```
/*******************************************************************************
 Module
   SensorService.c

 Revision
   2.0.1

 Description
   This is a template file for implementing state machines.

 Notes

 History
 When           Who     What/Why
 -------------- ---     --------
 02/07/13 21:00 jec      corrections to return variable (should have been
                         ReturnEvent, not CurrentEvent) and several EV_xxx
                         event names that were left over from the old version
 02/08/12 09:56 jec      revisions for the Events and Services Framework Gen2
 02/13/10 14:29 jec      revised Start and run to add new kind of entry
function
                         to make implementing history entry cleaner
 02/13/10 12:29 jec      added NewEvent local variable to During function and
                         comments about using either it or Event as the
return
 02/11/10 15:54 jec      more revised comments, removing last comment in
during
                         function that belongs in the run function
 02/09/10 17:21 jec      updated comments about internal transitions on
During funtion
 02/18/09 10:14 jec      removed redundant call to RunLowerlevelSM in
EV_Entry
                         processing in During function
 02/20/07 21:37 jec      converted to use enumerated type for events & states
 02/13/05 19:38 jec      added support for self-transitions, reworked
                         to eliminate repeated transition code
 02/11/05 16:54 jec      converted to implment hierarchy explicitly
 02/25/03 10:32 jec      converted to take a passed event parameter
 02/18/99 10:19 jec      built template from MasterMachine.c
 02/14/99 10:34 jec      Began Coding
*******************************************************************************/
/*----------------------------- Include Files -----------------------------*/
// Basic includes for a program using the Events and Services Framework
#include "ES_Configure.h"
#include "ES_Framework.h"

/* include header files for this state machine as well as any machines at the
   next lower level in the hierarchy that are sub-machines to this machine
*/
#include "SensorService.h"

/*----------------------------- Module Defines ----------------------------*/
// define constants for the states for this machine
// and any other local defines
static uint8_t MyPriority;
```

```
static short last_pot_input;

static unsigned char pot_value;

static unsigned char Button_State = 0;

static unsigned char last_dipswitch_value;

static unsigned char last_goal_read = 0;
static unsigned char last_knight_fwd_read = 0;
static unsigned char last_knight_rvs_read = 0;

static unsigned char goal_flag = 0; //has the IR signal corresponding to the
goal been seen within a specified timeout period?
static unsigned char knight_fwd_flag = 0;
static unsigned char knight_rvs_flag = 0;
static uint16_t goal_check_index = 0; //number of consecutive polling cycles
in which the IR signal corresponding to the goal has been read low
static uint16_t knight_fwd_check_index = 0;
static uint16_t knight_rvs_check_index = 0;


static unsigned char last_tape_left = 0;
static unsigned char last_tape_right = 0;
static unsigned char last_tape_center = 0;
static unsigned char last_home_fwd = 0;
static unsigned char last_home_rvs = 0;
static unsigned char last_bumper_fwd = 0;
static unsigned char last_bumper_rvs = 0;

static unsigned char home_fwd_count = 0;
static unsigned char home_rvs_count = 0;

static uint16_t last_sonic_edge_fwd;
static uint16_t last_sonic_edge_rvs;
static uint16_t last_sonic_edge_side_fwd;
static uint16_t last_sonic_edge_side_rvs;
static uint16_t pulse_width_fwd_ticks;
static uint16_t pulse_width_rvs_ticks;
static uint16_t pulse_width_side_fwd_ticks;
static uint16_t pulse_width_side_rvs_ticks;
static uint16_t last_pulse_width_fwd_ticks = 65531;
static uint16_t last_pulse_width_rvs_ticks = 65531;
static uint16_t last_pulse_width_side_fwd_ticks = 65531;
static uint16_t last_pulse_width_side_rvs_ticks = 65531;

static unsigned char last_ultrasonic_trigger = 0;


/*---------------------------- Module Functions ----------------------------*/
/* prototypes for private functions for this machine, things like during
   functions, entry & exit functions.They should be functions relevant to the
   behavior of this state machine
*/
//static ES_Event DuringOut_of_Balls( ES_Event Event);

/*---------------------------- Module Variables ----------------------------*/
```

```
// everybody needs a state variable, you may need others as well
static TemplateState_t CurrentState;

/*------------------------------ Module Code ------------------------------*/

/****************************************************************************
 Function
     PostServoService

 Parameters
     EF_Event ThisEvent ,the event to post to the queue

 Returns
     boolean False if the Enqueue operation failed, True otherwise

 Description
     Posts an event to this state machine's queue
 Notes

 Author
     J. Edward Carryer, 10/23/11, 19:25
****************************************************************************/
bool PostSensorService( ES_Event ThisEvent )
{
  return ES_PostToService( MyPriority, ThisEvent);
}




/****************************************************************************
 Function
    RunSensorService

 Parameters
   ES_Event: the event to process

 Returns
   ES_Event: an event to return

 Description
   add your description here
 Notes
   uses nested switch/case to implement the machine.
 Author
   J. Edward Carryer, 2/11/05, 10:45AM
****************************************************************************/
ES_Event RunSensorService( ES_Event CurrentEvent )
{
   //unsigned char MakeTransition = false;/* are we making a state
transition? */
   TemplateState_t NextState = CurrentState;
   ES_Event EntryEventKind = { ES_ENTRY, 0 };// default to normal entry to
new state
   ES_Event ReturnEvent = CurrentEvent; // assume we are not consuming event
   ES_Event PostEvent;
```

```
    short current_pot_input;

    unsigned char current_dipswitch_value;

    unsigned char Button_Value;

    unsigned char current_goal_read;
    unsigned char current_knight_fwd_read;
    unsigned char current_knight_rvs_read;

unsigned char current_tape_left;
unsigned char current_tape_right;
unsigned char current_tape_center;
unsigned char current_home_fwd;
unsigned char current_home_rvs;
unsigned char current_bumper_fwd;
unsigned char current_bumper_rvs;

static unsigned char Course_Correct_Flag = 0;
static unsigned char soccering_flag = 0;
static unsigned char home2_flag = 0;
static unsigned char home_fwd_flag = 0;
static unsigned char home_rvs_flag = 0;
static unsigned char slow_down_fwd_flag = 0;
static unsigned char slow_down_rvs_flag = 0;

static unsigned char toggle_command_switch = 1;

uint16_t current_sonic_side_fwd_distance;
uint16_t current_sonic_side_rvs_distance;
uint16_t current_sonic_fwd_distance;
uint16_t current_sonic_rvs_distance;
uint16_t average_sonic_side_distance;

  #ifdef PRINT_SENSOR_CALLS

    printf("RunSensorService\n\r");

  #endif

        //Read and process pots, switches, and buttons on JP5 board (for
calibration/test purposes)
        current_pot_input = ADS12_ReadADPin(POT_PIN);

        current_dipswitch_value = get_dipswitch();

        Button_Value = ((PTIAD & PUSHBUTTON) == PUSHBUTTON);

        #ifdef PRINT_SENSOR_READS

                //printf("Current Pot Input: %d\n\r", get_cal_pot());
                //printf("Dipswitch Value: %d\n\r", get_dipswitch());
                //printf("Dipswitch 1: %d\n\r", get_dipswitch_1());
                //printf("Dipswitch 2: %d\n\r", get_dipswitch_2());
                //printf("Dipswitch 3: %d\n\r", get_dipswitch_3());
                //printf("Pushbutton: %d\n\r", ((PTIAD & PUSHBUTTON) ==
PUSHBUTTON));
```

```
                //printf("Forward Ultrasonic: %d\n\r", get_sonic_fwd());
                //printf("Back Ultrasonic: %d\n\r", get_sonic_back());
                //printf("Side FWD Ultrasonic: %d\n\r", get_sonic_side_fwd());
                //printf("Side RVS Ultrasonic: %d\n\r",
get_sonic_side_back());

        #endif

        if(abs(current_pot_input - last_pot_input) > 10)
        {
                pot_value = (unsigned
char)(((float)(current_pot_input))*100/1024);

                PostEvent.EventType = New_Pot;

                PostEvent.EventParam = pot_value;

                PostMasterHSM(PostEvent);

                        #ifdef MANUAL_COMMANDS

                                                if( get_cal_pot() > 90 )
                                                {

        PostEvent.EventType = pas_d_armes;


        PostMasterHSM(PostEvent);



                                                }
                                                else if( get_cal_pot() < 10)
                                                {
                                                        PostEvent.EventType =
recess;


        PostMasterHSM(PostEvent);



                                                }

                #endif

        }

        if(last_dipswitch_value != current_dipswitch_value)
        {
                PostEvent.EventType = New_Dipswitch;

                PostEvent.EventParam = current_dipswitch_value;

                PostMasterHSM(PostEvent);

        /*      #ifdef MANUAL_COMMANDS
```

```
                                                if( (get_dipswitch() ==
1)|(get_dipswitch() == 3)|(get_dipswitch() == 5)|(get_dipswitch() == 7) )
                                                {

        PostEvent.EventType = pas_d_armes;


        PostMasterHSM(PostEvent);


        //toggle_command_switch = 0;

                                                }
                                                else if( (get_dipswitch() ==
0)|(get_dipswitch() == 2)|(get_dipswitch() == 4)|(get_dipswitch() == 6) )
                                                {
                                                        PostEvent.EventType =
recess;


        PostMasterHSM(PostEvent);

                                                        //toggle_command_switch
= 1;

                                                }

                #endif                  */


        }

        if(Button_Value == 0)
        {
            Button_State = 0;
        }
        else
        {
                switch(Button_State)
                {
                    case 0:
                    {
                        Button_State = 1;
                     }
                     break;

                     case 1:
                     {
                        Button_State = 2;

                        PostEvent.EventType = Button_Pushed;

                                        PostMasterHSM(PostEvent);

                                        #ifdef MANUAL_COMMANDS
```

```
                                                if(toggle_command_switch == 1)
                                                {
        PostEvent.EventType = pas_d_armes;

        PostMasterHSM(PostEvent);

        toggle_command_switch = 0;
                                                }
                                                else
                                                {
                                                        PostEvent.EventType =
recess;

        PostMasterHSM(PostEvent);
                                                        toggle_command_switch =
1;
                                                }
                                        #endif

                    }
                    break;

                    case 2:
                    {
                    //do nothing now that button is debounced
                    }
                    break;
                }
        }

        last_pot_input = current_pot_input;

        last_dipswitch_value = current_dipswitch_value;


        //poll IR sensors for goal and knight beacons
        current_goal_read = goal_flag;
        current_knight_fwd_read = knight_fwd_flag;
        current_knight_rvs_read = knight_rvs_flag;

        //post event if status of goal beacon IR sensor has changed since last
polling
        if (current_goal_read != last_goal_read)
        {
                if(current_goal_read == 1)
                {
                        PostEvent.EventType = found_goal;

                        PostMasterHSM(PostEvent);
```

```
                }
                else
                {
                        PostEvent.EventType = lost_goal;

                        PostMasterHSM(PostEvent);
                }
        }

        last_goal_read = current_goal_read ; //update last_goal_read


     //post event if status of front knight beacon IR sensor has changed
since last polling
        if (current_knight_fwd_read != last_knight_fwd_read)
        {
                if(current_knight_fwd_read == 1)
                {
                        PostEvent.EventType = found_knight_fwd;

                        PostMasterHSM(PostEvent);

                #ifdef PRINT_SENSOR_READS

                printf("found_knight_fwd\n\r");

                #endif

                }
                else
                {
                        PostEvent.EventType = lost_knight_fwd;

                        PostMasterHSM(PostEvent);

                        #ifdef PRINT_SENSOR_READS

                                printf("lost_knight_fwd\n\r");

                        #endif
                }
        }

        last_knight_fwd_read = current_knight_fwd_read ; //update
last_knight_fwd_read


        //post event if status of front knight beacon IR sensor has changed
since last polling
        if (current_knight_rvs_read != last_knight_rvs_read)
        {
                if(current_knight_rvs_read == 1)
                {
                        PostEvent.EventType = found_knight_rvs;

                        PostMasterHSM(PostEvent);
```

```
                        #ifdef PRINT_SENSOR_READS

                                printf("found_knight_rvs\n\r");

                        #endif

                }
                else
                {
                        PostEvent.EventType = lost_knight_rvs;

                        PostMasterHSM(PostEvent);

                        #ifdef PRINT_SENSOR_READS

                                printf("lost_knight_rvs\n\r");

                        #endif

                }
        }

        last_knight_rvs_read = current_knight_rvs_read ; //update
last_knight_fwd_read


//update other sensor reads:

current_tape_left = get_tape_left();
current_tape_right = get_tape_right();
current_tape_center = get_tape_center();
current_home_fwd = get_home_fwd();
current_home_rvs = get_home_rvs();
current_bumper_fwd = get_bumper_fwd();
current_bumper_rvs = get_bumper_rvs();
current_sonic_side_fwd_distance = get_sonic_side_fwd();
current_sonic_side_rvs_distance = get_sonic_side_back();
current_sonic_fwd_distance = get_sonic_fwd();
current_sonic_rvs_distance = get_sonic_back();


//update stored value of other sensor reads:
        /*if (current_tape_left != last_tape_left)
        {
                if(current_tape_left == 1)
                {
                        PostEvent.EventType = found_tape_left;

                        PostMasterHSM(PostEvent);
                }
                else
                {
                        PostEvent.EventType = lost_tape_left;

                        PostMasterHSM(PostEvent);
                }
        }
```

```
        last_tape_left = current_tape_left ;


        if (current_tape_right != last_tape_right)
        {
                if(current_tape_right == 1)
                {
                        PostEvent.EventType = found_tape_right;

                        PostMasterHSM(PostEvent);
                }
                else
                {
                        PostEvent.EventType = lost_tape_right;

                        PostMasterHSM(PostEvent);
                }
        }

        last_tape_right = current_tape_right ;


        if (current_tape_center != last_tape_center)
        {
                if(current_tape_center == 1)
                {
                        PostEvent.EventType = found_tape_center;

                        PostMasterHSM(PostEvent);
                }
                else
                {
                        PostEvent.EventType = lost_tape_center;

                        PostMasterHSM(PostEvent);
                }
        }

        last_tape_center = current_tape_center ;*/


        average_sonic_side_distance = (current_sonic_side_fwd_distance +
current_sonic_side_rvs_distance)/2;

/*      if( (average_sonic_side_distance > WALL_OFFSET_MAX) &&
(Course_Correct_Flag == 0))
        {
                PostEvent.EventType = Crossed_Offset_Max;

                PostMasterHSM(PostEvent);

                Course_Correct_Flag = 1; //Set flag to indicate correction
        }
        else if ((average_sonic_side_distance < WALL_OFFSET_MIN) &&
(Course_Correct_Flag == 0) )
        {
```

```
                PostEvent.EventType = Crossed_Offset_Min;

                PostMasterHSM(PostEvent);

                Course_Correct_Flag = 1; //Set flag to indicate correction

        }
        else if( (average_sonic_side_distance <= (WALL_OFFSET_NOMINAL +
WALL_OFFSET_HYSTERESIS))&&
        (average_sonic_side_distance >= (WALL_OFFSET_NOMINAL -
WALL_OFFSET_HYSTERESIS))&&
        (Course_Correct_Flag == 1) )
        {
                Course_Correct_Flag = 0;

                PostEvent.EventType = Crossed_Offset_Nominal;

                PostMasterHSM(PostEvent);

        }*/



    if ( (current_sonic_fwd_distance < SLOW_DOWN_FWD_OFFSET) &&
(slow_down_fwd_flag == 0) )
        {
                PostEvent.EventType = Crossed_Slow_Down_Fwd_Offset;

                PostMasterHSM(PostEvent);

                slow_down_fwd_flag = 1;

        }
        else if ( (current_sonic_fwd_distance > SLOW_DOWN_FWD_OFFSET) &&
(slow_down_fwd_flag == 1) )
        {
                slow_down_fwd_flag = 0;

        }

        if ( (current_sonic_rvs_distance < SLOW_DOWN_RVS_OFFSET) &&
(slow_down_rvs_flag == 0) )
        {
                PostEvent.EventType = Crossed_Slow_Down_Rvs_Offset;

                PostMasterHSM(PostEvent);

                slow_down_rvs_flag = 1;

        }
        else if ( (current_sonic_rvs_distance > SLOW_DOWN_RVS_OFFSET) &&
(slow_down_rvs_flag == 1) )
        {
                slow_down_rvs_flag = 0;

        }
```

```
        /*if ( (current_sonic_rvs_distance < HOME_RVS_DECELERATION_OFFSET) &&
(home2_flag == 0) )
        {
                PostEvent.EventType = Crossed_Home_Rvs_Deceleration_Offset;

                PostMasterHSM(PostEvent);

                home2_flag = 1;

        }
        else if ( (current_sonic_rvs_distance > HOME_RVS_DECELERATION_OFFSET)
&& (home2_flag == 1) )
        {
                home2_flag = 0;

        }*/


        if ( (current_sonic_fwd_distance < HOME_FWD_OFFSET) && (home_fwd_flag
== 0) )
        {
                PostEvent.EventType = Crossed_Home_Fwd_Offset;

                PostMasterHSM(PostEvent);

                home_fwd_flag = 1;

                                /*#ifdef MANUAL_COMMANDS

                                        if( toggle_command_switch == 1 )
                                        {

        PostEvent.EventType = pas_d_armes;


        PostMasterHSM(PostEvent);


        toggle_command_switch = 0;

                                        }
                                        else if( toggle_command_switch
== 0 )
                                        {
                                                PostEvent.EventType =
recess;


        PostMasterHSM(PostEvent);

                                                toggle_command_switch =
1;

                                        }

                #endif          */
```

```
        }
        else if ( (current_sonic_fwd_distance > HOME_RVS_OFFSET) &&
(home_fwd_flag == 1) )
        {
                home_fwd_flag = 0;

        }

        if ( (current_sonic_rvs_distance < HOME_RVS_OFFSET) && (home_rvs_flag
== 0) )
        {
                PostEvent.EventType = Crossed_Home_Rvs_Offset;

                PostMasterHSM(PostEvent);

                home_rvs_flag = 1;

        }
        else if ( (current_sonic_rvs_distance > HOME_RVS_OFFSET) &&
(home_rvs_flag == 1) )
        {
                home_rvs_flag = 0;

        }


/*
        if (current_home_fwd != last_home_fwd)
        {
                if(current_home_fwd == 1)
                {
                        //Only post an event if this is the third time that we
have crossed a line of tape
                        if(home_fwd_count == 2)
                        {
                                home_fwd_count = 0; //reset tape line counter

                                PostEvent.EventType = found_home_fwd;

                                PostMasterHSM(PostEvent);
                        }
                        else
                        {
                                home_fwd_count ++; //increment counter for the
number of lines that we have crossed
                        }
                }
                else
                {
                        PostEvent.EventType = lost_home_fwd;

                        PostMasterHSM(PostEvent);
                }
        }

        last_home_fwd = current_home_fwd ;
```

```
	if (current_home_rvs != last_home_rvs)
	{
		if(current_home_rvs == 1)
		{
			//Only post an event if this is the third time that we
have crossed a line of tape
			if(home_rvs_count == 2)
			{
				home_rvs_count = 0; //reset tape line counter

				PostEvent.EventType = found_home_rvs;

				PostMasterHSM(PostEvent);
			}
			else
			{
				home_rvs_count ++; //increment counter for the
number of lines that we have crossed
			}

		}
		else
		{
			PostEvent.EventType = lost_home_rvs;

			PostMasterHSM(PostEvent);
		}
	}

	last_home_rvs = current_home_rvs ;


	if (current_bumper_fwd != last_bumper_fwd)
	{
		if(current_bumper_fwd == 1)
		{
			PostEvent.EventType = front_bumper_pressed;

			PostMasterHSM(PostEvent);
		}
		else
		{
			//PostEvent.EventType = lost_bumper_fwd;

			//PostMasterHSM(PostEvent);
		}
	}

	last_bumper_fwd = current_bumper_fwd ;


	if (current_bumper_rvs != last_bumper_rvs)
	{
		if(current_bumper_rvs == 1)
		{
			PostEvent.EventType = rear_bumper_pressed;
```

```
                        PostMasterHSM(PostEvent);
                }
                else
                {
                //      PostEvent.EventType = lost_bumper_rvs;

                //      PostMasterHSM(PostEvent);
                }
        }

        last_bumper_rvs = current_bumper_rvs ;
        */


        //Set timer for next read of sensors
        ES_Timer_SetTimer(SENSOR_TIMER, SENSOR_INTERVAL_MS);
        ES_Timer_StartTimer(SENSOR_TIMER);

        /*
    //   If we are making a state transition
    if (MakeTransition == true)
    {
       //   Execute exit function for current state
       CurrentEvent.EventType = ES_EXIT;
       RunSensorService(CurrentEvent);

       CurrentState = NextState; //Modify state variable

       //   Execute entry function for new state
       // this defaults to ES_ENTRY
       RunSensorService(EntryEventKind);
     }
         */


        ReturnEvent.EventType = ES_NO_EVENT; // assume no errors


     return(ReturnEvent);
}


/****************************************************************************
 Function
     InitSensorService

 Parameters
     None

 Returns
     None

 Description
     Does any required initialization for this state machine
 Notes
```

```
 Author
     J. Edward Carryer, 2/18/99, 10:38AM
******************************************************************************/
bool InitSensorService ( uint8_t Priority )
{
   ES_Event ThisEvent;

   // local variable to get debugger to display the value of CurrentEvent
   //ES_Event LocalEvent = CurrentEvent;

    MyPriority = Priority;
   // to implement entry to a history state or directly to a substate
   // you can modify the initialization of the CurrentState variable
   // otherwise just start in the entry state every time the state machine
   // is started

      //set initial state
   CurrentState = SensorState;
   // call the entry function (if any) for the ENTRY_STATE
   //RunCalibrationServiceSM(CurrentEvent);

  #ifdef PRINT_SENSOR_CALLS

    printf("StartSensorService\n\r");

  #endif

    //Initialize analog port
    //ADS12_Init("OOOAIIII"); //Documentation suggests that first/leftmost
letter in string corresponds to MSB
        //ADS12_Init("IIIIAOOO");
     //Commented out so that this could be initialized in the MasterHSM init
function

        //Initialize last_Pot_Value to present value of Port AD0
    last_pot_input = ADS12_ReadADPin(POT_PIN);
        pot_value = (unsigned char)(((float)(last_pot_input))*100/1024);

        //Initialize dip switch value
        last_dipswitch_value = get_dipswitch();

        //Set timer for next read of calibration switches
        ES_Timer_SetTimer(SENSOR_TIMER, SENSOR_INTERVAL_MS);
        ES_Timer_StartTimer(SENSOR_TIMER);

        //Initialize navigation line tape sensor ports (pins 4-6) to be inputs
        DDRU &= BIT4LO;
        DDRU &= BIT5LO;
        DDRU &= BIT6LO;

        //Initialize home zone tape sensor ports (pins 6 and 7) to be inputs
        DDRU &= BIT7LO;
        DDRT &= BIT3LO;

        // Initialize Lance IR detector ports (pins 0 and 1) to inputs
        DDRP &= BIT0LO;
        DDRP &= BIT1LO;
```

```
        // Initialize Gun IR detector ports (pin 2) to be an input
        DDRP &= BIT2LO;

        //Initialize front and rear bump sensor (pins 0 and 1) ports to be
inputs
        DDRE &= BIT0LO;
        DDRE &= BIT1LO;

        //Initialize ports for reading ultrasonic pulses
        DDRT &= BIT6LO;
        DDRT &= BIT7LO;
        DDRT &= BIT0LO;
        DDRT &= BIT1LO;

        //Initialize port for sending trigger pulse to ultrasonic sensor
        DDRU |= BIT3HI;

        //Initialize Timer 1, Channel 5 to check for the presense of IR pulses
        TIM1_TSCR1 = _S12_TEN;                                                  // tur
        TIM1_TSCR2 |= _S12_PR2 | _S12_PR1 | _S12_PR0;        // set prescale
clock to / 128 = 187.5 kHZ
        TIM1_TIOS |= _S12_IOS5;
        // set up comp 5 to an output compare, leave the pins alone
        TIM1_TCTL1 &= ~(_S12_OL5 | _S12_OM5);
        TIM1_TC5 = TIM1_TCNT + BEACON_CHECK_PERIOD;  //initialize OC Timer
        TIM1_TFLG1 = _S12_C5F;
        TIM1_TIE |= _S12_C5I;


        //Initialize Timer 2, Channel 7 to send trigger pulse to ultrasonic
sensor
        TIM2_TSCR1 = _S12_TEN;                                                  // tur
        TIM2_TSCR2 |= _S12_PR2;        // set prescale clock to / 16
        TIM2_TSCR2 &= (~_S12_PR1)&(~_S12_PR0);
        TIM2_TIOS |= _S12_IOS7;
        // set up comp 7 to an output compare, leave the pins alone
        TIM2_TCTL1 &= ~(_S12_OL7 | _S12_OM7);
        TIM2_TC7 = TIM2_TCNT + ULTRASONIC_LOW_PERIOD;  //initialize OC Timer
        TIM2_TFLG1 = _S12_C7F;
        TIM2_TIE |= _S12_C7I;

        //Initialize ports T6 and T7 as input captures to measure pulse width
from front and back ultrasonic sensors
        TIM1_TIOS &= (~_S12_IOS6)&(~_S12_IOS7);
        TIM1_TCTL3 |= _S12_EDG6A|_S12_EDG6B|_S12_EDG7A|_S12_EDG7B; //capture
on any edge
        TIM1_TFLG1 = _S12_C6F; //clear flags
        TIM1_TFLG1 = _S12_C7F;
        TIM1_TIE |= _S12_C6I|_S12_C7I; //enable input capture ISRs

        //Initialize Timer 0 ports T4 and T5 as input captures to measure
pulse width from side ultrasonic sensors
        TIM0_TSCR1 |= _S12_TEN; // turn the TIM0 timer system ON
    TIM0_TSCR2 |= _S12_PR0 | _S12_PR1 | _S12_PR2; // set the pre-scale to
M/128 (187.5kHz clock)
        TIM0_TIOS &= (~_S12_IOS4)&(~_S12_IOS5);
```

```
        TIM0_TCTL3 |= _S12_EDG4A|_S12_EDG4B|_S12_EDG5A|_S12_EDG5B; //capture
on any edge
        TIM0_TFLG1 = _S12_C4F; //clear flags
        TIM0_TFLG1 = _S12_C5F;
        TIM0_TIE |= _S12_C4I|_S12_C5I; //enable input capture ISRs

        //initialize edge history for tracking pulses from the ultrasonic
sensors
        last_sonic_edge_fwd = TIM1_TCNT;
        last_sonic_edge_rvs = TIM1_TCNT;
    last_sonic_edge_side_fwd = TIM0_TCNT;
        last_sonic_edge_side_rvs = TIM0_TCNT;

        EnableInterrupts;

  // post the initial transition event
  ThisEvent.EventType = ES_INIT;
  if (ES_PostToService( MyPriority, ThisEvent) == true)
  {
      return true;
  }else
  {
      return false;
  }



}


/****************************************************************************
 Function
     QuerySensorServiceSM

 Parameters
     None

 Returns
     TemplateState_t The current state of the Template state machine

 Description
     returns the current state of the Template state machine
 Notes

 Author
     J. Edward Carryer, 2/11/05, 10:38AM
****************************************************************************/
TemplateState_t QuerySensorService ( void )
{
    #ifdef PRINT_SENSOR_CALLS

    printf("QuerySensorService\n\r");

  #endif

   return(CurrentState);
}
```

```
unsigned char get_cal_pot(void)
{
        return pot_value;
}

unsigned char get_pushbutton(void)
{
        unsigned char pushbutton = 0;

        if((PTIAD&PUSHBUTTON) == PUSHBUTTON)
        {
                pushbutton = 1;
        }

        return pushbutton;

}

unsigned char get_dipswitch(void)
{
        unsigned char dipswitch_1 = 0;
        unsigned char dipswitch_2 = 0;
        unsigned char dipswitch_3 = 0;
        unsigned char dipswitch_value;

        if((PTIAD&DIPSWITCH_1) == DIPSWITCH_1)
        {
                dipswitch_1 = 1;
        }

        if((PTIAD&DIPSWITCH_2) == DIPSWITCH_2)
        {
                dipswitch_2 = 1;
        }

        if((PTIAD&DIPSWITCH_3) == DIPSWITCH_3)
        {
                dipswitch_3 = 1;
        }

        dipswitch_value = 1*dipswitch_1 + 2*dipswitch_2 + 4*dipswitch_3;

        return dipswitch_value;

}

unsigned char get_dipswitch_1(void)
{
```