_R 140

#define RED_LOWER_LIM_C 0
#define RED_UPPER_LIM_C 0
#define GREEN_LOWER_LIM_C 0
#define GREEN_UPPER_LIM_C 0
#define BLACK_LOWER_LIM_C 0
#define BLACK_UPPER_LIM_C 0
#define WHITE_LOWER_LIM_C 0
#define WHITE_UPPER_LIM_C 0

#define WHITE 0
#define GREEN 1
#define RED 2
#define BLACK 3

#define DRIVE_STRAIGHT_DEG     45
#define LOST_LINE_ROTATE_DEG 6

#define SET_LEFT_BOARD_LOWER_LIM 400


//#define ROTATE_TO_CENTER_DEG 50

#define RIGHT_TAPE_AD  1
#define LEFT_TAPE_AD   0
#define EOL_TAPE_AD 2

/** PUBLIC PROTOTYPES **/
```

```
void Init_Tape(void);
void Follow_Tape(unsigned short lower_lim);
//void Center_On_Line(unsigned char is_clockwise);
void Check_Sensor(unsigned char ad_pin, unsigned short lower_lim,
unsigned short upper_lim);
void Find_Line_End(unsigned short lower_lim_r, unsigned short
lower_lim_l);
void Check_Both_Sensors(unsigned char color);
//test function
unsigned int Read_Center_Tape(void);
unsigned int Read_Right_Tape(void);
unsigned int Read_Left_Tape(void);

#endif
```

tape.c

```
#include <stdio.h>
#include <hidef.h>      /* common defines and macros */
#include <mc9s12e128.h>     /* derivative information */
#include "ADS12.H"
#include "bitdef.h"
#include "master_com.h"
#include "tape.h"

/** DATA STRUCTURES **/


/** MODULE PRIVATE PROTOTYPES **/


/** PUBLIC FUNCTIONS **/

/*****
** Function: void Init_Tape(void)
**
*****/
void Init_Tape(void)
{
    //tape AD pins initiated in main
    //TODO we need a way to deal with the symmetricity
    //of the playing fields
}

void Check_Sensor(unsigned char ad_pin, unsigned short lower_lim,
unsigned short upper_lim)
{
    unsigned short numTriggers = 0;
    unsigned short ad_val;


    while(numTriggers < 1000)
    {
        ad_val = ADS12_ReadADPin(ad_pin);
        //printf("%u Tape Value\r\n", ADS12_ReadADPin(L_TAPE_SENSOR));
        if( (ad_val > lower_lim) && (ad_val < upper_lim) )
        {
            numTriggers+=10;
            //foundTape = TRUE;
            //printf("\r\nTape Triggered %u times\r\n", numTriggers);
        } else
        {
            if(numTriggers > 0) numTriggers--;
            //printf("\r\nTape Not Triggered.  numtrig at %u
times\r\n", numTriggers);
        }
    }
    //printf("\r\nTape Found.  End of Tracking Function\r\n");

}

void Check_Both_Sensors(unsigned char color)
{
    unsigned short ad_val_l, ad_val_r;
    unsigned short lower_l, lower_r, upper_l, upper_r;
    unsigned short numTriggers = 0;
    unsigned char should_set_board = FALSE;

    if(!Is_Board_Set())
    {
        should_set_board = TRUE;
    }

    switch(color)
    {
        case GREEN:
            printf("\r\nGreen is the color of money");
            lower_l = GREEN_LOWER_LIM_L;
            lower_r = GREEN_LOWER_LIM_R;
            upper_l = GREEN_UPPER_LIM_L;
            upper_r = GREEN_UPPER_LIM_R;
            break;
        case RED:
            lower_l = RED_LOWER_LIM_L;
            lower_r = RED_LOWER_LIM_R;
            upper_l = RED_UPPER_LIM_L;
            upper_r = RED_UPPER_LIM_R;
            break;
        case BLACK:
            printf("\r\nBlack is the color of the month");
            lower_l = BLACK_LOWER_LIM_L;
            lower_r = BLACK_LOWER_LIM_R;
            upper_l = BLACK_UPPER_LIM_L;
            upper_r = BLACK_UPPER_LIM_R;
            break;
        default:
            //white
            lower_l = WHITE_LOWER_LIM_L;
            lower_r = WHITE_LOWER_LIM_R;
            upper_l = WHITE_UPPER_LIM_L;
            upper_r = WHITE_UPPER_LIM_R;
            break;
    }
    while(numTriggers < 2)
    {
        //printf("\r\nIn while loop of check both sensors");
        ad_val_l = ADS12_ReadADPin(LEFT_TAPE_AD);
        ad_val_r = ADS12_ReadADPin(RIGHT_TAPE_AD);
        //printf("\r\nLeft: %u Right: %u", ad_val_l, ad_val_r);

        if(should_set_board)
        {
            if(ad_val_l > SET_LEFT_BOARD_LOWER_LIM)
            {
                Set_Board(TRUE);
                should_set_board = FALSE;
            } else if(ad_val_r > SET_LEFT_BOARD_LOWER_LIM)
            {
                Set_Board(FALSE);
                should_set_board = FALSE;
            }
        }


        if( (ad_val_l > lower_l) && (ad_val_l < upper_l) )
        {
            printf("\r\nLeft triggered!");
            printf("\r\nNum triggers = %u", numTriggers);
            numTriggers++;
            //foundTape = TRUE;
            //printf("\r\nTape Triggered %u times\r\n", numTriggers);
        } else
        {
            //if(numTriggers > 0) numTriggers--;
            //printf("\r\nTape Not Triggered.  Left AD val %u",
ad_val_l);
        }
        if( (ad_val_r > lower_r) && (ad_val_r < upper_r) )
        {
            printf("\r\nRight triggered!");
            printf("\r\nNum triggers = %u", numTriggers);
            numTriggers++;
            //foundTape = TRUE;
            //printf("\r\nTape Triggered %u times\r\n", numTriggers);
        } else
        {
            //printf("\r\nTape Not Triggered.  Right AD val %u",
ad_val_r);
            //if(numTriggers > 0) numTriggers--;
            //printf("\r\nTape Not Triggered.  numtrig at %u
times\r\n", numTriggers);
        }

    }
    //printf("\r\nTape Found.  End of Tracking Function\r\n");

}
```

```
void Find_Line_End(unsigned short lower_lim_r, unsigned short
lower_lim_l)
{
    unsigned short ad_val_l, ad_val_r;
    unsigned short numTriggers = 0;

    while(numTriggers < 2000)
    {
        ad_val_l = ADS12_ReadADPin(LEFT_TAPE_AD);
        ad_val_r = ADS12_ReadADPin(RIGHT_TAPE_AD);
        //printf("%u Tape Value\r\n", ADS12_ReadADPin(L_TAPE_SENSOR));
        if( (ad_val_l < lower_lim_l) )
        {
            numTriggers+=10;
            //foundTape = TRUE;
            //printf("\r\nTape Triggered %u times\r\n", numTriggers);
        } else
        {
            if(numTriggers > 0) numTriggers--;
            //printf("\r\nTape Not Triggered.  numtrig at %u
times\r\n", numTriggers);
        }
        if( (ad_val_r < lower_lim_r) )
        {
            numTriggers+=10;
            //foundTape = TRUE;
            //printf("\r\nTape Triggered %u times\r\n", numTriggers);
        } else
        {
            if(numTriggers > 0) numTriggers--;
            //printf("\r\nTape Not Triggered.  numtrig at %u
times\r\n", numTriggers);
        }
    }
}

void Follow_Tape(unsigned short lower_lim)
{
    /*
    best practices from lab 8 dictated driving
    straight for a while then checking and perhaps
    correcting error when necessary
    */
    short reading_right = 0;
      short reading_left = 0;
      short reading_eol = ADS12_ReadADPin(EOL_TAPE_AD);;

    //TODO what is the right way to check for readings
    while(reading_eol < lower_lim)
    {
        //optimize this to drive straight right until
        //you'd lose it one way or another
        Drive_Forward(DRIVE_STRAIGHT_DEG);
        reading_right = ADS12_ReadADPin(RIGHT_TAPE_AD);
          reading_left  = ADS12_ReadADPin(LEFT_TAPE_AD);

          //optimize rotation for not overcorrection
          if(reading_right < lower_lim)
          {
              //turn left
              Turn_CCW(LOST_LINE_ROTATE_DEG);
          } else if(reading_left < lower_lim)
          {
              //turn right
              Turn_CW(LOST_LINE_ROTATE_DEG);
          }
    }
}

unsigned int Read_Center_Tape(void)
{
    return(ADS12_ReadADPin(EOL_TAPE_AD));
}

unsigned int Read_Left_Tape(void)
{
    return(ADS12_ReadADPin(LEFT_TAPE_AD));
}

unsigned int Read_Right_Tape(void)
{
    return(ADS12_ReadADPin(RIGHT_TAPE_AD));
}
/** PRIVATE FUNCTIONS **/
```

timer.h

```
#ifndef _timer_h
#define _timer_h

/** E128 HARDWARE MAPPINGS **/


/** CONSTANTS **/


/** PUBLIC PROTOTYPES **/
void Init_Timer(void);
void Delay_Overflows(unsigned char num_overflows);

#endif
```

timer.c

```
#include <hidef.h>      /* common defines and macros */
#include <mc9s12e128.h>     /* derivative information */
#include "bitdef.h"
#include "timer.h"

/** DATA STRUCTURES **/


/** MODULE PRIVATE PROTOTYPES **/
/** PUBLIC FUNCTIONS **/

/*****
** Function: void Init_Ball(void)
**
*****/
void Init_Timer(void)
{
    //Turn the timer system on
      TIM0_TSCR1 |= TIM0_TSCR1_TEN_MASK;

      //Set the pre-scale (TBD with some testing in the lab) for now
16, 187.5 kHz
      TIM0_TSCR2 |= TIM0_TSCR2_PR2_MASK;
      TIM0_TSCR2 &= ~(TIM0_TSCR2_PR1_MASK | TIM0_TSCR2_PR0_MASK);

      //set T3
      TIM0_TIOS |= TIM0_TIOS_IOS7_MASK;

      //Set output compare to pin disconnected
      TIM0_TCTL1 &= ~(TIM0_TCTL3_EDG7A_MASK | TIM0_TCTL3_EDG7B_MASK);

    //Clear the Flags (probably don't need to do this, but good to make
sure)
      TIM0_TFLG1 = TIM0_TFLG1_C7F_MASK;

      //NO Enable Interrupts
}

void Delay_Overflows(unsigned char num_overflows)
{
    int i = 0;
    while(i < num_overflows)
    {
        //check the flag
        if((TIM0_TFLG2 & TIM0_TFLG2_TOF_MASK) == TIM0_TFLG2_TOF_MASK)
        {
            i++;  //increment the overflows count
            TIM0_TFLG2 = TIM0_TFLG2_TOF_MASK; //clear the flag
        }
    }
}

/*void Delay_Milliseconds(unsigned char num_ms)
{



}*/

/** PRIVATE FUNCTIONS **/
```