//#############################################################################

// FILE: LAB6starter\_main.c

//

// TITLE: Lab Starter

//#############################################################################

// Included Files

**#include** <stdio.h>

**#include** <stdlib.h>

**#include** <stdarg.h>

**#include** <string.h>

**#include** <math.h>

**#include** <limits.h>

**#include** "F28x\_Project.h"

**#include** "driverlib.h"

**#include** "device.h"

**#include** "F28379dSerial.h"

**#include** "LEDPatterns.h"

**#include** "song.h"

**#include** "dsp.h"

**#include** "fpu32/fpu\_rfft.h"

**#define** PI 3.1415926535897932384626433832795

**#define** TWOPI 6.283185307179586476925286766559

**#define** HALFPI 1.5707963267948966192313216916398

// The Launchpad's CPU Frequency set to 200 you should not change this value

**#define** LAUNCHPAD\_CPU\_FREQUENCY 200

// ----- code for CAN start here -----

**#include** "F28379dCAN.h"

//#define TX\_MSG\_DATA\_LENGTH 4

//#define TX\_MSG\_OBJ\_ID 0 //transmit

**#define** RX\_MSG\_DATA\_LENGTH 8

**#define** RX\_MSG\_OBJ\_ID\_1 1 //measurement from sensor 1

**#define** RX\_MSG\_OBJ\_ID\_2 2 //measurement from sensor 2

**#define** RX\_MSG\_OBJ\_ID\_3 3 //quality from sensor 1

**#define** RX\_MSG\_OBJ\_ID\_4 4 //quality from sensor 2

**#define** E4NOTE ((uint16\_t)(((50000000/2)/2)/329.63))

// ----- code for CAN end here -----

**void** **init\_eQEPs**(**void**);

**float** **readEncLeft**(**void**);

**float** **readEncRight**(**void**);

**float** LeftWheel\_rad = 0.0;

**float** RightWheel\_rad = 0.0;

**float** LeftWheel\_rad\_1 = 0.0;

**float** RightWheel\_rad\_1 = 0.0;

**float** LeftWheel\_meter = 0.0;

**float** RightWheel\_meter = 0.0;

**void** **setEPWM2A**(**float** controleffort);

**void** **setEPWM2B**(**float** controleffort);

**void** **setEPWM9\_buzzer**();

**void** **stopEPWM9\_buzzer**();

//WL

**float** angle1 = -90;

**float** angle2 = 0.0;

**void** **setEPWM8A**(**float** angle1);

**void** **setEPWM8B**(**float** angle2);

**float** **saturate**(**float** input,**float** saturation\_limit);

//WL

**int** updown1 = 1;

**int** updown2 = 0;

uint16\_t mycount = 0;

**float** mycount2 = 85;

**float** mycount3 = -85;

**int** wait = 0;

//WL - Robot Commands

**int** drop = 0;

**float** uLeft = 5.0;

**float** uRight = 5.0;

//DW – Global variables to store the current and previous positions of the left and right wheels in order to calculate the left and right wheel velocities. The previous positions of the left and right wheels are the positions of the wheels 4 milliseconds ago.

**float** PosLeft\_K = 0.0;

**float** PosLeft\_K\_1 = 0.0;

**float** PosRight\_K = 0.0;

**float** PosRight\_K\_1 = 0.0;

**float** VLeftK = 0.0;

**float** VRightK = 0.0;

**float** e\_K\_Left = 0.0;

**float** I\_K\_Left = 0.0;

**float** I\_K\_Left\_1 = 0.0;

**float** e\_K\_Left\_1 = 0.0;

**float** u\_K\_Left = 0.0;

**float** e\_K\_Right = 0.0;

**float** I\_K\_Right = 0.0;

**float** I\_K\_Right\_1 = 0.0;

**float** e\_K\_Right\_1 = 0.0;

**float** u\_K\_Right = 0.0;

**float** Kp = 3.0;

**float** Ki = 25.0;

**float** Vref = 0.1;

**float** ref\_right = 400;

**float** ref\_front = 1400;

**float** Kp\_right = 0.001;

**float** Kp\_front = 0.0002;

**float** distfront = 0.0;

**float** distright = 0.0;

uint16\_t rightwallfollow = 0;

**float** threshold1 = 400;

**float** threshold2 = 500;

**float** Vfront = 0.1;

**float** Vright = 0.25;

**float** e\_turn = 0.0;

**float** K\_turn = 3.0;

**float** turn = 0.0;

**float** robot\_width = 0.173;

**float** R\_wheel = 0.0593;

**float** theta\_dot\_left = 0.0;

**float** theta\_dot\_right = 0.0;

**float** theta\_dot\_avg = 0.0;

**float** theta\_avg = 0.0;

**float** x\_dot = 0.0;

**float** x\_dot\_1 = 0.0;

**float** y\_dot = 0.0;

**float** y\_dot\_1 = 0.0;

**float** x = 0;

**float** y = 0;

**float** x\_1 = 0.0;

**float** y\_1 = 0.0;

**float** phi = 0.0;

**float** printLV4 = 0;

**float** printLV5 = 0;

**float** printLV6 = 0;

**float** printLV7 = 0;

**float** printLV8 = 0;

**extern** uint16\_t NewLVData;

**extern** **float** fromLVvalues[LVNUM\_TOFROM\_FLOATS];

**extern** LVSendFloats\_t DataToLabView;

**extern** **char** LVsenddata[LVNUM\_TOFROM\_FLOATS\*4+2];

**extern** uint16\_t newLinuxCommands;

**extern** **float** LinuxCommands[CMDNUM\_FROM\_FLOATS];

// Interrupt Service Routines predefinition

\_\_interrupt **void** **cpu\_timer0\_isr**(**void**);

\_\_interrupt **void** **cpu\_timer1\_isr**(**void**);

\_\_interrupt **void** **cpu\_timer2\_isr**(**void**);

\_\_interrupt **void** **SWI\_isr**(**void**);

// ----- code for CAN start here -----

\_\_interrupt **void** **can\_isr**(**void**);

// ----- code for CAN end here -----

\_\_interrupt **void** **SPIB\_isr**(**void**);

**void** **setupSpib**(**void**);

// Count variables

uint32\_t numTimer0calls = 0;

uint32\_t numTimer1calls = 0;

uint32\_t numSWIcalls = 0;

**extern** uint32\_t numRXA;

uint16\_t UARTPrint = 0;

uint16\_t LEDdisplaynum = 0;

int16\_t accelx\_raw = 0;

int16\_t accely\_raw = 0;

int16\_t accelz\_raw = 0;

int16\_t gyrox\_raw = 0;

int16\_t gyroy\_raw = 0;

int16\_t gyroz\_raw = 0;

**float** accelx = 0;

**float** accely = 0;

**float** accelz = 0;

**float** gyrox = 0;

**float** gyroy = 0;

**float** gyroz = 0;

int32\_t SpibNumCalls = 0;

// ----- code for CAN start here -----

// volatile uint32\_t txMsgCount = 0;

// extern uint16\_t txMsgData[4];

**volatile** uint32\_t rxMsgCount\_1 = 0;

**volatile** uint32\_t rxMsgCount\_3 = 0;

**extern** uint16\_t rxMsgData[8];

uint32\_t dis\_raw\_1[2];

uint32\_t dis\_raw\_3[2];

uint32\_t dis\_1 = 0;

uint32\_t dis\_3 = 0;

uint32\_t quality\_raw\_1[4];

uint32\_t quality\_raw\_3[4];

**float** quality\_1 = 0.0;

**float** quality\_3 = 0.0;

uint32\_t lightlevel\_raw\_1[4];

uint32\_t lightlevel\_raw\_3[4];

**float** lightlevel\_1 = 0.0;

**float** lightlevel\_3 = 0.0;

uint32\_t measure\_status\_1 = 0;

uint32\_t measure\_status\_3 = 0;

**volatile** uint32\_t errorFlag = 0;

// ----- code for CAN end here -----

**void** **main**(**void**)

{

// PLL, WatchDog, enable Peripheral Clocks

// This example function is found in the F2837xD\_SysCtrl.c file.

**InitSysCtrl**();

**InitGpio**();

// DW - setup GPIO pin 55 (SIMO Slave in Master Out)

// Blue LED on LaunchPad

**GPIO\_SetupPinMux**(31, GPIO\_MUX\_CPU1, 0);

**GPIO\_SetupPinOptions**(31, GPIO\_OUTPUT, GPIO\_PUSHPULL);

GpioDataRegs.GPASET.bit.GPIO31 = 1;

// Red LED on LaunchPad

**GPIO\_SetupPinMux**(34, GPIO\_MUX\_CPU1, 0);

**GPIO\_SetupPinOptions**(34, GPIO\_OUTPUT, GPIO\_PUSHPULL);

GpioDataRegs.GPBSET.bit.GPIO34 = 1;

// LED1 and PWM Pin

**GPIO\_SetupPinMux**(22, GPIO\_MUX\_CPU1, 0);

**GPIO\_SetupPinOptions**(22, GPIO\_OUTPUT, GPIO\_PUSHPULL);

GpioDataRegs.GPACLEAR.bit.GPIO22 = 1;

// LED2

**GPIO\_SetupPinMux**(94, GPIO\_MUX\_CPU1, 0);

**GPIO\_SetupPinOptions**(94, GPIO\_OUTPUT, GPIO\_PUSHPULL);

GpioDataRegs.GPCCLEAR.bit.GPIO94 = 1;

// LED3

**GPIO\_SetupPinMux**(95, GPIO\_MUX\_CPU1, 0);

**GPIO\_SetupPinOptions**(95, GPIO\_OUTPUT, GPIO\_PUSHPULL);

GpioDataRegs.GPCCLEAR.bit.GPIO95 = 1;

// LED4

**GPIO\_SetupPinMux**(97, GPIO\_MUX\_CPU1, 0);

**GPIO\_SetupPinOptions**(97, GPIO\_OUTPUT, GPIO\_PUSHPULL);

GpioDataRegs.GPDCLEAR.bit.GPIO97 = 1;

// LED5

**GPIO\_SetupPinMux**(111, GPIO\_MUX\_CPU1, 0);

**GPIO\_SetupPinOptions**(111, GPIO\_OUTPUT, GPIO\_PUSHPULL);

GpioDataRegs.GPDCLEAR.bit.GPIO111 = 1;

// LED6

**GPIO\_SetupPinMux**(130, GPIO\_MUX\_CPU1, 0);

**GPIO\_SetupPinOptions**(130, GPIO\_OUTPUT, GPIO\_PUSHPULL);

GpioDataRegs.GPECLEAR.bit.GPIO130 = 1;

// LED7

**GPIO\_SetupPinMux**(131, GPIO\_MUX\_CPU1, 0);

**GPIO\_SetupPinOptions**(131, GPIO\_OUTPUT, GPIO\_PUSHPULL);

GpioDataRegs.GPECLEAR.bit.GPIO131 = 1;

// LED8

**GPIO\_SetupPinMux**(25, GPIO\_MUX\_CPU1, 0);

**GPIO\_SetupPinOptions**(25, GPIO\_OUTPUT, GPIO\_PUSHPULL);

GpioDataRegs.GPACLEAR.bit.GPIO25 = 1;

// LED9

**GPIO\_SetupPinMux**(26, GPIO\_MUX\_CPU1, 0);

**GPIO\_SetupPinOptions**(26, GPIO\_OUTPUT, GPIO\_PUSHPULL);

GpioDataRegs.GPACLEAR.bit.GPIO26 = 1;

// LED10

**GPIO\_SetupPinMux**(27, GPIO\_MUX\_CPU1, 0);

**GPIO\_SetupPinOptions**(27, GPIO\_OUTPUT, GPIO\_PUSHPULL);

GpioDataRegs.GPACLEAR.bit.GPIO27 = 1;

// LED11

**GPIO\_SetupPinMux**(60, GPIO\_MUX\_CPU1, 0);

**GPIO\_SetupPinOptions**(60, GPIO\_OUTPUT, GPIO\_PUSHPULL);

GpioDataRegs.GPBCLEAR.bit.GPIO60 = 1;

// LED12

**GPIO\_SetupPinMux**(61, GPIO\_MUX\_CPU1, 0);

**GPIO\_SetupPinOptions**(61, GPIO\_OUTPUT, GPIO\_PUSHPULL);

GpioDataRegs.GPBCLEAR.bit.GPIO61 = 1;

// LED13

**GPIO\_SetupPinMux**(157, GPIO\_MUX\_CPU1, 0);

**GPIO\_SetupPinOptions**(157, GPIO\_OUTPUT, GPIO\_PUSHPULL);

GpioDataRegs.GPECLEAR.bit.GPIO157 = 1;

// LED14

**GPIO\_SetupPinMux**(158, GPIO\_MUX\_CPU1, 0);

**GPIO\_SetupPinOptions**(158, GPIO\_OUTPUT, GPIO\_PUSHPULL);

GpioDataRegs.GPECLEAR.bit.GPIO158 = 1;

// LED15

**GPIO\_SetupPinMux**(159, GPIO\_MUX\_CPU1, 0);

**GPIO\_SetupPinOptions**(159, GPIO\_OUTPUT, GPIO\_PUSHPULL);

GpioDataRegs.GPECLEAR.bit.GPIO159 = 1;

// LED16

**GPIO\_SetupPinMux**(160, GPIO\_MUX\_CPU1, 0);

**GPIO\_SetupPinOptions**(160, GPIO\_OUTPUT, GPIO\_PUSHPULL);

GpioDataRegs.GPFCLEAR.bit.GPIO160 = 1;

//WIZNET Reset

**GPIO\_SetupPinMux**(0, GPIO\_MUX\_CPU1, 0);

**GPIO\_SetupPinOptions**(0, GPIO\_OUTPUT, GPIO\_PUSHPULL);

GpioDataRegs.GPASET.bit.GPIO0 = 1;

//ESP8266 Reset

**GPIO\_SetupPinMux**(1, GPIO\_MUX\_CPU1, 0);

**GPIO\_SetupPinOptions**(1, GPIO\_OUTPUT, GPIO\_PUSHPULL);

GpioDataRegs.GPASET.bit.GPIO1 = 1;

//SPIRAM CS Chip Select

**GPIO\_SetupPinMux**(19, GPIO\_MUX\_CPU1, 0);

**GPIO\_SetupPinOptions**(19, GPIO\_OUTPUT, GPIO\_PUSHPULL);

GpioDataRegs.GPASET.bit.GPIO19 = 1;

//DRV8874 #1 DIR Direction

**GPIO\_SetupPinMux**(29, GPIO\_MUX\_CPU1, 0);

**GPIO\_SetupPinOptions**(29, GPIO\_OUTPUT, GPIO\_PUSHPULL);

GpioDataRegs.GPASET.bit.GPIO29 = 1;

//DRV8874 #2 DIR Direction

**GPIO\_SetupPinMux**(32, GPIO\_MUX\_CPU1, 0);

**GPIO\_SetupPinOptions**(32, GPIO\_OUTPUT, GPIO\_PUSHPULL);

GpioDataRegs.GPBSET.bit.GPIO32 = 1;

//DAN28027 CS Chip Select

**GPIO\_SetupPinMux**(9, GPIO\_MUX\_CPU1, 0);

**GPIO\_SetupPinOptions**(9, GPIO\_OUTPUT, GPIO\_PUSHPULL);

GpioDataRegs.GPASET.bit.GPIO9 = 1;

//MPU9250 CS Chip Select

**GPIO\_SetupPinMux**(66, GPIO\_MUX\_CPU1, 0);

**GPIO\_SetupPinOptions**(66, GPIO\_OUTPUT, GPIO\_PUSHPULL);

GpioDataRegs.GPCSET.bit.GPIO66 = 1;

//WIZNET CS Chip Select

**GPIO\_SetupPinMux**(125, GPIO\_MUX\_CPU1, 0);

**GPIO\_SetupPinOptions**(125, GPIO\_OUTPUT, GPIO\_PUSHPULL);

GpioDataRegs.GPDSET.bit.GPIO125 = 1;

//PushButton 1

**GPIO\_SetupPinMux**(4, GPIO\_MUX\_CPU1, 0);

**GPIO\_SetupPinOptions**(4, GPIO\_INPUT, GPIO\_PULLUP);

//PushButton 2

**GPIO\_SetupPinMux**(5, GPIO\_MUX\_CPU1, 0);

**GPIO\_SetupPinOptions**(5, GPIO\_INPUT, GPIO\_PULLUP);

//PushButton 3

**GPIO\_SetupPinMux**(6, GPIO\_MUX\_CPU1, 0);

**GPIO\_SetupPinOptions**(6, GPIO\_INPUT, GPIO\_PULLUP);

//PushButton 4

**GPIO\_SetupPinMux**(7, GPIO\_MUX\_CPU1, 0);

**GPIO\_SetupPinOptions**(7, GPIO\_INPUT, GPIO\_PULLUP);

//Joy Stick Pushbutton

**GPIO\_SetupPinMux**(8, GPIO\_MUX\_CPU1, 0);

**GPIO\_SetupPinOptions**(8, GPIO\_INPUT, GPIO\_PULLUP);

//DW - DC Motors EPWM2A and EPWM2B. These motors spin the left and right wheels on the segbot.

**GPIO\_SetupPinMux**(2, GPIO\_MUX\_CPU1, 1);

**GPIO\_SetupPinMux**(3, GPIO\_MUX\_CPU1, 1);

//WL-Servo motors EPWM8A and EPWM8B

**GPIO\_SetupPinMux**(14, GPIO\_MUX\_CPU1, 1);

**GPIO\_SetupPinMux**(15, GPIO\_MUX\_CPU1, 1);

//DW - Buzzer EPWM9A

**GPIO\_SetupPinMux**(16, GPIO\_MUX\_CPU1, 5);

EALLOW; // Below are protected registers

GpioCtrlRegs.GPAPUD.bit.GPIO16 = 5; // For EPWM9A

EDIS;

// ----- code for CAN start here -----

//GPIO17 - CANRXB

**GPIO\_SetupPinMux**(17, GPIO\_MUX\_CPU1, 2);

**GPIO\_SetupPinOptions**(17, GPIO\_INPUT, GPIO\_ASYNC);

//GPIO12 - CANTXB

**GPIO\_SetupPinMux**(12, GPIO\_MUX\_CPU1, 2);

**GPIO\_SetupPinOptions**(12, GPIO\_OUTPUT, GPIO\_PUSHPULL);

// ----- code for CAN end here -----

// ----- code for CAN start here -----

// Initialize the CAN controller

InitCANB();

// Set up the CAN bus bit rate to 1000 kbps

setCANBitRate(200000000, 1000000);

// Enables Interrupt line 0, Error & Status Change interrupts in CAN\_CTL register.

CanbRegs.CAN\_CTL.bit.IE0= 1;

CanbRegs.CAN\_CTL.bit.EIE= 1;

// ----- code for CAN end here -----

// Clear all interrupts and initialize PIE vector table:

// Disable CPU interrupts

DINT;

// Initialize the PIE control registers to their default state.

// The default state is all PIE interrupts disabled and flags

// are cleared.

// This function is found in the F2837xD\_PieCtrl.c file.

**InitPieCtrl**();

// Disable CPU interrupts and clear all CPU interrupt flags:

IER = 0x0000;

IFR = 0x0000;

// Initialize the PIE vector table with pointers to the shell Interrupt

// Service Routines (ISR).

// This will populate the entire table, even if the interrupt

// is not used in this example. This is useful for debug purposes.

// The shell ISR routines are found in F2837xD\_DefaultIsr.c.

// This function is found in F2837xD\_PieVect.c.

**InitPieVectTable**();

// Interrupts that are used in this example are re-mapped to

// ISR functions found within this project

EALLOW; // This is needed to write to EALLOW protected registers

PieVectTable.TIMER0\_INT = &cpu\_timer0\_isr;

PieVectTable.TIMER1\_INT = &cpu\_timer1\_isr;

PieVectTable.TIMER2\_INT = &cpu\_timer2\_isr;

PieVectTable.SCIA\_RX\_INT = &RXAINT\_recv\_ready;

PieVectTable.SCIB\_RX\_INT = &RXBINT\_recv\_ready;

PieVectTable.SCIC\_RX\_INT = &RXCINT\_recv\_ready;

PieVectTable.SCID\_RX\_INT = &RXDINT\_recv\_ready;

PieVectTable.SCIA\_TX\_INT = &TXAINT\_data\_sent;

PieVectTable.SCIB\_TX\_INT = &TXBINT\_data\_sent;

PieVectTable.SCIC\_TX\_INT = &TXCINT\_data\_sent;

PieVectTable.SCID\_TX\_INT = &TXDINT\_data\_sent;

PieVectTable.SPIB\_RX\_INT = &SPIB\_isr;

PieVectTable.EMIF\_ERROR\_INT = &SWI\_isr;

// ----- code for CAN start here -----

PieVectTable.CANB0\_INT = &can\_isr;

// ----- code for CAN end here -----

EDIS; // This is needed to disable write to EALLOW protected registers

// Initialize the CpuTimers Device Peripheral. This function can be

// found in F2837xD\_CpuTimers.c

**InitCpuTimers**();

// Configure CPU-Timer 0, 1, and 2 to interrupt every given period:

// 200MHz CPU Freq, Period (in uSeconds)

**ConfigCpuTimer**(&CpuTimer0, LAUNCHPAD\_CPU\_FREQUENCY, 1000);

**ConfigCpuTimer**(&CpuTimer1, LAUNCHPAD\_CPU\_FREQUENCY, 4000);

**ConfigCpuTimer**(&CpuTimer2, LAUNCHPAD\_CPU\_FREQUENCY, 10000);

// Enable CpuTimer Interrupt bit TIE

CpuTimer0Regs.TCR.all = 0x4000;

CpuTimer1Regs.TCR.all = 0x4000;

CpuTimer2Regs.TCR.all = 0x4000;

init\_serialSCIA(&SerialA,115200);

//DW - EPWM2A settings

EPwm2Regs.TBCTL.bit.CTRMODE = 0;

EPwm2Regs.TBCTL.bit.FREE\_SOFT = 2;

EPwm2Regs.TBCTL.bit.PHSEN = 1;

EPwm2Regs.TBCTL.bit.CLKDIV = 0;

EPwm2Regs.TBCTR = 0;

EPwm2Regs.TBPRD = 2500;

EPwm2Regs.CMPA.bit.CMPA = 0;

EPwm2Regs.CMPB.bit.CMPB = 0;

EPwm2Regs.AQCTLA.bit.CAU = 1;

EPwm2Regs.AQCTLB.bit.CBU = 1;

EPwm2Regs.AQCTLA.bit.ZRO = 2;

EPwm2Regs.AQCTLB.bit.ZRO = 2;

EPwm2Regs.TBPHS.bit.TBPHS = 0;

//WL-EPWM8A

EPwm8Regs.TBCTL.bit.CTRMODE = 0;

EPwm8Regs.TBCTL.bit.FREE\_SOFT = 2;

EPwm8Regs.TBCTL.bit.PHSEN = 1;

EPwm8Regs.TBCTL.bit.CLKDIV = 100;

EPwm8Regs.TBCTR = 0;

EPwm8Regs.TBPRD = 62500;

//WL - set CMPA register to 8% duty cycle

EPwm8Regs.CMPA.bit.CMPA = 5000;

EPwm8Regs.CMPB.bit.CMPB = 5000;

EPwm8Regs.AQCTLA.bit.CAU = 1;

EPwm8Regs.AQCTLB.bit.CBU = 1;

EPwm8Regs.AQCTLA.bit.ZRO = 2;

EPwm8Regs.AQCTLB.bit.ZRO = 2;

EPwm8Regs.TBPHS.bit.TBPHS = 0;

//DW - EPWM9A settings

EPwm9Regs.TBCTL.bit.CTRMODE = 0;

EPwm9Regs.TBCTL.bit.FREE\_SOFT = 2;

EPwm9Regs.TBCTL.bit.PHSEN = 1;

EPwm9Regs.TBCTL.bit.CLKDIV = 0; //sets the clock divider for the EWPM = 1. This was chosen so as to not affect the desired note to sound.

EPwm9Regs.TBCTR = 0;

//DW - We configured the period of EPWM9 (which is the register connected to the buzzer on the segbot)

EPwm9Regs.TBPRD = E4NOTE;

//DW - We set the compare A register = 0 because we wanted to initialize the register with a 0% duty cycle

EPwm9Regs.CMPA.bit.CMPA = 0;

//DW - We set CAU = 2 because this tells the Action Qualifier to clear (set low) the PWM output when the

//counter reaches the compare A value. Because CMPA is set to 0, this clear action will keep the PWM

//output low until otherwise changed. This is a simple trick to keep the buzzer from sounding unless

//otherwise programmed. Later on, in the cputimer 1 interrupt function, we use functions setEPWM9\_buzzer()

//and stopEPWM9\_buzzer(), which simply turns on and off the buzzer by setting the CMPA register to a 50%

//duty cycle and then back to a 0% duty cycle.

EPwm9Regs.AQCTLA.bit.CAU = 2;

setupSpib();

init\_eQEPs();

// Enable CPU int1 which is connected to CPU-Timer 0, CPU int13

// which is connected to CPU-Timer 1, and CPU int 14, which is connected

// to CPU-Timer 2: int 12 is for the SWI.

IER |= M\_INT1;

IER |= M\_INT6;

IER |= M\_INT8; // SCIC SCID

IER |= M\_INT9; // SCIA CANB

IER |= M\_INT12;

IER |= M\_INT13;

IER |= M\_INT14;

// Enable TINT0 in the PIE: Group 1 interrupt 7

PieCtrlRegs.PIEIER1.bit.INTx7 = 1;

// Enable SWI in the PIE: Group 12 interrupt 9

PieCtrlRegs.PIEIER12.bit.INTx9 = 1;

PieCtrlRegs.PIEIER6.bit.INTx3 = 1; //SPiB

// ----- code for CAN start here -----

// Enable CANB in the PIE: Group 9 interrupt 7

PieCtrlRegs.PIEIER9.bit.INTx7 = 1;

// ----- code for CAN end here -----

// ----- code for CAN start here -----

// Enable the CAN interrupt signal

CanbRegs.CAN\_GLB\_INT\_EN.bit.GLBINT0\_EN = 1;

// ----- code for CAN end here -----

init\_serialSCIC(&SerialC,115200);

init\_serialSCID(&SerialD,115200);

// Enable global Interrupts and higher priority real-time debug events

EINT; // Enable Global interrupt INTM

ERTM; // Enable Global realtime interrupt DBGM

// ----- code for CAN start here -----

// // Transmit Message

// // Initialize the transmit message object used for sending CAN messages.

// // Message Object Parameters:

// // Message Object ID Number: 0

// // Message Identifier: 0x1

// // Message Frame: Standard

// // Message Type: Transmit

// // Message ID Mask: 0x0

// // Message Object Flags: Transmit Interrupt

// // Message Data Length: 4 Bytes

// //

// CANsetupMessageObject(CANB\_BASE, TX\_MSG\_OBJ\_ID, 0x1, CAN\_MSG\_FRAME\_STD,

// CAN\_MSG\_OBJ\_TYPE\_TX, 0, CAN\_MSG\_OBJ\_TX\_INT\_ENABLE,

// TX\_MSG\_DATA\_LENGTH);

// Measured Distance from 1

// Initialize the receive message object 1 used for receiving CAN messages.

// Message Object Parameters:

// Message Object ID Number: 1

// Message Identifier: 0x060b0101

// Message Frame: Standard

// Message Type: Receive

// Message ID Mask: 0x0

// Message Object Flags: Receive Interrupt

// Message Data Length: 8 Bytes (Note that DLC field is a "don't care"

// for a Receive mailbox)

//

CANsetupMessageObject(CANB\_BASE, RX\_MSG\_OBJ\_ID\_1, 0x060b0101, *CAN\_MSG\_FRAME\_EXT*,

*CAN\_MSG\_OBJ\_TYPE\_RX*, 0, CAN\_MSG\_OBJ\_RX\_INT\_ENABLE,

RX\_MSG\_DATA\_LENGTH);

// Measured Distance from 2

// Initialize the receive message object 2 used for receiving CAN messages.

// Message Object Parameters:

// Message Object ID Number: 2

// Message Identifier: 0x060b0102

// Message Frame: Standard

// Message Type: Receive

// Message ID Mask: 0x0

// Message Object Flags: Receive Interrupt

// Message Data Length: 8 Bytes (Note that DLC field is a "don't care"

// for a Receive mailbox)

//

CANsetupMessageObject(CANB\_BASE, RX\_MSG\_OBJ\_ID\_2, 0x060b0103, *CAN\_MSG\_FRAME\_EXT*,

*CAN\_MSG\_OBJ\_TYPE\_RX*, 0, CAN\_MSG\_OBJ\_RX\_INT\_ENABLE,

RX\_MSG\_DATA\_LENGTH);

// Measurement Quality from 1

// Initialize the receive message object 2 used for receiving CAN messages.

// Message Object Parameters:

// Message Object ID Number: 3

// Message Identifier: 0x060b0201

// Message Frame: Standard

// Message Type: Receive

// Message ID Mask: 0x0

// Message Object Flags: Receive Interrupt

// Message Data Length: 8 Bytes (Note that DLC field is a "don't care"

// for a Receive mailbox)

//

CANsetupMessageObject(CANB\_BASE, RX\_MSG\_OBJ\_ID\_3, 0x060b0201, *CAN\_MSG\_FRAME\_EXT*,

*CAN\_MSG\_OBJ\_TYPE\_RX*, 0, CAN\_MSG\_OBJ\_RX\_INT\_ENABLE,

RX\_MSG\_DATA\_LENGTH);

// Measurement Quality from 2

// Initialize the receive message object 2 used for receiving CAN messages.

// Message Object Parameters:

// Message Object ID Number: 4

// Message Identifier: 0x060b0202

// Message Frame: Standard

// Message Type: Receive

// Message ID Mask: 0x0

// Message Object Flags: Receive Interrupt

// Message Data Length: 8 Bytes (Note that DLC field is a "don't care"

// for a Receive mailbox)

//

CANsetupMessageObject(CANB\_BASE, RX\_MSG\_OBJ\_ID\_4, 0x060b0203, *CAN\_MSG\_FRAME\_EXT*,

*CAN\_MSG\_OBJ\_TYPE\_RX*, 0, CAN\_MSG\_OBJ\_RX\_INT\_ENABLE,

RX\_MSG\_DATA\_LENGTH);

//

// Start CAN module operations

//

CanbRegs.CAN\_CTL.bit.Init = 0;

CanbRegs.CAN\_CTL.bit.CCE = 0;

// // Initialize the transmit message object data buffer to be sent

// txMsgData[0] = 0x12;

// txMsgData[1] = 0x34;

// txMsgData[2] = 0x56;

// txMsgData[3] = 0x78;

// // Loop Forever - A message will be sent once per second.

// for(;;)

// {

//

// CANsendMessage(CANB\_BASE, TX\_MSG\_OBJ\_ID, TX\_MSG\_DATA\_LENGTH, txMsgData);

// txMsgCount++;

// DEVICE\_DELAY\_US(1000000);

// }

// ----- code for CAN end here -----

//Servo position initialization

setEPWM8A(angle1);

// IDLE loop. Just sit and loop forever (optional):

// while(1)

// {

// if (UARTPrint == 1 ) {

// serial\_printf(&&SerialA, "Left Wheel [rad]: %.3f Right Wheel [rad]: %.3f Left Wheel [m]: %.3f Right WheeSerialA,"Left velocity: %.3f Right velocity: %.3f\r\n",VLeftK,VRightK);

// UARTPrint = 0;

// }

// }

}

// SWI\_isr, Using this interrupt as a Software started interrupt

\_\_interrupt **void** **SWI\_isr**(**void**) {

// These three lines of code allow SWI\_isr, to be interrupted by other interrupt functions

// making it lower priority than all other Hardware interrupts.

PieCtrlRegs.PIEACK.all = PIEACK\_GROUP12;

**asm**(" NOP"); // Wait one cycle

EINT; // Clear INTM to enable interrupts

// Insert SWI ISR Code here.......

numSWIcalls++;

DINT;

}

// cpu\_timer0\_isr - CPU Timer0 ISR

\_\_interrupt **void** **cpu\_timer0\_isr**(**void**)

{

CpuTimer0.InterruptCount++;

numTimer0calls++;

**if** ((numTimer0calls%250) == 0) {

displayLEDletter(LEDdisplaynum);

LEDdisplaynum++;

**if** (LEDdisplaynum == 0xFFFF) { // prevent roll over exception

LEDdisplaynum = 0;

}

}

GpioDataRegs.GPCCLEAR.bit.GPIO66 = 1;

SpibRegs.SPIFFRX.bit.RXFFIL = 8;

SpibRegs.SPITXBUF = 0xBA00;

SpibRegs.SPITXBUF = 0x0000;

SpibRegs.SPITXBUF = 0x0000;

SpibRegs.SPITXBUF = 0x0000;

SpibRegs.SPITXBUF = 0x0000;

SpibRegs.SPITXBUF = 0x0000;

SpibRegs.SPITXBUF = 0x0000;

SpibRegs.SPITXBUF = 0x0000;

**if** ((numTimer0calls%50) == 0) {

// Blink LaunchPad Red LED

GpioDataRegs.GPBTOGGLE.bit.GPIO34 = 1;

}

// Acknowledge this interrupt to receive more interrupts from group 1

PieCtrlRegs.PIEACK.all = PIEACK\_GROUP1;

}

// cpu\_timer1\_isr - CPU Timer1 ISR

\_\_interrupt **void** **cpu\_timer1\_isr**(**void**)

{

numTimer1calls++;

//DW – Previous values of the left and right wheel rotation angles are stored as LeftWheel\_rad\_1 and RightWheel\_rad\_1. The average of the left and right wheel rotation angles are calculated and named as theta\_avg.

LeftWheel\_rad\_1 = LeftWheel\_rad;

LeftWheel\_rad = readEncLeft();

RightWheel\_rad\_1 = RightWheel\_rad;

RightWheel\_rad = readEncRight();

theta\_avg = 0.5 \* (RightWheel\_rad + LeftWheel\_rad);

//DW – Angular speed is calculated by dividing the dividing the difference between the current rotation angle and the previous rotation angle (4 milliseconds ago) by 4 milliseconds.

theta\_dot\_left = (LeftWheel\_rad - LeftWheel\_rad\_1)/0.004;

theta\_dot\_right = (RightWheel\_rad - RightWheel\_rad\_1)/0.004;

theta\_dot\_avg = 0.5 \* (theta\_dot\_left + theta\_dot\_right);

//DW – Converted the radians reading to meters based on table calibration. By moving the robot forward by 0.5 m (using the floor tiles), we got a radian value of approximately 8.3. Therefore, multiply the radians by 0.06 to convert the radian values to meters.

LeftWheel\_meter = LeftWheel\_rad \* 0.06;

RightWheel\_meter = RightWheel\_rad \* 0.06;

PosLeft\_K\_1 = PosLeft\_K;

PosLeft\_K = LeftWheel\_meter;

VLeftK = (PosLeft\_K - PosLeft\_K\_1)/0.004;

PosRight\_K\_1 = PosRight\_K;

PosRight\_K = RightWheel\_meter;

VRightK = (PosRight\_K - PosRight\_K\_1)/0.004;

phi = (R\_wheel/robot\_width)\*(RightWheel\_rad - LeftWheel\_rad);

x\_dot = R\_wheel\*theta\_dot\_avg\***cos**(phi);

x\_dot\_1 = x\_dot;

y\_dot = R\_wheel\*theta\_dot\_avg\***sin**(phi);

y\_dot\_1 = y\_dot;

//DW - integrate x\_dot and y\_dot using trapezoidal rule to find x and y robot position

x\_1 = x;

x = x\_1 + 0.004\*(x\_dot + x\_dot\_1)/2;

y\_1 = y;

y = y\_1 + 0.004\*(y\_dot + y\_dot\_1)/2;

//DW - Before entering the wall following algo we want to check if the two distance sensors are giving

//us good readings. Otherwise we will set them to the max.

// if (measure\_status\_1 == 0) {

// distright = dis\_1;

// }

// else {

// distright = 1400; // set to max reading if error

// }

// if (measure\_status\_3 == 0) {

// distfront = dis\_3;

// }

// else {

// distfront = 1400; // set to max reading if error

// }

//DW - When

// if (rightwallfollow == 1){

// turn = Kp\_right \* (ref\_right - distright);

// Vref = Vright;

//

// //DW - this conditional statement tells the segbot when to stop right wall following.

// //Essentially is signals the segbot to turn left because the front distance sensor has detected

// //a wall is too close ahead. This "safety distance" is threshold 1 and is configured by us.

// //For exercise 8, we decided to sound the buzzer whenever the segbot turns left, so we also call

// //the function setEPWM9\_buzzer().

// if (distfront < threshold1) {

// rightwallfollow = 0;

// setEPWM9\_buzzer();

// }

// }

// //DW - This are the parameter equations when it is time for the segbot to turn left. Vref is changed to Vfront.

// //We have set Vfront = 0.1, relative to Vref = 0.25. Vfront is relatively lower than Vref, which makes sense

// //because the segbot should slow down when a wall is detected in front of it.

// else {

// turn = Kp\_front \* (ref\_front - distfront);

// Vref = Vfront;

//

//

// if (distfront > threshold2) {

// rightwallfollow = 1;

// stopEPWM9\_buzzer();

// }

// }

//DW – The functions setEPWM2A() and setEPWM2B() are called in order to determine which one controls either the left or right motor. The uLeft and uRight are control effort variables that controls the speed and direction of the motors when it is passed through setEPWM2A() and setEPWM2B(). By first setting uLeft to 0, we were able to figure out that EPWM2A drives the right motor and EPWM2B drives the left motor. In addition, we negate uLeft since the wheel rotates in the negative direction when it is passed a positive number while the robot is moving in its forward direction.

// setEPWM2A(uRight);

// setEPWM2B(-uLeft);

//DW - Setting up the decoupled PI controller and the steering controller by coupling the motor control loops. “turn” controls the difference between the two motor speeds. Also, within the steering controller, VLeftK and VRightK tracks the reference.

e\_turn = VLeftK - VRightK + turn;

e\_K\_Left\_1 = e\_K\_Left;

e\_K\_Left = Vref - VLeftK - K\_turn\*e\_turn;

I\_K\_Left\_1 = I\_K\_Left;

I\_K\_Left = I\_K\_Left\_1 + 0.004\*( e\_K\_Left + e\_K\_Left\_1)/2;

**if** ( u\_K\_Left > 10 || u\_K\_Left<-10) {

I\_K\_Left = I\_K\_Left\_1\*0.95;

}

u\_K\_Left = Kp\*e\_K\_Left + Ki\*I\_K\_Left;

e\_K\_Right\_1 = e\_K\_Right;

e\_K\_Right = Vref - VRightK + K\_turn\*e\_turn;

I\_K\_Right\_1 = I\_K\_Right;

I\_K\_Right = I\_K\_Right\_1 + 0.004\*( e\_K\_Right + e\_K\_Right\_1)/2;

**if** ( u\_K\_Right > 10 || u\_K\_Right<-10) {

I\_K\_Right = I\_K\_Right\_1\*0.95;

}

u\_K\_Right = Kp\*e\_K\_Right + Ki\*I\_K\_Right;

setEPWM2A(u\_K\_Right);

setEPWM2B(-u\_K\_Left);

//DW - This is the wireless receiving and sending, respectively, of segbot data using the ESP32 and LabVIEW

//When new data has been received from labVIEW (NEWLVData == 1) we want to update our variables.

//This is super useful for real time adjustment of the robots functionality because we can update

//certain values in LabVIEW such as threshold1 and threshold2 which are configurable distances that

//tell the segbot when to go straight (right wall follow) or turn left (distfront < threshold1).

//We were able to change these values while the robot was running and see its motion improve

//almost immediately. LabVIEW is also useful because it simulated the robots position on a drawing board.

**if** (NewLVData == 1) {

NewLVData = 0;

Vref = fromLVvalues[0];

turn = fromLVvalues[1];

drop = (**int**)fromLVvalues[2];

printLV4 = fromLVvalues[3];

printLV5 = fromLVvalues[4];

printLV6 = fromLVvalues[5];

printLV7 = fromLVvalues[6];

printLV8 = fromLVvalues[7];

// threshold1 = printLV3;

threshold2 = printLV4;

ref\_right = printLV5;

ref\_front = printLV6;

}

//DW - every 0.2 seconds we are sending data on the position (x,y) and pose (phi) of the segbot.

**if**((numTimer1calls%50) == 0) {

DataToLabView.floatData[0] = x;

DataToLabView.floatData[1] = y;

DataToLabView.floatData[2] = phi;

DataToLabView.floatData[3] = 2.0\*((**float**)numTimer1calls)\*.001;

DataToLabView.floatData[4] = 3.0\*((**float**)numTimer1calls)\*.001;

DataToLabView.floatData[5] = (**float**)numTimer1calls;

DataToLabView.floatData[6] = (**float**)numTimer1calls\*4.0;

DataToLabView.floatData[7] = (**float**)numTimer1calls\*5.0;

LVsenddata[0] = '\*'; // header for LVdata

LVsenddata[1] = '$';

**for** (**int** i=0;i<LVNUM\_TOFROM\_FLOATS\*4;i++) {

**if** (i%2==0) {

LVsenddata[i+2] = DataToLabView.rawData[i/2] & 0xFF;

} **else** {

LVsenddata[i+2] = (DataToLabView.rawData[i/2]>>8) & 0xFF;

}

}

serial\_sendSCID(&SerialD, LVsenddata, 4\*LVNUM\_TOFROM\_FLOATS + 2);

}

CpuTimer1.InterruptCount++;

}

// cpu\_timer2\_isr CPU Timer2 ISR

//WL - Using one servo motor

\_\_interrupt **void** **cpu\_timer2\_isr**(**void**)

{

// Blink LaunchPad Blue LED

GpioDataRegs.GPATOGGLE.bit.GPIO31 = 1;

CpuTimer2.InterruptCount++;

**if** ((CpuTimer2.InterruptCount % 50) == 0) {

// UARTPrint = 1;

}

**if** (drop == 1){

// if (front == 1){

//

// }

**if** (updown1 == 1){

angle1 += 1;

setEPWM8A(angle1);

**if** (angle1 >= 0.0) {

updown1 = 0;

}

}

**if** (updown1 == 0) {

wait ++;

**if** (wait > 200) {

angle1 -= 1;

setEPWM8A(angle1);

**if** (angle1 <= -90) {

updown1 = 1;

drop = 0;

wait = 0;

}

}

}

}

}

**void** **setupSpib**(**void**) //Call this function in main() somewhere after the DINT; line of code.

{

int16\_t temp = 0;

//Step 1.

// cut and paste here all the SpibRegs initializations you found for part 3. Make sure the TXdelay in between each transfer to 0. Also donÃ¢â‚¬â„¢t forget to cut and paste the GPIO settings for GPIO9, 63, 64, 65, 66 which are also a part of the SPIB setup.

**GPIO\_SetupPinMux**(9, GPIO\_MUX\_CPU1, 0); // Set as GPIO9 and used as DAN28027 SS

**GPIO\_SetupPinOptions**(9, GPIO\_OUTPUT, GPIO\_PUSHPULL); // Make GPIO9 an Output Pin

GpioDataRegs.GPASET.bit.GPIO9 = 1; //Initially Set GPIO9/SS High so DAN28027 is not selected

**GPIO\_SetupPinMux**(66, GPIO\_MUX\_CPU1, 0); // Set as GPIO66 and used as MPU-9250 SS

**GPIO\_SetupPinOptions**(66, GPIO\_OUTPUT, GPIO\_PUSHPULL); // Make GPIO66 an Output Pin

GpioDataRegs.GPCSET.bit.GPIO66 = 1; //Initially Set GPIO66/SS High so MPU-9250 is not selected

**GPIO\_SetupPinMux**(63, GPIO\_MUX\_CPU1, 15); //Set GPIO63 pin to SPISIMOB

**GPIO\_SetupPinMux**(64, GPIO\_MUX\_CPU1, 15); //Set GPIO64 pin to SPISOMIB

**GPIO\_SetupPinMux**(65, GPIO\_MUX\_CPU1, 15); //Set GPIO65 pin to SPICLKB

EALLOW;

GpioCtrlRegs.GPBPUD.bit.GPIO63 = 0; // Enable Pull-ups on SPI PINs Recommended by TI for SPI Pins

GpioCtrlRegs.GPCPUD.bit.GPIO64 = 0;

GpioCtrlRegs.GPCPUD.bit.GPIO65 = 0;

GpioCtrlRegs.GPBQSEL2.bit.GPIO63 = 3; // Set I/O pin to asynchronous mode recommended for SPI

GpioCtrlRegs.GPCQSEL1.bit.GPIO64 = 3; // Set I/O pin to asynchronous mode recommended for SPI

GpioCtrlRegs.GPCQSEL1.bit.GPIO65 = 3; // Set I/O pin to asynchronous mode recommended for SPI

EDIS;

// ---------------------------------------------------------------------------

SpibRegs.SPICCR.bit.SPISWRESET = 0; // Put SPI in Reset

SpibRegs.SPICTL.bit.CLK\_PHASE = 1; //This happens to be the mode for both the DAN28027 and

SpibRegs.SPICCR.bit.CLKPOLARITY = 0; //The MPU-9250, Mode 01.

SpibRegs.SPICTL.bit.MASTER\_SLAVE = 1; // Set to SPI Master

SpibRegs.SPICCR.bit.SPICHAR = 0xF; // Set to transmit and receive 16-bits each write to SPITXBUF

SpibRegs.SPICTL.bit.TALK = 1; // Enable transmission

SpibRegs.SPIPRI.bit.FREE = 1; // Free run, continue SPI operation

SpibRegs.SPICTL.bit.SPIINTENA = 0; // Disables the SPI interrupt

SpibRegs.SPIBRR.bit.SPI\_BIT\_RATE = 49; // Set SCLK bit rate to 1 MHz so 1us period. SPI base clock is

// 50MHZ. And this setting divides that base clock to create SCLKÃ¢â‚¬â„¢s period

SpibRegs.SPISTS.all = 0x0000; // Clear status flags just in case they are set for some reason

SpibRegs.SPIFFTX.bit.SPIRST = 1;// Pull SPI FIFO out of reset, SPI FIFO can resume transmit or receive.

SpibRegs.SPIFFTX.bit.SPIFFENA = 1; // Enable SPI FIFO enhancements

SpibRegs.SPIFFTX.bit.TXFIFO = 0; // Write 0 to reset the FIFO pointer to zero, and hold in reset

SpibRegs.SPIFFTX.bit.TXFFINTCLR = 1; // Write 1 to clear SPIFFTX[TXFFINT] flag just in case it is set

SpibRegs.SPIFFRX.bit.RXFIFORESET = 0; // Write 0 to reset the FIFO pointer to zero, and hold in reset

SpibRegs.SPIFFRX.bit.RXFFOVFCLR = 1; // Write 1 to clear SPIFFRX[RXFFOVF] just in case it is set

SpibRegs.SPIFFRX.bit.RXFFINTCLR = 1; // Write 1 to clear SPIFFRX[RXFFINT] flag just in case it is set

SpibRegs.SPIFFRX.bit.RXFFIENA = 1; // Enable the RX FIFO Interrupt. RXFFST >= RXFFIL

//SpibRegs.SPIFFCT.bit.TXDLY = 16; //Set delay between transmits to 16 spi clocks. Needed by DAN28027 chip

SpibRegs.SPIFFCT.bit.TXDLY = 0;

SpibRegs.SPICCR.bit.SPISWRESET = 1; // Pull the SPI out of reset

SpibRegs.SPIFFTX.bit.TXFIFO = 1; // Release transmit FIFO from reset.

SpibRegs.SPIFFRX.bit.RXFIFORESET = 1; // Re-enable receive FIFO operation

SpibRegs.SPICTL.bit.SPIINTENA = 1; // Enables SPI interrupt. !! I dont think this is needed. Need to Test

SpibRegs.SPIFFRX.bit.RXFFIL =16; //Interrupt Level to 16 words or more received into FIFO causes interrupt. This is just the initial setting for the register. Will be changed below

//-----------------------------------------------------------------------------------------------------------------

//Step 2.

// perform a multiple 16-bit transfer to initialize MPU-9250 registers 0x13,0x14,0x15,0x16

// 0x17, 0x18, 0x19, 0x1A, 0x1B, 0x1C 0x1D, 0x1E, 0x1F. Use only one SS low to high for all these writes

// some code is given, most you have to fill you yourself.

GpioDataRegs.GPCCLEAR.bit.GPIO66 = 1; // Slave Select Low

// Perform the number of needed writes to SPITXBUF to write to all 13 registers. Remember we are sending 16-bit transfers, so two registers at a time after the first 16-bit transfer.

// To address 00x13 write 0x00

SpibRegs.SPITXBUF = (0x1300 | 0x0000);

// To address 00x14 write 0x00

// To address 00x15 write 0x00

SpibRegs.SPITXBUF = (0x0000 | 0x0000);

// To address 00x16 write 0x00

// To address 00x17 write 0x00

SpibRegs.SPITXBUF = (0x0000 | 0x0000);

// To address 00x18 write 0x00

// To address 00x19 write 0x13

SpibRegs.SPITXBUF = (0x0000 | 0x0013);

// To address 00x1A write 0x02

// To address 00x1B write 0x00

SpibRegs.SPITXBUF = (0x0200 | 0x0000);

// To address 00x1C write 0x08

// To address 00x1D write 0x06

SpibRegs.SPITXBUF = (0x0800 | 0x0006);

// To address 00x1E write 0x00

// To address 00x1F write 0x00

SpibRegs.SPITXBUF = (0x0000 | 0x0000);

// wait for the correct number of 16-bit values to be received into the RX FIFO

**while**(SpibRegs.SPIFFRX.bit.RXFFST !=7);

GpioDataRegs.GPCSET.bit.GPIO66 = 1; // Slave Select High

temp = SpibRegs.SPIRXBUF;

temp = SpibRegs.SPIRXBUF;

temp = SpibRegs.SPIRXBUF;

temp = SpibRegs.SPIRXBUF;

temp = SpibRegs.SPIRXBUF;

temp = SpibRegs.SPIRXBUF;

temp = SpibRegs.SPIRXBUF;

// ???? read the additional number of garbage receive values off the RX FIFO to clear out the RX FIFO

DELAY\_US(10); // Delay 10us to allow time for the MPU-2950 to get ready for next transfer.

//Step 3.

// perform a multiple 16-bit transfer to initialize MPU-9250 registers 0x23,0x24,0x25,0x26

// 0x27, 0x28, 0x29. Use only one SS low to high for all these writes

// some code is given, most you have to fill you yourself.

GpioDataRegs.GPCCLEAR.bit.GPIO66 = 1; // Slave Select Low

// Perform the number of needed writes to SPITXBUF to write to all 7 registers

// To address 00x23 write 0x00

SpibRegs.SPITXBUF = (0x2300 | 0x0000);

// To address 00x24 write 0x40

// To address 00x25 write 0x8C

SpibRegs.SPITXBUF = (0x4000 | 0x008C);

// To address 00x26 write 0x02

// To address 00x27 write 0x88

SpibRegs.SPITXBUF = (0x0200 | 0x0088);

// To address 00x28 write 0x0C

// To address 00x29 write 0x0A

SpibRegs.SPITXBUF = (0x0C00 | 0x000A);

// wait for the correct number of 16-bit values to be received into the RX FIFO

**while**(SpibRegs.SPIFFRX.bit.RXFFST !=4);

GpioDataRegs.GPCSET.bit.GPIO66 = 1; // Slave Select High

temp = SpibRegs.SPIRXBUF;

temp = SpibRegs.SPIRXBUF;

temp = SpibRegs.SPIRXBUF;

temp = SpibRegs.SPIRXBUF;

// ???? read the additional number of garbage receive values off the RX FIFO to clear out the RX FIFO

DELAY\_US(10); // Delay 10us to allow time for the MPU-2950 to get ready for next transfer.

//Step 4.

// perform a single 16-bit transfer to initialize MPU-9250 register 0x2A

GpioDataRegs.GPCCLEAR.bit.GPIO66 = 1;

// Write to address 0x2A the value 0x81

SpibRegs.SPITXBUF = (0x2A00 | 0x0081);

// wait for one byte to be received

**while**(SpibRegs.SPIFFRX.bit.RXFFST !=1);

GpioDataRegs.GPCSET.bit.GPIO66 = 1;

temp = SpibRegs.SPIRXBUF;

DELAY\_US(10);

// The Remainder of this code is given to you and you do not need to make any changes.

GpioDataRegs.GPCCLEAR.bit.GPIO66 = 1;

SpibRegs.SPITXBUF = (0x3800 | 0x0001); // 0x3800

**while**(SpibRegs.SPIFFRX.bit.RXFFST !=1);

GpioDataRegs.GPCSET.bit.GPIO66 = 1;

temp = SpibRegs.SPIRXBUF;

DELAY\_US(10);

GpioDataRegs.GPCCLEAR.bit.GPIO66 = 1;

SpibRegs.SPITXBUF = (0x3A00 | 0x0001); // 0x3A00

**while**(SpibRegs.SPIFFRX.bit.RXFFST !=1);

GpioDataRegs.GPCSET.bit.GPIO66 = 1;

temp = SpibRegs.SPIRXBUF;

DELAY\_US(10);

GpioDataRegs.GPCCLEAR.bit.GPIO66 = 1;

SpibRegs.SPITXBUF = (0x6400 | 0x0001); // 0x6400

**while**(SpibRegs.SPIFFRX.bit.RXFFST !=1);

GpioDataRegs.GPCSET.bit.GPIO66 = 1;

temp = SpibRegs.SPIRXBUF;

DELAY\_US(10);

GpioDataRegs.GPCCLEAR.bit.GPIO66 = 1;

SpibRegs.SPITXBUF = (0x6700 | 0x0003); // 0x6700

**while**(SpibRegs.SPIFFRX.bit.RXFFST !=1);

GpioDataRegs.GPCSET.bit.GPIO66 = 1;

temp = SpibRegs.SPIRXBUF;

DELAY\_US(10);

GpioDataRegs.GPCCLEAR.bit.GPIO66 = 1;

SpibRegs.SPITXBUF = (0x6A00 | 0x0020); // 0x6A00

**while**(SpibRegs.SPIFFRX.bit.RXFFST !=1);

GpioDataRegs.GPCSET.bit.GPIO66 = 1;

temp = SpibRegs.SPIRXBUF;

DELAY\_US(10);

GpioDataRegs.GPCCLEAR.bit.GPIO66 = 1;

SpibRegs.SPITXBUF = (0x6B00 | 0x0001); // 0x6B00

**while**(SpibRegs.SPIFFRX.bit.RXFFST !=1);

GpioDataRegs.GPCSET.bit.GPIO66 = 1;

temp = SpibRegs.SPIRXBUF;

DELAY\_US(10);

GpioDataRegs.GPCCLEAR.bit.GPIO66 = 1;

SpibRegs.SPITXBUF = (0x7500 | 0x00EE); // 0x7500

**while**(SpibRegs.SPIFFRX.bit.RXFFST !=1);

GpioDataRegs.GPCSET.bit.GPIO66 = 1;

temp = SpibRegs.SPIRXBUF;

DELAY\_US(10);

GpioDataRegs.GPCCLEAR.bit.GPIO66 = 1;

SpibRegs.SPITXBUF = (0x7700 | 0x0011); // 0x7700

**while**(SpibRegs.SPIFFRX.bit.RXFFST !=1);

GpioDataRegs.GPCSET.bit.GPIO66 = 1;

temp = SpibRegs.SPIRXBUF;

DELAY\_US(10);

GpioDataRegs.GPCCLEAR.bit.GPIO66 = 1;

SpibRegs.SPITXBUF = (0x7800 | 0x0010); // 0x7800

**while**(SpibRegs.SPIFFRX.bit.RXFFST !=1);

GpioDataRegs.GPCSET.bit.GPIO66 = 1;

temp = SpibRegs.SPIRXBUF;

DELAY\_US(10);

GpioDataRegs.GPCCLEAR.bit.GPIO66 = 1;

SpibRegs.SPITXBUF = (0x7A00 | 0x0017); // 0x7A00

**while**(SpibRegs.SPIFFRX.bit.RXFFST !=1);

GpioDataRegs.GPCSET.bit.GPIO66 = 1;

temp = SpibRegs.SPIRXBUF;

DELAY\_US(10);

GpioDataRegs.GPCCLEAR.bit.GPIO66 = 1;

SpibRegs.SPITXBUF = (0x7B00 | 0x0032); // 0x7B00

**while**(SpibRegs.SPIFFRX.bit.RXFFST !=1);

GpioDataRegs.GPCSET.bit.GPIO66 = 1;

temp = SpibRegs.SPIRXBUF;

DELAY\_US(10);

GpioDataRegs.GPCCLEAR.bit.GPIO66 = 1;

SpibRegs.SPITXBUF = (0x7D00 | 0x001C); // 0x7D00

**while**(SpibRegs.SPIFFRX.bit.RXFFST !=1);

GpioDataRegs.GPCSET.bit.GPIO66 = 1;

temp = SpibRegs.SPIRXBUF;

DELAY\_US(10);

GpioDataRegs.GPCCLEAR.bit.GPIO66 = 1;

SpibRegs.SPITXBUF = (0x7E00 | 0x0038); // 0x7E00

**while**(SpibRegs.SPIFFRX.bit.RXFFST !=1);

GpioDataRegs.GPCSET.bit.GPIO66 = 1;

temp = SpibRegs.SPIRXBUF;

DELAY\_US(50);

// Clear SPIB interrupt source just in case it was issued due to any of the above initializations.

SpibRegs.SPIFFRX.bit.RXFFOVFCLR=1; // Clear Overflow flag

SpibRegs.SPIFFRX.bit.RXFFINTCLR=1; // Clear Interrupt flag

PieCtrlRegs.PIEACK.all = PIEACK\_GROUP6;

}

int16\_t spivalue1 = 0;

int16\_t spivalue2 = 0;

int16\_t spivalue3 = 0;

\_\_interrupt **void** **SPIB\_isr**(**void**) {

GpioDataRegs.GPCSET.bit.GPIO66 = 1;

spivalue1 = SpibRegs.SPIRXBUF;

accelx\_raw = SpibRegs.SPIRXBUF;

accely\_raw = SpibRegs.SPIRXBUF;

accelz\_raw = SpibRegs.SPIRXBUF;

spivalue2 = SpibRegs.SPIRXBUF;

gyrox\_raw = SpibRegs.SPIRXBUF;

gyroy\_raw = SpibRegs.SPIRXBUF;

gyroz\_raw = SpibRegs.SPIRXBUF;

accelx = accelx\_raw\*(4.0/32767.0);

accely = accely\_raw\*(4.0/32767.0);

accelz = accelz\_raw\*(4.0/32767.0);

gyrox = gyrox\_raw\*(250.0/32767.0);

gyroy = gyroy\_raw\*(250.0/32767.0);

gyroz = gyroz\_raw\*(250.0/32767.0);

**if** ((SpibNumCalls % 200) == 0) {

UARTPrint = 1;

}

SpibNumCalls++;

SpibRegs.SPIFFRX.bit.RXFFOVFCLR=1; // Clear Overflow flag

SpibRegs.SPIFFRX.bit.RXFFINTCLR=1; // Clear Interrupt flag

PieCtrlRegs.PIEACK.all = PIEACK\_GROUP6;

}

// ----- code for CAN start here -----

\_\_interrupt **void** **can\_isr**(**void**)

{

**int** i = 0;

uint32\_t status;

GpioDataRegs.GPBSET.bit.GPIO52 = 1;

//

// Read the CAN interrupt status to find the cause of the interrupt

//

status = CANgetInterruptCause(CANB\_BASE);

//

// If the cause is a controller status interrupt, then get the status

//

**if**(status == CAN\_INT\_INT0ID\_STATUS)

{

//

// Read the controller status. This will return a field of status

// error bits that can indicate various errors. Error processing

// is not done in this example for simplicity. Refer to the

// API documentation for details about the error status bits.

// The act of reading this status will clear the interrupt.

//

status = CANgetStatus(CANB\_BASE);

}

//

// Check if the cause is the transmit message object 1

//

// else if(status == TX\_MSG\_OBJ\_ID)

// {

// //

// // Getting to this point means that the TX interrupt occurred on

// // message object 1, and the message TX is complete. Clear the

// // message object interrupt.

// //

// CANclearInterruptStatus(CANB\_BASE, TX\_MSG\_OBJ\_ID);

//

// //

// // Since the message was sent, clear any error flags.

// //

// errorFlag = 0;

// }

//

// Check if the cause is the receive message object 2

//

**else** **if**(status == RX\_MSG\_OBJ\_ID\_1)

{

//

// Get the received message

//

CANreadMessage(CANB\_BASE, RX\_MSG\_OBJ\_ID\_1, rxMsgData);

**for**(i = 0; i<2; i++)

{

dis\_raw\_1[i] = rxMsgData[i];

}

dis\_1 = 256\*dis\_raw\_1[1] + dis\_raw\_1[0];

measure\_status\_1 = rxMsgData[2];

//

// Getting to this point means that the RX interrupt occurred on

// message object 2, and the message RX is complete. Clear the

// message object interrupt.

//

CANclearInterruptStatus(CANB\_BASE, RX\_MSG\_OBJ\_ID\_1);

//

// Increment a counter to keep track of how many messages have been

// received. In a real application this could be used to set flags to

// indicate when a message is received.

//

rxMsgCount\_1++;

//

// Since the message was received, clear any error flags.

//

errorFlag = 0;

GpioDataRegs.GPBCLEAR.bit.GPIO52 = 1;

}

**else** **if**(status == RX\_MSG\_OBJ\_ID\_2)

{

//

// Get the received message

//

CANreadMessage(CANB\_BASE, RX\_MSG\_OBJ\_ID\_2, rxMsgData);

**for**(i = 0; i<2; i++)

{

dis\_raw\_3[i] = rxMsgData[i];

}

dis\_3 = 256\*dis\_raw\_3[1] + dis\_raw\_3[0];

measure\_status\_3 = rxMsgData[2];

//

// Getting to this point means that the RX interrupt occurred on

// message object 2, and the message RX is complete. Clear the

// message object interrupt.

//

CANclearInterruptStatus(CANB\_BASE, RX\_MSG\_OBJ\_ID\_2);

//

// Increment a counter to keep track of how many messages have been

// received. In a real application this could be used to set flags to

// indicate when a message is received.

//

rxMsgCount\_3++;

//

// Since the message was received, clear any error flags.

//

errorFlag = 0;

GpioDataRegs.GPBCLEAR.bit.GPIO52 = 1;

}

**else** **if**(status == RX\_MSG\_OBJ\_ID\_3)

{

//

// Get the received message

//

CANreadMessage(CANB\_BASE, RX\_MSG\_OBJ\_ID\_3, rxMsgData);

**for**(i = 0; i<4; i++)

{

lightlevel\_raw\_1[i] = rxMsgData[i];

quality\_raw\_1[i] = rxMsgData[i+4];

}

lightlevel\_1 = ((256.0\*256.0\*256.0)\*lightlevel\_raw\_1[3] + (256.0\*256.0)\*lightlevel\_raw\_1[2] + 256.0\*lightlevel\_raw\_1[1] + lightlevel\_raw\_1[0])/65535;

quality\_1 = ((256.0\*256.0\*256.0)\*quality\_raw\_1[3] + (256.0\*256.0)\*quality\_raw\_1[2] + 256.0\*quality\_raw\_1[1] + quality\_raw\_1[0])/65535;

//

// Getting to this point means that the RX interrupt occurred on

// message object 2, and the message RX is complete. Clear the

// message object interrupt.

//

CANclearInterruptStatus(CANB\_BASE, RX\_MSG\_OBJ\_ID\_3);

//

// Since the message was received, clear any error flags.

//

errorFlag = 0;

GpioDataRegs.GPBCLEAR.bit.GPIO52 = 1;

}

**else** **if**(status == RX\_MSG\_OBJ\_ID\_4)

{

//

// Get the received message

//

CANreadMessage(CANB\_BASE, RX\_MSG\_OBJ\_ID\_4, rxMsgData);

**for**(i = 0; i<4; i++)

{

lightlevel\_raw\_3[i] = rxMsgData[i];

quality\_raw\_3[i] = rxMsgData[i+4];

}

lightlevel\_3 = ((256.0\*256.0\*256.0)\*lightlevel\_raw\_3[3] + (256.0\*256.0)\*lightlevel\_raw\_3[2] + 256.0\*lightlevel\_raw\_3[1] + lightlevel\_raw\_3[0])/65535;

quality\_3 = ((256.0\*256.0\*256.0)\*quality\_raw\_3[3] + (256.0\*256.0)\*quality\_raw\_3[2] + 256.0\*quality\_raw\_3[1] + quality\_raw\_3[0])/65535;

//

// Getting to this point means that the RX interrupt occurred on

// message object 2, and the message RX is complete. Clear the

// message object interrupt.

//

CANclearInterruptStatus(CANB\_BASE, RX\_MSG\_OBJ\_ID\_4);

//

// Since the message was received, clear any error flags.

//

errorFlag = 0;

GpioDataRegs.GPBCLEAR.bit.GPIO52 = 1;

}

//

// If something unexpected caused the interrupt, this would handle it.

//

**else**

{

//

// Spurious interrupt handling can go here.

//

}

//

// Clear the global interrupt flag for the CAN interrupt line

//

CANclearGlobalInterruptStatus(CANB\_BASE, CAN\_GLOBAL\_INT\_CANINT0);

//

// Acknowledge this interrupt located in group 9

//

InterruptclearACKGroup(INTERRUPT\_ACK\_GROUP9);

}

// ----- code for CAN end here -----

// DW - copied code with multiplication factor that converts eQEP counts to

// the number of radians the wheel has turned.

**void** **init\_eQEPs**(**void**) {

// setup eQEP1 pins for input

EALLOW;

//Disable internal pull-up for the selected output pins for reduced power consumption

GpioCtrlRegs.GPAPUD.bit.GPIO20 = 1; // Disable pull-up on GPIO20 (EQEP1A)

GpioCtrlRegs.GPAPUD.bit.GPIO21 = 1; // Disable pull-up on GPIO21 (EQEP1B)

GpioCtrlRegs.GPAQSEL2.bit.GPIO20 = 2; // Qual every 6 samples

GpioCtrlRegs.GPAQSEL2.bit.GPIO21 = 2; // Qual every 6 samples

EDIS;

// This specifies which of the possible GPIO pins will be EQEP1 functional pins.

// Comment out other unwanted lines.

**GPIO\_SetupPinMux**(20, GPIO\_MUX\_CPU1, 1);

**GPIO\_SetupPinMux**(21, GPIO\_MUX\_CPU1, 1);

EQep1Regs.QEPCTL.bit.QPEN = 0; // make sure eqep in reset

EQep1Regs.QDECCTL.bit.QSRC = 0; // Quadrature count mode

EQep1Regs.QPOSCTL.all = 0x0; // Disable eQep Position Compare

EQep1Regs.QCAPCTL.all = 0x0; // Disable eQep Capture

EQep1Regs.QEINT.all = 0x0; // Disable all eQep interrupts

EQep1Regs.QPOSMAX = 0xFFFFFFFF; // use full range of the 32 bit count

EQep1Regs.QEPCTL.bit.FREE\_SOFT = 2; // EQep uneffected by emulation suspend in Code Composer

EQep1Regs.QPOSCNT = 0;

EQep1Regs.QEPCTL.bit.QPEN = 1; // Enable EQep

// setup QEP2 pins for input

EALLOW;

//Disable internal pull-up for the selected output pinsfor reduced power consumption

GpioCtrlRegs.GPBPUD.bit.GPIO54 = 1; // Disable pull-up on GPIO54 (EQEP2A)

GpioCtrlRegs.GPBPUD.bit.GPIO55 = 1; // Disable pull-up on GPIO55 (EQEP2B)

GpioCtrlRegs.GPBQSEL2.bit.GPIO54 = 2; // Qual every 6 samples

GpioCtrlRegs.GPBQSEL2.bit.GPIO55 = 2; // Qual every 6 samples

EDIS;

**GPIO\_SetupPinMux**(54, GPIO\_MUX\_CPU1, 5); // set GPIO54 and eQep2A

**GPIO\_SetupPinMux**(55, GPIO\_MUX\_CPU1, 5); // set GPIO54 and eQep2B

EQep2Regs.QEPCTL.bit.QPEN = 0; // make sure qep reset

EQep2Regs.QDECCTL.bit.QSRC = 0; // Quadrature count mode

EQep2Regs.QPOSCTL.all = 0x0; // Disable eQep Position Compare

EQep2Regs.QCAPCTL.all = 0x0; // Disable eQep Capture

EQep2Regs.QEINT.all = 0x0; // Disable all eQep interrupts

EQep2Regs.QPOSMAX = 0xFFFFFFFF; // use full range of the 32 bit count.

EQep2Regs.QEPCTL.bit.FREE\_SOFT = 2; // EQep uneffected by emulation suspend

EQep2Regs.QPOSCNT = 0;

EQep2Regs.QEPCTL.bit.QPEN = 1; // Enable EQep

}

//DW- Since one rotation of the motor counts 100 square waves per revolution and the eQEP counts this in quadrature count mode, the resulting number of counts per revolution will be 400 counts per revolution. Also, the gear ratio of the DC motor relative to the wheel is 30:1, we can calculate the radians of the wheel has turned by multiplying it with (2pi/12000). If we simplify this multiplication factor, we get (pi/6000). We also add a negative sign to the left encoder since the left motor gives a negative when the robot is moving in the forward direction.

**float** **readEncLeft**(**void**) {

int32\_t raw = 0;

uint32\_t QEP\_maxvalue = 0xFFFFFFFFU; //4294967295U

raw = EQep1Regs.QPOSCNT;

**if** (raw >= QEP\_maxvalue/2) raw -= QEP\_maxvalue; // I don't think this is needed and never true

// 100 slits in the encoder disk so 100 square waves per one revolution of the

// DC motor's back shaft. Then Quadrature Decoder mode multiplies this by 4 so 400 counts per one rev

// of the DC motor's back shaft. Then the gear motor's gear ratio is 30:1.

**return** (raw\*(-PI/6000));

}

**float** **readEncRight**(**void**) {

int32\_t raw = 0;

uint32\_t QEP\_maxvalue = 0xFFFFFFFFU; //4294967295U -1 32bit signed int

raw = EQep2Regs.QPOSCNT;

**if** (raw >= QEP\_maxvalue/2) raw -= QEP\_maxvalue; // I don't think this is needed and never true

// 100 slits in the encoder disk so 100 square waves per one revolution of the

// DC motor's back shaft. Then Quadrature Decoder mode multiplies this by 4 so 400 counts per one rev

// of the DC motor's back shaft. Then the gear motor's gear ratio is 30:1.

**return** (raw\*(PI/6000));

}

**void** **setEPWM2A**(**float** controleffort) {

**if** (controleffort > 10) {

controleffort = 10;

} **else** **if** (controleffort < -10) {

controleffort = -10;

}

controleffort = 125\*(controleffort + 10);

EPwm2Regs.CMPA.bit.CMPA = controleffort;

}

**void** **setEPWM2B**(**float** controleffort) {

**if** (controleffort > 10) {

controleffort = 10;

} **else** **if** (controleffort < -10) {

controleffort = -10;

}

controleffort = 125\*(controleffort + 10);

EPwm2Regs.CMPB.bit.CMPB = controleffort;

}

//DW - Exercise 3

**void** **setEPWM8A**(**float** angle1) {

**if** (angle1 > 90) {

angle1 = 90;

} **else** **if** (angle1 < -90) {

angle1 = -90;

}

**float** setVal = 27.778\*(angle1 + 180.0);

EPwm8Regs.CMPA.bit.CMPA = setVal;

}

**void** **setEPWM8B**(**float** angle2) {

**if** (angle2 > 90) {

angle2 = 90;

} **else** **if** (angle2 < -90) {

angle2 = -90;

}

**float** setVal = 27.778\*(angle2 + 180);

EPwm8Regs.CMPB.bit.CMPB = setVal;

}

//DW - turns buzzer on by changing duty cycle from 0% to 50%

**void** **setEPWM9\_buzzer**() {

EPwm9Regs.CMPA.bit.CMPA = EPwm9Regs.TBPRD / 2;

}

//DW - turns buzzer off by changing duty cycle from 60% to 0%

**void** **stopEPWM9\_buzzer**() {

EPwm9Regs.CMPA.bit.CMPA = 0;

}