```
/*
 * Team Space
 * 1.30.10
 * Lab 8
 * Beacon Module
 *
 * Andi Kleissner
 * Eli Michael
 * Elico Teixeira
 * Nina Joshi
 */


#ifndef _BEACON_H_
#define _BEACON_H_

#include "events.h"

Event_t BEACON_CheckForBeacon(void);
void BEACON_Debug(void);

#endif
```

```
/*
 * Team Space
 * 2.3.10
 * Lab 8
 * Beacon Module
 *
 * Andi Kleissner
 * Eli Michael
 * Elico Teixeira
 * Nina Joshi
 */


#include "defs.h"
#include "beacon.h"

/* Variables */
// The previous beacon event.
static Event_t lastEvent = NO_EVENT;
// The current beacon event.
static Event_t thisEvent = NO_EVENT;

/* Functions */
/* Returns a BEACON_ON event if the phototransistor signal is between the specified thresholds.
 * Otherwise, will return BEACON_OFF event. Will only return event if different from previous.*/
Event_t BEACON_CheckForBeacon() {
	Event_t event = NO_EVENT;

	if ((ADS12_ReadADPin(BEACON_BIT) > BEACON_ON_LOW_THRESHOLD) &&
	    (ADS12_ReadADPin(BEACON_BIT) < BEACON_ON_HIGH_THRESHOLD)) {
	    event = BEACON_ON;
	}   else {
	  event = BEACON_OFF;
	}

	if (event != lastEvent) {
	    lastEvent = event;
	    return event;
	}

	return NO_EVENT;
}

/* Beacon debug. */
void BEACON_Debug(void) {
	(void)printf("\nDebugging BEACON...Controls:\r\n");
	(void)printf("q: quit\r\n");
	// Break out of loop when user hits 'q'
	for(;;) {
	    // Prints beacon value until user quits.
	    while (!kbhit()) {
			Event_t event = BEACON_CheckForBeacon();
			if (event == BEACON_ON) {
				(void)printf("Beacon ON: %d\r\n", ADS12_ReadADPin(BEACON_BIT));
			} else if (event == BEACON_OFF) {
			  (void)printf("Beacon OFF: %d\r\n", ADS12_ReadADPin(BEACON_BIT));
```

```
                } else {
                  //(void)printf("%d\r\n", ADS12_ReadADPin(BEACON_BIT));
                }
        }
        // Clears input character and returns from the function.
        if (getchar() == 'q') {
                lastEvent = NO_EVENT;
                return;
        }
    }
}
```

```
/*
 * Team Space
 * 2.3.10
 * Lab 8
 * Definitions
 *
 * Andi Kleissner
 * Eli Michael
 * Elico Teixeira
 * Nina Joshi
 */


#ifndef _DEFS_H_
#define _DEFS_H_

#include <stdio.h>
#include <hidef.h> // common defines and macros
#include <mc9s12e128.h> // derivative information
#include <S12E128bits.h> // E128 bit definitions
#include <bitdefs.h> // BIT0HI, BIT0LO....definitions
#include <ADS12.h> //.c file is ADS12e.c
#include "S12eVec.h"
#include "ME218_E128.h"

/* PWM and Timing */
#define PWM_PERIOD 150 //10kHz

/* Port U bits: 0-7 */ //PWM
#define LEFT_DUTY_BIT 0
#define LEFT_DUTY_DDR DDRU
#define LEFT_DUTY_PORT PTU

#define RIGHT_DUTY_BIT 1
#define RIGHT_DUTY_DDR DDRU
#define RIGHT_DUTY_PORT PTU

#define LEFT_DIR_BIT 2
#define LEFT_DIR_DDR DDRU
#define LEFT_DIR_PORT PTU

#define RIGHT_DIR_BIT 3
#define RIGHT_DIR_DDR DDRU
#define RIGHT_DIR_PORT PTU

/* Port E bits: 0-1 */

/* Port T bits: 0-7 */ //Timers

/* Port S bits: 2-7 */

/* Port P bits: 0-5 */
#define TAPE_STATE_BIT 0
#define TAPE_STATE_DDR DDRP
#define TAPE_STATE_PORT PTP
```

```
#define ALIGN_STATE_BIT 1
#define ALIGN_STATE_DDR DDRP
#define ALIGN_STATE_PORT PTP

#define BEACON_ON_BIT 2
#define BEACON_ON_DDR DDRP
#define BEACON_ON_PORT PTP

/* Port AD bits: 0-7 */
#define AD_Init_String "OOOOOAAA" //bits "76543210"

#define BEACON_BIT 0
#define BEACON_DDR DDRAD
#define BEACON_PORT PTAD

#define TAPE_BIT 2
#define TAPE_DDR DDRAD
#define TAPE_PORT PTAD

/* Beacon thresholds */
#define BEACON_ON_LOW_THRESHOLD 2
#define BEACON_ON_HIGH_THRESHOLD 1023

/* Tape threshold*/
#define TAPE_THRESHOLD 200 //daylight //night time: 440

/* Timers */
#define ROTATE90_TIMER 0
#define ROTATE90_TIME 1250

#define ROTATE45_TIMER 1
#define ROTATE45_TIME 625

/* Wheels */
#define FULL 100
#define HALF 65

#define FORWARD 0
#define REVERSE 1

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

#define LOW 0
#define HIGH 1

#define LEFT 0
#define RIGHT 1

#define SETBIT(ADDRESS,BIT) (ADDRESS |= (1<<BIT)) //set bit (high)      //output
#define CLEARBIT(ADDRESS,BIT) (ADDRESS &= ~(1<<BIT)) //clear bit (low)  //input
#define CHECKBIT(ADDRESS,BIT) (ADDRESS & (1<<BIT)) //test single bit in I/O location
#define TOGGLEBIT(ADDRESS,BIT) (ADDRESS ^= (1<<BIT)) //toggle single bit

#endif
```

ME218b: Lab 8 Code

Nina Joshi, Andi Kleissner, Eli Michaels, Elico Teixeira

```
/*
 * Team Space
 * 2.3.10
 * Lab 8
 * Events Module
 *
 * Andi Kleissner
 * Eli Michael
 * Elico Teixeira
 * Nina Joshi
 */


#ifndef _EVENTS_H_
#define _EVENTS_H_

typedef enum {
	NO_EVENT,
	BEACON_ON,
	BEACON_OFF,
	TAPE_FOUND,
	ROTATE90_TIMER_EXPIRED,
	ROTATE45_TIMER_EXPIRED,
	NEW_COMMAND
} Event_t;

Event_t EVENTS_CheckForEvents(void);
void EVENTS_Debug(void);

/* Helper function to check if any timers have expired since no separate routine for that. */
static Event_t CheckForTimerExpired(void);

#endif
```

```
/*
 * Team Space
 * 2.3.10
 * Lab 8
 * Events Module
 *
 * Andi Kleissner
 * Eli Michael
 * Elico Teixeira
 * Nina Joshi
 */


#include "defs.h"
#include "events.h"
#include "beacon.h"
#include "pwm.h"
#include "tape.h"
#include "spi.h"
#include "timer.h"
#include "wheels.h"

/* Functions */
/* Checks and returns any events that have occurred. */
Event_t EVENTS_CheckForEvents(void) {
	Event_t event = NO_EVENT;

	//Check if command given
	event = SPI_CheckNewCommand();
	if (event != NO_EVENT) {
	    return event;
	}

	//Check if a timer expired
	event = CheckForTimerExpired();
	if (event != NO_EVENT)
	    return event;

	//Check if tape event has occured
	event = TAPE_CheckForTape();
	if (event != NO_EVENT)
	    return event;

	//Check if beacon event has occured
	event = BEACON_CheckForBeacon();
	if (event != NO_EVENT)
	    return event;

	return event;
}

/* Checks all timers and if any has expired, clear the timer and return the corresponding event. */
static Event_t CheckForTimerExpired() {
	if(TIMER0_IsTimerExpired(ROTATE90_TIMER) == TIMER0_EXPIRED) {
	    TIMER0_ClearTimerExpired(ROTATE90_TIMER);
	    return ROTATE90_TIMER_EXPIRED;
```

```
    }
    if(TIMER0_IsTimerExpired(ROTATE45_TIMER) == TIMER0_EXPIRED) {
        TIMER0_ClearTimerExpired(ROTATE45_TIMER);
        return ROTATE45_TIMER_EXPIRED;
    }

    return NO_EVENT;
}

/* Events debug. */
void EVENTS_Debug(void){
    (void)printf("\nDebugging events...Controls:\r\n");
    (void)printf("n: 90 timer, f: 45 timer, q: quit \r\n");
    // Break out of loop when user hits 'q'
    for(;;) {
    if (kbhit()) {
            // User selects options.
            char ch = getchar();
            switch (ch) {
                case 'n':
                    (void)printf("Setting rotate 90 timer...\r\n");
                    TIMER0_InitTimer(ROTATE90_TIMER, ROTATE90_TIME);
                    break;
                case 'f':
                    (void)printf("Setting rotate 45 timer...\r\n");
                    TIMER0_InitTimer(ROTATE45_TIMER, ROTATE45_TIME);
                    break;
                case 'q':
                    return;
                default:
                    break;
            }
    }
        // Print out event that has occurred.
        while (!kbhit()) {
         switch(EVENTS_CheckForEvents()) {
            case NO_EVENT:
                //printf("No event\r\n");
                break;
            case BEACON_ON:
                (void)printf("Beacon on\r\n");
                break;
            case BEACON_OFF:
                (void)printf("Beacon off\r\n");
                break;
            case TAPE_FOUND:
                (void)printf("Tape found\r\n");
                break;
            case ROTATE90_TIMER_EXPIRED:
                (void)printf("Rotate 90 timer expired\r\n");
                break;
            case ROTATE45_TIMER_EXPIRED:
                (void)printf("Rotate 45 timer expired\r\n");
                break;
         case NEW_COMMAND:
                (void)printf("New command received\r\n");
```

```
                        break;
                default:
                        (void)printf("Error: undefined event!\r\n");
                        break;
            }
        }
    }
}
```

```
/*
 * Team Space
 * 2.3.10
 * Lab 8
 * Main Module
 *
 * Andi Kleissner
 * Eli Michael
 * Elico Teixeira
 * Nina Joshi
 */


#include "defs.h"
#include "wheels.h"
#include "beacon.h"
#include "events.h"
#include "pwm.h"
#include "tape.h"
#include "timer.h"
#include "state.h"
#include "spi.h"

#pragma LINK_INFO DERIVATIVE "SampleS12"

/* Prototypes */
static void MAIN_Debug(void);
static void MAIN_Init(void);

/* Functions */
void main (void) {
   TIMER0_Init(TIMERS_RATE_1MS); //Initialize timers
   PWM_Init(); //Initialize PWM
   MAIN_Init(); //Initialize ports
   SPI_Init(); //Initialize SPI communication

   //Uncomment the following line to debug and not run the state machine.
   //MAIN_Debug();

   //Initialize the state machine and loop forever to run it.
        STATE_InitStateMachine();
   for(;;) {
          STATE_RunStateMachine(EVENTS_CheckForEvents());
       }
}

/* Initialize port data directions for outputs and inputs, initialize pwms, and set opto output low. */
static void MAIN_Init(void) {
       //set outputs
       SETBIT(LEFT_DUTY_DDR, LEFT_DUTY_BIT);
       SETBIT(RIGHT_DUTY_DDR, RIGHT_DUTY_BIT);
       SETBIT(LEFT_DIR_DDR, LEFT_DIR_BIT);
       SETBIT(RIGHT_DIR_DDR, RIGHT_DIR_BIT);
       //clear inputs

       (void)ADS12_Init(AD_Init_String); //ad port
```

```
        //Debugging LEDs
        SETBIT(BEACON_ON_DDR, BEACON_ON_BIT);
        SETBIT(TAPE_STATE_DDR, TAPE_STATE_BIT);
        SETBIT(ALIGN_STATE_DDR, ALIGN_STATE_BIT);

        WHEELS_StopBoth();
}

/* Will always loop as it runs debugging tests for each module. These debugging
 * tests are ONLY for debugging and have blocking code. */
static void MAIN_Debug(void) {
        (void)printf("\nDebugging...\r\n");
        (void)printf("\nNote that debugging is case sensitive.\r\n");
        for(;;) {
            WHEELS_Debug();
            BEACON_Debug();
            TAPE_Debug();
            SPI_Debug();
            EVENTS_Debug(); //includes timers
            PWM_Debug();
        }
}
```

```
/*
 * Team Space
 * 1.30.10
 * Lab 8
 * PWM Module
 *
 * Andi Kleissner
 * Eli Michael
 * Elico Teixeira
 * Nina Joshi
 */


#ifndef _PWM_H_
#define _PWM_H_

#include "events.h"

void PWM_Init(void);
void PWM_SetDuty(unsigned char wheel, unsigned char duty, unsigned char direction);
void PWM_Debug(void);

#endif
```

```
/*
 * Team Space
 * 2.3.10
 * Lab 8
 * PWM Module
 *
 * Andi Kleissner
 * Eli Michael
 * Elico Teixeira
 * Nina Joshi
 */


#include "defs.h"
#include "pwm.h"

/* Prototypes */
static void SetDuty(unsigned char duty, unsigned char wheel);
static void SetDirection(unsigned char direction, unsigned char wheel);

/* Functions */
/* Initializes PWM functions for wheels. */
void PWM_Init(void) {
        //initializes port U0 to be used by the PWM subsystem
        PWME |= _S12_PWME0; //enable PWM channel 0 for left wheel
        //initializes port U1 to be used by the PWM subsystem
        PWME |= _S12_PWME1; //enable PWM channel 1 for right wheel

        MODRR |= _S12_MODRR0; //give U0 pin control to PWM subsystem for left wheel
        MODRR |= _S12_MODRR1; //give U1 pin control to PWM subsystem for right wheel

        //10 kHz
        PWMPRCLK = PWMPRCLK & ~_S12_PCKA0 & ~_S12_PCKA1 | _S12_PCKA2; //prescale factor
16
        PWMCLK &= ~_S12_PCLK0; //set to clock A

        PWMPER0 = PWM_PERIOD; //set left wheel period in ticks
        PWMPER1 = PWM_PERIOD; //set right wheel period in ticks
}

/* Sets PWM specifically for wheels: which wheel, duty cycle, and which direction. */
void PWM_SetDuty(unsigned char wheel, unsigned char duty, unsigned char direction) {
  SetDirection(direction, wheel);
  SetDuty(duty, wheel);
}

/* Sets rotational direction of a particular wheel. */
static void SetDirection(unsigned char direction, unsigned char wheel) {
  if (wheel == LEFT) {
    if (direction == FORWARD) {
      SETBIT(LEFT_DIR_PORT, LEFT_DIR_BIT); //Forward
      PWMPOL &= ~_S12_PPOL0; //set polarity opposite
    }
    else if (direction == REVERSE) {
      CLEARBIT(LEFT_DIR_PORT, LEFT_DIR_BIT); //Reverse
      PWMPOL |= _S12_PPOL0; //set polarity normal
```

```
    }
  } else if (wheel == RIGHT) {
    if (direction == FORWARD) {
      CLEARBIT(RIGHT_DIR_PORT, RIGHT_DIR_BIT); //Forward
      PWMPOL |= _S12_PPOL1; //set polarity opposite
    }
    else if (direction == REVERSE) {
      SETBIT(RIGHT_DIR_PORT, RIGHT_DIR_BIT); //Reverse
      PWMPOL &= ~_S12_PPOL1; //set polarity normal
    }
  }
}

/* Sets duty cycle of a particular wheel. */
static void SetDuty(unsigned char duty, unsigned char wheel) {
        if (wheel == LEFT) { //set duty as a fraction of the peiod in clock ticks
          PWMDTY0 = (unsigned char) ((long) duty * PWM_PERIOD / 100);
  } else if (wheel == RIGHT) { //set duty as a fraction of the peiod in clock ticks
    PWMDTY1 = (unsigned char) ((long) duty * PWM_PERIOD / 100);
  }
}

/* PWM debug. */
void PWM_Debug(void) {
        (void)printf("\nDebugging PWM...Controls:\r\n");
        (void)printf("0: 0%, 1: 10%, 2: 20%, 3: 30%, 4: 40%, 5: 50%\r\n");
        (void)printf("6: 60%, 7: 70%, 8: 80%, 9: 90%, h: 100%, q: quit\r\n");
        // Break out of loop when user hits 'q'
        for(;;) {
            // Clears input character and returns from the function.
            if (kbhit()) {
            // User selects options.
                        char ch = (char) getchar();
                        switch (ch) {
                          case '0':
                            PWM_SetDuty(LEFT, 0, FORWARD);
                                    PWM_SetDuty(RIGHT, 0, FORWARD);
                                    break;
                          case '1':
                            PWM_SetDuty(LEFT, 10, FORWARD);
                                    PWM_SetDuty(RIGHT, 10, FORWARD);
                                    break;
                                  case '2':
                            PWM_SetDuty(LEFT, 20, FORWARD);
                                    PWM_SetDuty(RIGHT, 20, FORWARD);
                                    break;
                                  case '3':
                            PWM_SetDuty(LEFT, 30, FORWARD);
                                    PWM_SetDuty(RIGHT, 30, FORWARD);
                                    break;
                                  case '4':
                            PWM_SetDuty(LEFT, 40, FORWARD);
                                    PWM_SetDuty(RIGHT, 40, FORWARD);
                                    break;
                          case '5':
                            PWM_SetDuty(LEFT, 50, FORWARD);
```

```
                    PWM_SetDuty(RIGHT, 50, FORWARD);
                    break;
                case '6':
                  PWM_SetDuty(LEFT, 60, FORWARD);
                    PWM_SetDuty(RIGHT, 60, FORWARD);
                    break;
                  case '7':
                  PWM_SetDuty(LEFT, 70, FORWARD);
                    PWM_SetDuty(RIGHT, 70, FORWARD);
                    break;
                  case '8':
                  PWM_SetDuty(LEFT, 80, FORWARD);
                    PWM_SetDuty(RIGHT, 80, FORWARD);
                    break;
                  case '9':
                  PWM_SetDuty(LEFT, 90, FORWARD);
                    PWM_SetDuty(RIGHT, 90, FORWARD);
                    break;
                  case 'h':
                  PWM_SetDuty(LEFT, 100, FORWARD);
                    PWM_SetDuty(RIGHT, 100, FORWARD);
                    break;
                  case 'q':
                        (void)printf("QUIT\r\n");
                        return;
                  default:
                        //(void)printf("ERROR\r\n");
                        break;
            }
        }
    }
}
```

```
/*****************************************************
Name: spi.h
Functionality: Communicates with the 'Command' chip using
SPI communication protocal.

Public Functions:
SPI_Init(): Will initialize the SPI system.
SPI_CheckNewCommand(): returns if there is a new command from the
  chip
SPI_GetCommand(): Will query the command chip and return the
command (State_t type) the chip sends.

Author:  Eli Michael
Date: Jan, 2010
Version:  1.0
*****************************************************/
#ifndef _SPI_H_
#define _SPI_H_

#include "events.h"
#include "state.h"

void SPI_Debug(void);
void SPI_Init(void);
Event_t SPI_CheckNewCommand(void);
State_t SPI_GetCommand(void);

#endif
```

```
/*****************************************************
Name: spi.c
Functionality: Communicates with the 'Command' chip using
SPI communication protocal. The command chip is queried
every COMMAND_SAMPLE_INTEVAL by a timer OC.  Once
a single ping/query communication is initiated SPI interrupts
take care of the rest.

Public Functions:
SPI_Init(): Will initialize the SPI system.
SPI_CheckNewCommand(): returns whether or not a new command
  has happened
SPI_GetCommand(): Will  return the latest command.

Authors:  Eli Michael, Andi Kleissner
with Team SPACE: Elico Teixeira and Nina Joshi
Date: Feb, 2010
Version:  3.0

Notes:
Interrupt based.
Uses timer OC channel 0, timer 5

Right now the baud rate is set to the slowest possible. We
need to determine how fast the Command chip can transmit.

*****************************************************/
#include <stdio.h>
#include <hidef.h>      /* common defines and macros */
#include <mc9s12e128.h>     /* derivative information */
#include <S12E128bits.h> /* E128 bit definitions  */
#include <bitdefs.h> /* BIT0HI, BIT0LO....definitions */
#include "S12eVec.h"
#include "ME218_E128.h"
#include "spi.h"
#include "events.h"
#include "state.h"
#include "wheels.h"
#include "timer.h"
#pragma LINK_INFO DERIVATIVE "SampleS12"

#define COMMAND_SAMPLE_RATE 60000 //40ms with timer inited at 1ms

//*************************************************************
//Public Functions Prototypes
//*************************************************************
void SPI_Debug(void);
void SPI_Init(void);
Event_t SPI_CheckNewCommand(void);
State_t SPI_GetCommand(void);

//*************************************************************
//Private Functions Prototypes
//*************************************************************

//*************************************************************
```

```
//Module Level Variables
//*************************************************************
static unsigned int newcommand = 0;
static State_t command = STOP_BOTH;

//*************************************************************
//Module Code
//*************************************************************
void SPI_Init(void){
  //set the baud rate
  //prescale
  SPIBR |= (_S12_SPPR2 | _S12_SPPR1 | _S12_SPPR0);
  //scale
  SPIBR |= (_S12_SPR2 | _S12_SPR1 | _S12_SPR0);

/*111.6   //prescale
  SPIBR |= (_S12_SPPR2 | _S12_SPPR1);
  //scale
  SPIBR |= (_S12_SPR2 );
  */
  //enable SPI system
  SPICR1 |= _S12_SPE;
  //set for master
  SPICR1 |= _S12_MSTR;
  //set MSB first
  //SPICR1 &= _S12_LSBFE;
  //set polarity active low
  SPICR1 |= _S12_CPOL;
  //set clock phasae sample even edges
  SPICR1 |= _S12_CPHA;
  //SS pin control
  SPICR1 |= _S12_SSOE;
  SPICR2 |= _S12_MODFEN;
  //enable SPI interrupt
  SPICR1 |= _S12_SPIE;

        //Setup OC5 to time the command updates
        TIM0_TIOS = TIM0_TIOS | _S12_IOS5;    //Set IOC5 of the timer to be an output compare
        TIM0_TCTL1 = TIM0_TCTL1 & ~(_S12_OL5 | _S12_OM5);   //No pin connected to IOC4, which
means pin PT0 remains free
        TIM0_TC5 = TIM0_TCNT + COMMAND_SAMPLE_RATE;    //Set the first output capture to
happen 40ms into the future
        TIM0_TFLG1 = _S12_C5F;  //Clear the flag for the IOC5
        TIM0_TIE = TIM0_TIE | _S12_C5F;   //enable the interrupt for IOC5

  EnableInterrupts;
}


State_t SPI_GetCommand(void){
  return command;
}

Event_t SPI_CheckNewCommand(void){
  if(newcommand == 1) {
    newcommand = 0;
```

```
   return NEW_COMMAND;
  }
  else {
   return NO_EVENT;
  }
}

void interrupt _Vec_tim0ch5 SPI_OC (void){
  TIM0_TFLG1 = _S12_C5F;  //Clear the flag for the IOC5
        TIM0_TC5 += COMMAND_SAMPLE_RATE;                                    //Update for the next
interrupt
        //check to make sure the trx reg is clear
        if((SPISR & _S12_SPTEF) == _S12_SPTEF) {
          SPIDR = 0xAA;
        }
}

void interrupt _Vec_spi SPI(void) {
  static unsigned char ping = 1;
  static unsigned char read = 0xff;
  static unsigned char readnewcommand = 0;  //tells if last data was 0xFF
  if(readnewcommand == 0) {
    if(ping == 1) {
      //next time we really want the data
      ping = 0;
      //read status reg to clear
      read = SPISR;
      //read garbage
      read = SPIDR;
      //send for the real command from the chip
      SPIDR = 0xAA;
    }
    else {
      ping = 1;
      //read status reg to clear
      read = SPISR;
      //read garbage
      read = SPIDR;
      //this time we really want the data
      if(read == 0xFF) {
        readnewcommand = 1;
      }
    }
  }
  if(readnewcommand == 1) {  //read in the new command
    if(ping == 1) {
      //next time we really want the data
      ping = 0;
       //read status reg to clear
      read = SPISR;
      //read garbage
      read = SPIDR;
      //send for the real command from the chip
      SPIDR = 0xAA;
    }
    else {
```

```
ping = 1;
//read status reg to clear
read = SPISR;
//read garbage
read = SPIDR;
//this time we really want the data
switch(read) {
  case 0xFF:
    readnewcommand = 1;
    break;
  case 0x00:
    command = STOP_BOTH;
    newcommand = 1;
    readnewcommand = 0;
    break;
  case 0x02:
    command = ROTATE_CW_90;
    newcommand = 1;
    readnewcommand = 0;
    break;
  case 0x03:
    command = ROTATE_CW_45;
    newcommand = 1;
    readnewcommand = 0;
    break;
  case 0x04:
    command = ROTATE_CCW_90;
    newcommand = 1;
    readnewcommand = 0;
    break;
  case 0x05:
          command = ROTATE_CCW_45;
    newcommand = 1;
    readnewcommand = 0;
    break;
  case 0x08:
    command = FORWARD_HALF;
    newcommand = 1;
    readnewcommand = 0;
    break;
  case 0x09:
        command = FORWARD_FULL;
    newcommand = 1;
    readnewcommand = 0;
    break;
  case 0x10:
    command = REVERSE_HALF;
    newcommand = 1;
    readnewcommand = 0;
    break;
  case 0x11:
    command = REVERSE_FULL;
    newcommand = 1;
    readnewcommand = 0;
    break;
  case 0x20:
```

```
        command = ALIGN_WITH_BEACON;
        newcommand = 1;
        readnewcommand = 0;
        break;
      case 0x40:
        command = FORWARD_UNTIL_TAPE;
        newcommand = 1;
        readnewcommand = 0;
        break;
      default:
        readnewcommand = 1;
        break;
    }
   }
  }
}

void SPI_Debug(void){
 State_t cmd;
 //TIMER0_Init(TIMERS_RATE_1MS);
 SPI_Init();
 (void)printf("\nDebugging SPI Commands:\r\n");
       (void)printf("q: quit\r\n");
       // Break out of loop when user hits 'q'
 for(;;) {
           while (!kbhit()) {
    if (SPI_CheckNewCommand() == NEW_COMMAND) {
    (void)printf("New Command Event: ");
    cmd = SPI_GetCommand();
    switch(cmd){
      case STOP_BOTH:
        WHEELS_StopBoth();
        (void)printf("Stop.\n\r");
        break;
      case ROTATE_CW_90:
             WHEELS_RotateCW();
        (void)printf("CW 90deg.\n\r");
        break;
      case ROTATE_CW_45:
        WHEELS_RotateCW();
        (void)printf("CW 45deg.\n\r");
        break;
      case ROTATE_CCW_90:
        WHEELS_RotateCCW();
        (void)printf("CCW 90deg.\n\r");
        break;
      case ROTATE_CCW_45:
        WHEELS_RotateCCW();
        (void)printf("CCW 45deg.\n\r");
        break;
      case FORWARD_HALF:
        WHEELS_ForwardBothHalf();
                                    (void)printf("Forward half speed.\n\r");
        break;
      case FORWARD_FULL:
        WHEELS_ForwardBothFull();
```

```
        (void)printf("Forward full speed.\n\r");
        break;
      case REVERSE_HALF:
        WHEELS_ReverseBothHalf();
        (void)printf("Reverse half speed.\n\r");
        break;
      case REVERSE_FULL:
        WHEELS_ForwardBothFull();
        (void)printf("Reverse full speed.\n\r");
        break;
      case ALIGN_WITH_BEACON:
        //align();
        (void)printf("Align with beacon.\n\r");
        break;
      case FORWARD_UNTIL_TAPE:
        //gototape();
        (void)printf("Drive forward till tape is detected.\n\r");
        break;
      default:
        (void)printf("Error.\n\r");
        break;
      }
    }
        }
        // Clears input character and returns from the function.
        if (kbhit()) {
        // User selects options.
                char ch = getchar();
                switch (ch) {
                        case 'q':
                          WHEELS_StopBoth();
      (void)printf("QUIT\r\n");
                                return;
                        default:
                                //(void)printf("ERROR\r\n");
                                break;
                }
        }
      }
}
```

```
/*
 * Team Space
 * 2.2.10
 * Lab 8
 * State Module
 *
 * Andi Kleissner
 * Eli Michael
 * Elico Teixeira
 * Nina Joshi
 */

#ifndef _STATE_H_
#define _STATE_H_

typedef enum {
        STOP_BOTH,
        ROTATE_CW_90,
        ROTATE_CW_45,
        ROTATE_CCW_90,
        ROTATE_CCW_45,
        FORWARD_HALF,
        FORWARD_FULL,
        REVERSE_HALF,
        REVERSE_FULL,
        ALIGN_WITH_BEACON,
        FORWARD_UNTIL_TAPE
} State_t;

void STATE_InitStateMachine(void);
void STATE_RunStateMachine(Event_t event);

#endif
```

```
/*
 * Team Space
 * 2.3.10
 * Lab 8
 * State Module
 *
 * Andi Kleissner
 * Eli Michael
 * Elico Teixeira
 * Nina Joshi
 */


#include "defs.h"
#include "wheels.h"
#include "beacon.h"
#include "events.h"
#include "pwm.h"
#include "tape.h"
#include "timer.h"
#include "state.h"
#include "spi.h"

/* Variables */
static State_t state;

/* Prototypes */
static void StopBothState(Event_t event);
static void RotateCW90State(Event_t event);
static void RotateCW45State(Event_t event);
static void RotateCCW90State(Event_t event);
static void RotateCCW45State(Event_t event);
static void ForwardHalfState(Event_t event);
static void ForwardFullState(Event_t event);
static void ReverseHalfState(Event_t event);
static void ReverseFullState(Event_t event);
static void AlignWithBeaconState(Event_t event);
static void ForwardUntilTapeState(Event_t event);

/* Initialize state machine by setting initial state and final destination. */
void STATE_InitStateMachine() {
	state = STOP_BOTH;
}

/* Figure out which state we are in, and call the corresponding function to handle the event. */
void STATE_RunStateMachine(Event_t event) {
	switch (state) {
	    case STOP_BOTH:
			printf("Both stopped \r\n");
			StopBothState(event);
			break;
	    case ROTATE_CW_90:
			printf("Rotating CW 90 \r\n");
			RotateCW90State(event);
			break;
	    case ROTATE_CW_45:
```

```
                printf("Rotating CW 45 \r\n");
                RotateCW45State(event);
                break;
            case ROTATE_CCW_90:
                printf("Rotating CCW 90 \r\n");
                RotateCCW90State(event);
                break;
            case ROTATE_CCW_45:
                printf("Rotating CCW 45 \r\n");
                RotateCCW45State(event);
                break;
            case FORWARD_HALF:
                printf("Forward half \r\n");
                ForwardHalfState(event);
                break;
            case FORWARD_FULL:
                printf("Forward full \r\n");
                ForwardFullState(event);
                break;
            case REVERSE_HALF:
                printf("Reverse half \r\n");
                ReverseHalfState(event);
                break;
            case REVERSE_FULL:
                printf("Reverse full \r\n");
                ReverseFullState(event);
                break;
            case ALIGN_WITH_BEACON:
                printf("Aligning with beacon \r\n");
                AlignWithBeaconState(event);
                break;
            case FORWARD_UNTIL_TAPE:
                printf("Forward until tape \r\n");
                ForwardUntilTapeState(event);
                break;
            default:
                printf("ERROR: Undefined state\n");
                break;
        }
}

/* On entering function, stops both wheels until a new command is given. */
static void StopBothState(Event_t event) {
        static char entered  = 0;
        if (entered == 0) {
            WHEELS_StopBoth();
          entered = 1;
        }

        switch (event) {
            case NEW_COMMAND:
              entered = 0;
                state = SPI_GetCommand();
                break;
            default:
                break;
```

```
        }
}

/* On entering function, rotates clockwise 90 degrees and sets timer. If the timer expires,
 * will switch to stop state. Otherwise, will switch to next given command state. */
static void RotateCW90State(Event_t event) {
        static char entered  = 0;
        if (entered == 0) {
            WHEELS_RotateCW();
          TIMER0_InitTimer(ROTATE90_TIMER, ROTATE90_TIME); //set 90 timer
          entered = 1;
        }

        switch (event) {
          case ROTATE90_TIMER_EXPIRED:
            entered = 0;
            state = STOP_BOTH;
            break;
             case NEW_COMMAND:
               entered = 0;
                        state = SPI_GetCommand();
                        break;
             default:
                        break;
        }
}

/* On entering function, rotates clockwise 45 degrees and sets timer. If the timer expires,
 * will switch to stop state. Otherwise, will switch to next given command state. */
static void RotateCW45State(Event_t event) {
        static char entered  = 0;
        if (entered == 0) {
            WHEELS_RotateCW();
        TIMER0_InitTimer(ROTATE45_TIMER, ROTATE45_TIME); //set 45 timer
        entered = 1;
        }

        switch (event) {
            case ROTATE45_TIMER_EXPIRED:
              entered = 0;
            state = STOP_BOTH;
            break;
          case NEW_COMMAND:
            entered = 0;
                        state = SPI_GetCommand();
                        break;
             default:
                        break;
        }
}

/* On entering function, rotates counterclockwise 90 degrees and sets timer. If the timer
 *  expires, will switch to stop state. Otherwise, will switch to next given command state. */
static void RotateCCW90State(Event_t event) {
        static char entered  = 0;
        if (entered == 0) {
```

```
            WHEELS_RotateCCW();
        TIMER0_InitTimer(ROTATE90_TIMER, ROTATE90_TIME); //set 90 timer
        entered = 1;
        }

        switch (event) {
            case ROTATE90_TIMER_EXPIRED:
              entered = 0;
            state = STOP_BOTH;
            break;
          case NEW_COMMAND:
            entered = 0;
                        state = SPI_GetCommand();
                        break;
            default:
                        break;
        }
}

/* On entering function, rotates counterclockwise 45 degrees and sets timer. If the timer
 *  expires, will switch to stop state. Otherwise, will switch to next given command state. */
static void RotateCCW45State(Event_t event) {
        static char entered  = 0;
        if (entered == 0) {
            WHEELS_RotateCCW();
          TIMER0_InitTimer(ROTATE45_TIMER, ROTATE45_TIME); //set 45 timer
        entered = 1;
        }

        switch (event) {
            case ROTATE45_TIMER_EXPIRED:
              entered = 0;
            state = STOP_BOTH;
            break;
          case NEW_COMMAND:
            entered = 0;
                        state = SPI_GetCommand();
                        break;
            default:
                        break;
        }
}

/* On entering function, rotates both wheels forward at half duty until a new command is given. */
static void ForwardHalfState(Event_t event) {
        static char entered  = 0;
        if (entered == 0) {
            WHEELS_ForwardBothHalf();
          entered = 1;
        }

        switch (event) {
            case NEW_COMMAND:
              entered = 0;
                        state = SPI_GetCommand();
                        break;
```

```
            default:
                        break;
        }
}

/* On entering function, rotates both wheels forward at full duty until a new command is given. */
static void ForwardFullState(Event_t event) {
        static char entered  = 0;
        if (entered == 0) {
            WHEELS_ForwardBothFull();
          entered = 1;
        }

        switch (event) {
            case NEW_COMMAND:
              entered = 0;
                        state = SPI_GetCommand();
                        break;
            default:
                        break;
        }
}

/* On entering function, rotates both wheels reverse at half duty until a new command is given. */
static void ReverseHalfState(Event_t event) {
        static char entered  = 0;
        if (entered == 0) {
          WHEELS_ReverseBothHalf();
          entered = 1;
        }

        switch (event) {
            case NEW_COMMAND:
              entered = 0;
                        state = SPI_GetCommand();
                        break;
            default:
                        break;
        }
}

/* On entering function, rotates both wheels reverse at full duty until a new command is given. */
static void ReverseFullState(Event_t event) {
  static char entered  = 0;
        if (entered == 0) {
          WHEELS_ReverseBothFull();
          entered = 1;
        }

        switch (event) {
            case NEW_COMMAND:
              entered = 0;
                        state = SPI_GetCommand();
                        break;
            default:
                        break;
```

```
        }
}

/* On entering function, rotates clockwise. If beacon is seen, will switch to stop state.
 * Otherwise, will switch to next given command state. */
static void AlignWithBeaconState(Event_t event) {
  static char entered  = 0;
        if (entered == 0) {
          SETBIT(ALIGN_STATE_PORT, ALIGN_STATE_BIT);
          WHEELS_RotateCW();
          entered = 1;
        }

        switch (event) {
          case BEACON_ON:
            SETBIT(BEACON_ON_PORT, BEACON_ON_BIT);
            CLEARBIT(ALIGN_STATE_PORT, ALIGN_STATE_BIT);
            entered = 0;
            state = STOP_BOTH;
            break;
          case NEW_COMMAND:
            CLEARBIT(ALIGN_STATE_PORT, ALIGN_STATE_BIT);
            entered = 0;
            state = SPI_GetCommand();
            break;
             default:
                        break;
        }
}

/* On entering function, rotates both wheels forward. If tape is found, will switch
 * to stop state. Otherwise, will switch to next given command state. */
static void ForwardUntilTapeState(Event_t event) {
  static char entered  = 0;
        if (entered == 0) {
          SETBIT(TAPE_STATE_PORT, TAPE_STATE_BIT);
          WHEELS_ForwardBothFull();
          entered = 1;
        }

        switch (event) {
            case TAPE_FOUND:
              CLEARBIT(TAPE_STATE_PORT, TAPE_STATE_BIT);
              entered = 0;
              state = STOP_BOTH;
              break;
            case NEW_COMMAND:
                        CLEARBIT(TAPE_STATE_PORT, TAPE_STATE_BIT);
                        entered = 0;
                        state = SPI_GetCommand();
                        break;
            default:
                        break;
        }
}
```

```
/*
 * Team Space
 * 1.30.10
 * Lab 8
 * Tape Module
 *
 * Andi Kleissner
 * Eli Michael
 * Elico Teixeira
 * Nina Joshi
 */


#ifndef _TAPE_H_
#define _TAPE_H_

#include "events.h"

Event_t TAPE_CheckForTape(void);
void TAPE_Debug(void);

#endif
```

```
/*
 * Team Space
 * 2.2.10
 * Lab 8
 * Tape Module
 *
 * Andi Kleissner
 * Eli Michael
 * Elico Teixeira
 * Nina Joshi
 */


#include "defs.h"
#include "tape.h"

/* Variables */
static Event_t lastEvent = NO_EVENT;
static Event_t thisEvent = NO_EVENT;

/* Functions */
/* Returns a TAPE_FOUND event if the tape sensor signal greater than the specified threshold. */
 Event_t TAPE_CheckForTape() {
        Event_t event = NO_EVENT;

        if ((ADS12_ReadADPin(TAPE_BIT) > TAPE_THRESHOLD)) {
            event = TAPE_FOUND;
        }

        if (event != lastEvent) {
            lastEvent = event;
            return event;
        }

        return NO_EVENT;
}

/* Tape debug. */
void TAPE_Debug(void) {
        (void)printf("\nDebugging TAPE...Controls:\r\n");
        (void)printf("q: quit\r\n");
        // Break out of loop when user hits 'q'
        for(;;) {
            // Prints tape value until user quits.
            while (!kbhit()) {
                        Event_t event = TAPE_CheckForTape();
                        if (event == TAPE_FOUND) {
                                    (void)printf("Tape found\r\n");
                        } else if (event == NO_EVENT) {
                                    //(void)printf("%d\r\n", ADS12_ReadADPin(TAPE_BIT));
                        }
            }
            // Clears input character and returns from the function.
            if (getchar() == 'q') {
                        lastEvent = NO_EVENT;
                        return;
```

```
            }
        }
    }
```

```
/*
 * Team Space
 * 2.2.10
 * Lab 8
 * Timer Module
 *
 * Andi Kleissner
 * Eli Michael
 * Elico Teixeira
 * Nina Joshi
 */

#ifndef _TIMER_H
#define _TIMER_H

/****************************************Type
Definitions/Enumerations****************************************/
// typedef enums for rate setting:
typedef enum {
        TIMERS_RATE_1MS  = 1, /* overflow for timer0 is 1.09 min */
        TIMERS_RATE_2MS  = 2, /* overflow is 2.18 min */
        TIMERS_RATE_4MS  = 4, /* overflow is 4.36 min */
        TIMERS_RATE_8MS  = 8, /* overflow is 8.73 min */
        TIMERS_RATE_16MS = 16, /* overflow is 17.47 min */
        TIMERS_RATE_32MS = 32  /* overflow is 34.9 mins */
} TimerRate_t;
//typedef enums to keep track of flags being set
typedef enum {
        SET,
        CLEARED,
        ACTIVE,
        NOTACTIVE
} Flags_t;
//typedef enum to return the state of whether a timer has expired or not
typedef enum {
        TIMER0_ERR,
        TIMER0_NOT_EXPIRED,
        TIMER0_EXPIRED
} TimerReturn_t;



/****************************************Function Prototypes and
Details****************************************/

void TIMER0_Init(unsigned char NewRate);
/* Turns timer system on and sets the timer to count at a certain
 rate using the TIMER_RATE_XX definitions. Enables interrupts.  */


void TIMER0_InitTimer(unsigned char Num, unsigned int NewTime);
/* This function will take in two paramaters, the first a char Num which
 will choose which timer we are starting/restarting, and the second
 an unsigned int NewTime which will give the number of ticks until that
 timer is expired. Everytime this function is called, that timer will start
 counting up to the the number of NewTime ticks that it is asked for. The
```

```
 user must be sure that this NewTime of ticks does not pass the overflow
 otherwise this will be inaccurate. */


TimerReturn_t TIMER0_IsTimerExpired(unsigned char Num);
/* Checks to see if the the timer associated with the parameter input Num
 has expired (which clock, 0-7), the number of the timer to test. This
 function will return TIMER0_EXPIRED or TIMER0_NOT_EXPIRED, or TIMER0_ERR */

void TIMER0_ClearTimerExpired(unsigned char Num);
/* This takes as a paramater an unsigned char Num which chooses which timer
 flag that we want to clear. This can be used to show that an event
 has been serviced. */

unsigned int TIMER0_GetTime(void);
/* This function takes in no parameters, and will return an unsigned int
 representing the current clock count which will be between 0 and
 65535. It simply returns the free-running counter of timer0. */

#endif
```

```
/*
 * Team Space
 * 2.2.10
 * Lab 8
 * Timer Module
 *
 * Andi Kleissner
 * Eli Michael
 * Elico Teixeira
 * Nina Joshi
 */


#include <hidef.h>          /* common defines and macros */
#include <mc9s12e128.h>     /* derivative information */
#include <stdio.h>          /* standard library */
#include <Bin_Const.h>      /* Carryer constants */
#include "S12eVec.h"        /* Interrupts for E128 */
#include <S12E128bits.h> // E128 bit definitions
#include <bitdefs.h> // BIT0HI, BIT0LO....definitions
#include "ME218_E128.h"
#include "timer.h"    /* This header include file */

//E128 Specific
#pragma LINK_INFO DERIVATIVE "SampleS12"


// Module Level Variables
static unsigned int timer0;   // used to keep track of Timer overflows by using the Input Capture
static unsigned int Period; //based on the 43.69mS overflow, or 1.5 MHz clock
static Flags_t flag0_0 = CLEARED;                       // flags used to keep track of whether the time has
expired for each of the four timers
static Flags_t flag0_1 = CLEARED;
static Flags_t flag0_2 = CLEARED;
static Flags_t flag0_3 = CLEARED;
static Flags_t flag0_4 = CLEARED;
static Flags_t flag0_5 = CLEARED;
static Flags_t flag0_6 = CLEARED;
static Flags_t flag0_7 = CLEARED;
static Flags_t flagact0_0 = NOTACTIVE;                  // flags used to keep track of whether the time has
expired for each of the four timers
static Flags_t flagact0_1 = NOTACTIVE;
static Flags_t flagact0_2 = NOTACTIVE;
static Flags_t flagact0_3 = NOTACTIVE;
static Flags_t flagact0_4 = NOTACTIVE;
static Flags_t flagact0_5 = NOTACTIVE;
static Flags_t flagact0_6 = NOTACTIVE;
static Flags_t flagact0_7 = NOTACTIVE;
static unsigned int NewTime0_0;                 // the new time for each of the four timers that will be used to
see if we have time expired
static unsigned int NewTime0_1;
static unsigned int NewTime0_2;
static unsigned int NewTime0_3;
static unsigned int NewTime0_4;
static unsigned int NewTime0_5;
static unsigned int NewTime0_6;
```

```
static unsigned int NewTime0_7;


/*-------------------------------------------------------------------------*/

/* TIMER0_Init()
    Turns timer system on and sets the timer to count at a certain
    rate using the TIMER_RATE_XX definitions.
                                                        */
void TIMER0_Init(unsigned char NewRate){

        TIM0_TSCR1 = _S12_TEN;              //Turn the timer system on
        TIM0_TSCR2 = TIM0_TSCR2 = ((TIM0_TSCR2 & ~_S12_PR0) & ~_S12_PR1) | _S12_PR2;
//100, divide by 16, so timer overflow occurs every 43.69mS

        switch( NewRate )                   //Depending on the input from NewRate, set the timer Period
        {                                                       //which will be used to trigger the output
compare at the correct times
            case 1:
                        Period = 1500;
                        break;
            case 2:
                        Period = 3000;
                        break;
            case 4:
                        Period = 6000;
                        break;
            case 8:
                        Period = 12000;
                        break;
            case 16:
                        Period = 24000;
                        break;
            case 32:
                        Period = 48000;
                        break;
        }

        //Setup OC4 to time the updates, and to trigger the first interrupt that will keep track of these times
        TIM0_TIOS = TIM0_TIOS | _S12_IOS4;    //Set IOC4 of the timer to be an output compare, the rest
remain as inputs
        TIM0_TCTL1 = TIM0_TCTL1 & ~(_S12_OL4 | _S12_OM4);   //No pin connected to IOC4, which
means pin PT0 remains free
        TIM0_TC4 = TIM0_TCNT + Period;    //Set the first output capture to happen one "Period" into the
future
        TIM0_TFLG1 = _S12_C4F;  //Clear the flag for the IOC4
        TIM0_TIE = TIM0_TIE|_S12_C4F;   //enable the interrupt for IOC4
        EnableInterrupts;
}



/*-------------------------------------------------------------------------*/
/*-------------------------------------------------------------------------*/
/***** Interrupt Response for Timer IOC4 *****/
/* interrupt _Vec_tim0ch4 TimerCounter
```

```
    This interrupt will be keeping track for timer0 channel 4, and will
        be setting flags for timer0 on channel 4-7 which will allow
        the TIMER0_InitTimer(TimerNumber, TicksToCount) function
    to let the user know if the timer is expired
                                                */

void interrupt _Vec_tim0ch4 Timer0Counter (void){
        TIM0_TFLG1 = _S12_C4F;  //Clear the flag for the IOC4
        TIM0_TC4 += Period;                             //Update for the next interrupt to keep track of the
ticking rate
        EnableInterrupts;
        timer0++;                                   //Add one to timer0 to indicate that one clock tick has passed
by
        /* Now we check to see if any of the flags have expired and update */
        if (timer0==NewTime0_0 && flagact0_0 == ACTIVE){  /********* NewTime0_1 will have to be
UpdateTime+timer0 in the InitTimer function *******/
            flag0_0 = SET;
            flagact0_0 = NOTACTIVE;
        }
        if (timer0==NewTime0_1 && flagact0_1 == ACTIVE){
            flag0_1 = SET;
            flagact0_1 = NOTACTIVE;
        }
        if (timer0==NewTime0_2 && flagact0_2 == ACTIVE){
            flag0_2 = SET;
            flagact0_2 = NOTACTIVE;
        }
        if (timer0==NewTime0_3 && flagact0_3 == ACTIVE){
            flag0_3 = SET;
            flagact0_3 = NOTACTIVE;
        }
        if (timer0==NewTime0_4 && flagact0_4 == ACTIVE){
            flag0_4 = SET;
            flagact0_4 = NOTACTIVE;
        }
        if (timer0==NewTime0_5 && flagact0_5 == ACTIVE){
            flag0_5 = SET;
            flagact0_5 = NOTACTIVE;
        }
        if (timer0==NewTime0_6 && flagact0_6 == ACTIVE){
            flag0_6 = SET;
            flagact0_6 = NOTACTIVE;
        }
        if (timer0==NewTime0_7 && flagact0_7 == ACTIVE){
            flag0_7 = SET;
            flagact0_7 = NOTACTIVE;
        }
}

/*-------------------------------------------------------------------------*/
/*-------------------------------------------------------------------------*/

/* TIMER0_InitTimer
    This function will take in two paramaters, the first a char Num which
        will choose which timer we are starting/restarting, and the second
        an unsigned int NewTime which will give the number of ticks until that
```

```
        timer is expired. Everytime this function is called, that timer will start
        counting up to the the number of NewTime ticks that it is asked for. The
        user must be sure that this NewTime of ticks does not pass the overflow
        otherwise this will be inaccurate.
                                                */
void TIMER0_InitTimer(unsigned char Num, unsigned int NewTime){
      switch( Num )                         //Choose which timer that the user wants by the timer number
and
      {                                                 //be sure to set the NewTime for that timer as well as
to clear its flag
          case 0:
                NewTime0_0 = timer0+NewTime;
                flag0_0 = CLEARED;
                flagact0_0 = ACTIVE;
                break;
          case 1:
                NewTime0_1 = timer0+NewTime;
                flag0_1 = CLEARED;
                flagact0_1 = ACTIVE;
                break;
          case 2:
                NewTime0_2 = timer0+NewTime;
                flag0_2 = CLEARED;
                flagact0_2 = ACTIVE;
                break;
          case 3:
                NewTime0_3 = timer0+NewTime;
                flag0_3 = CLEARED;
                flagact0_3 = ACTIVE;
                break;
          case 4:
                NewTime0_4 = timer0+NewTime;
                flag0_4 = CLEARED;
                flagact0_4 = ACTIVE;
                break;
          case 5:
                NewTime0_5 = timer0+NewTime;
                flag0_5 = CLEARED;
                flagact0_5 = ACTIVE;
                break;
          case 6:
                NewTime0_6 = timer0+NewTime;
                flag0_6 = CLEARED;
                flagact0_6 = ACTIVE;
                break;
          case 7:
                NewTime0_7 = timer0+NewTime;
                flag0_7 = CLEARED;
                flagact0_7 = ACTIVE;
                break;
      }
}


/*---------------------------------------------------------------------------*/
/*---------------------------------------------------------------------------*/
```

```
/* TIMER0_IsTimerExpired
    Checks to see if the the timer associated with the parameter input Num
        has expired (which clock, 0-7), the number of the timer to test. This
        function will return TIMER0_EXPIRED or TIMER0_NOT_EXPIRED, or TIMER0_ERR
                                                */

TimerReturn_t TIMER0_IsTimerExpired(unsigned char Num){
      switch( Num )                         //Choose which timer that the user wants to check and then
return
      {                                                 //on the basis of whether it has expired or not
         case 0:
                  if (flag0_0 == SET){
                           return TIMER0_EXPIRED;
                  }else if (flag0_0 == CLEARED) {
                           return TIMER0_NOT_EXPIRED;
                  }else {
                           return TIMER0_ERR;
                  }
                  break;
         case 1:
                  if (flag0_1 == SET){
                           return TIMER0_EXPIRED;
                  }else if (flag0_1 == CLEARED) {
                           return TIMER0_NOT_EXPIRED;
                  }else {
                           return TIMER0_ERR;
                  }
                  break;
         case 2:
                  if (flag0_2 == SET){
                           return TIMER0_EXPIRED;
                  }else if (flag0_2 == CLEARED) {
                           return TIMER0_NOT_EXPIRED;
                  }else {
                           return TIMER0_ERR;
                  }
                  break;
         case 3:
                  if (flag0_3 == SET){
                           return TIMER0_EXPIRED;
                  }else if (flag0_3 == CLEARED) {
                           return TIMER0_NOT_EXPIRED;
                  }else {
                           return TIMER0_ERR;
                  }
                  break;
         case 4:
                  if (flag0_4 == SET){
                           return TIMER0_EXPIRED;
                  }else if (flag0_4 == CLEARED) {
                           return TIMER0_NOT_EXPIRED;
                  }else {
                           return TIMER0_ERR;
                  }
                  break;
```

```
        case 5:
                if (flag0_5 == SET){
                        return TIMER0_EXPIRED;
                }else if (flag0_5 == CLEARED) {
                        return TIMER0_NOT_EXPIRED;
                }else {
                        return TIMER0_ERR;
                }
                break;
        case 6:
                if (flag0_6 == SET){
                        return TIMER0_EXPIRED;
                }else if (flag0_6 == CLEARED) {
                        return TIMER0_NOT_EXPIRED;
                }else {
                        return TIMER0_ERR;
                }
                break;
        case 7:
                if (flag0_7 == SET){
                        return TIMER0_EXPIRED;
                }else if (flag0_7 == CLEARED) {
                        return TIMER0_NOT_EXPIRED;
                }else {
                        return TIMER0_ERR;
                }
                break;
    }
}



/*---------------------------------------------------------------------------*/
/*---------------------------------------------------------------------------*/

/* TIMER0_ClearTimerExpired
    This takes as a paramater an unsigned char Num which chooses which timer
        flag that we want to clear. This can be used to show that an event
        has been serviced.
                                                */

void TIMER0_ClearTimerExpired(unsigned char Num){

    switch( Num )                       //Choose which timer that the user wants to clear and clear the
according flag
    {
        case 0:
                flag0_0 = CLEARED;
                break;
        case 1:
                flag0_1 = CLEARED;
                break;
        case 2:
                flag0_2 = CLEARED;
                break;
        case 3:
```

```
                flag0_3 = CLEARED;
                break;
        case 4:
                flag0_4 = CLEARED;
                break;
        case 5:
                flag0_5 = CLEARED;
                break;
        case 6:
                flag0_6 = CLEARED;
                break;
        case 7:
                flag0_7 = CLEARED;
                break;
    }
}


/*-------------------------------------------------------------------------*/
/*-------------------------------------------------------------------------*/

/* TIMER0_GetTime
    This function takes in no parameters, and will return an unsigned int
        representing the current clock count which will be between 0 and
        65535. It simply returns the free-running counter of timer0.
                                                        */


unsigned int TIMER0_GetTime(void){
    return timer0;
}


/*-------------------------------------------------------------------------*/
/*-------------------------------------------------------------------------*/

/*
 // Testing Harness


void main(void) {

    TIMER0_Init(TIMERS_RATE_32MS);

    TIMER0_InitTimer(2, 156);
    (void) printf("Start timing now.../n/r");
    while (1) {
      if (TIMER0_IsTimerExpired(2) == TIMER0_EXPIRED) {
          (void) printf("Time has expired!");
        TIMER0_ClearTimerExpired(2);
      }
    }
}

*/
/*-------------------------------------------------------------------------*/
```

```
/*
 * Team Space
 * 1.30.10
 * Lab 8
 * Wheels Module
 *
 * Andi Kleissner
 * Eli Michael
 * Elico Teixeira
 * Nina Joshi
 */


#ifndef _WHEELS_H_
#define _WHEELS_H_

void WHEELS_StopBoth(void); //0x00
void WHEELS_RotateCW(void); //0x02 and 0x03 - adjust timers for degree
void WHEELS_RotateCCW(void);//0x04 and 0x05 - adjust timers for degree
void WHEELS_ForwardBothHalf(void); //0x08
void WHEELS_ForwardBothFull(void); //0x09
void WHEELS_ReverseBothHalf(void); //0x10
void WHEELS_ReverseBothFull(void); //0x11
void WHEELS_Debug(void);

#endif
```

```
/*
 * Team Space
 * 2.3.10
 * Lab 8
 * Wheels Module
 *
 * Andi Kleissner
 * Eli Michael
 * Elico Teixeira
 * Nina Joshi
 */


#include "defs.h"
#include "wheels.h"
#include "pwm.h"

/* Functions */
/* Stop both wheels */
void WHEELS_StopBoth(void) { //0x00
	PWM_SetDuty(LEFT, 0, FORWARD);
	PWM_SetDuty(RIGHT, 0, FORWARD);
}

/* Rotate wheels clockwise */
void WHEELS_RotateCW(void) { //0x02 and 0x03 - adjust timers for degree
	PWM_SetDuty(LEFT, FULL, FORWARD);
	PWM_SetDuty(RIGHT, FULL, REVERSE);
}

/* Rotate wheels counterclockwise */
void WHEELS_RotateCCW(void) { //0x04 and 0x05 - adjust timers for degree
	PWM_SetDuty(LEFT, FULL, REVERSE);
	PWM_SetDuty(RIGHT, FULL, FORWARD);
}

/* Wheels forward at half duty cycle */
void WHEELS_ForwardBothHalf(void) { //0x08
	PWM_SetDuty(LEFT, HALF, FORWARD);
	PWM_SetDuty(RIGHT, HALF, FORWARD);
}

/* Wheels forward at full duty cycle */
void WHEELS_ForwardBothFull(void) { //0x09
	PWM_SetDuty(LEFT, FULL, FORWARD);
	PWM_SetDuty(RIGHT, FULL, FORWARD);
}

/* Wheels reverse at half duty cycle */
void WHEELS_ReverseBothHalf(void) { //0x10
	PWM_SetDuty(LEFT, HALF, REVERSE);
	PWM_SetDuty(RIGHT, HALF, REVERSE);
}

/* Wheels reverse at full duty cycle */
void WHEELS_ReverseBothFull(void) { //0x11
```

```
	PWM_SetDuty(LEFT, FULL, REVERSE);
	PWM_SetDuty(RIGHT, FULL, REVERSE);
}

/* WHEELS debug. */
void WHEELS_Debug(void) {
	(void)printf("\nDebugging WHEELS...Controls:\r\n");
	(void)printf("0: 0x00 stopped, 2: 0x02 rotate cw, 4: 0x04 rotate ccw, 8: 0x08 fwd half\r\n");
	(void)printf("9: 0x09 fwd full, t: 0x10 rev half, e: 0x11 rev full, q: quit\r\n");
	// Break out of loop when user hits 'q'
	for(;;) {
	    // Clears input character and returns from the function.
	    if (kbhit()) {
			// User selects options.
			char ch = (char) getchar();
			switch (ch) {
				case '0':
					WHEELS_StopBoth();
					(void)printf("0x00 stopped\r\n");
					break;
				case '2':
					WHEELS_RotateCW();
					(void)printf("0x02 or 0x03 rotate CW\r\n");
					break;
				case '4':
					WHEELS_RotateCCW();
					(void)printf("0x04 or 0x05 rotate CCW\r\n");
					break;
				case '8':
					WHEELS_ForwardBothHalf();
					(void)printf("0x08 forward half\r\n");
					break;
				case '9':
					WHEELS_ForwardBothFull();
					(void)printf("0x09 forward full\r\n");
					break;
				case 't':
					WHEELS_ReverseBothHalf();
					(void)printf("0x10 reverse half\r\n");
					break;
				case 'e':
					WHEELS_ReverseBothFull();
					(void)printf("0x11 reverse full\r\n");
					break;
				case 'q':
					WHEELS_StopBoth();
					(void)printf("QUIT\r\n");
					return;
				default:
					//(void)printf("ERROR\r\n");
					break;
			}
	    }
```