```c
/****************************************************************************
Module
Location.c

Description
This is the location control module.

****************************************************************************/
/*----------------------------- Include Files -----------------------------*/
/* include header files for the framework and this service
*/
#include "ES_Configure.h"
#include "ES_Framework.h"
#include "ES_DeferRecall.h"
#include "OwnPWM.h"
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "inc/hw_gpio.h"
#include "inc/hw_sysctl.h"
#include "driverlib/sysctl.h"
#include "driverlib/pin_map.h"  // Define PART_TM4C123GH6PM in project
#include "driverlib/gpio.h"
#include "inc/hw_timer.h"
#include "inc/hw_nvic.h"
#include "MasterVehicle.h"
#include "Location.h"
#include "Ultrasonic.h"
#include "PWMTiva.h"
/*----------------------------- Module Defines ----------------------------*/
// these times assume a 1.000mS/tick timing
#define ONE_SEC 976
#define HALF_SEC (ONE_SEC /2)
#define TWO_SEC (ONE_SEC *2)
#define FIVE_SEC (ONE_SEC *5)
#define TEN_SEC (ONE_SEC *10)
#define FORTY_FIVE_DEGREE_ROTATE_TIME 600
#define PWM_FREQ 1000
//#define FULL_DUTY 70 //Set for testing
#define RIGHT_FULL_DUTY 90 //Adjusted pwm for right wheel to ensure the robot runs straight
#define HALF_DUTY 60
#define SMALL_DUTY 40
#define STOP_DUTY 0
#define NF 0x08 //North Full Speed command
#define SF 0x09 //West Full Speed command
#define EF 0x10 //East Full Speed command
#define WF 0x11 //WestFull speed command
#define OneShotTimeout 5000000 //empirically determined, and quite arbitrary to be honest
#define CLOCK_RATE 40000000
#define test_tapedetect false
#define test_rotate_90_CW false
#define test_rotate_45_CW false
#define MAX_X 90 // Maximum location for x
#define MAX_Y 225 // Maximum location for y
#define DISTANCE_PER_PULSE 1
#define MAX_DUTY 99
#define MIN_DUTY 0
static uint32_t FULL_DUTY = 90;

//#define ENCODER_TRACKING
#define RPM_TRACKING
#define OPEN_LOOP //no control
#define LOCATION_TOLERANCE 1
#define REJECTION_TOLERANCE 100
#define Y_AXIS_OFFSET 222
#define USING_FRONT_BACK_ULTRASONIC
//#define NOT_USING_FRONT_BACK_ULTRASONIC
#define WALL_BREAK_DISTANCE 10
#define COMPETITION_FULL_DUTY 75
#define CORRECTION_FULL_DUTY 55
/*---------------------------- Module Functions -------------------------*/
/* prototypes for private functions for this service.They should be functions
relevant to the behavior of this service
```

```c
*/
void Init_PWM_port(void);
void runMotor (uint8_t mode);
static void rotateMotor_CW (void);
static void rotateMotor_CCW (void);
void stopMotor(void);
void set_PWM_Full_Duty(uint32_t input);
float clamp(float val, float clampL, float clampH);

/*---------------------------- Module Variables ---------------------------*/
// with the introduction of Gen2, we need a module level Priority variable
static uint32_t location_x = 0;
static uint32_t location_y = 0;
static uint32_t location_ultrasonic_x = 0;
static uint32_t location_ultrasonic_y = 0;
static uint32_t prev_location_x = 0;
static uint32_t prev_location_y = 0;
static uint32_t target_x = 0;
static uint32_t target_y = 0;
static uint32_t target_ultrasonic_x = 0;
static uint32_t target_ultrasonic_y = 0;
static uint32_t encoder1_count = 0;
static uint32_t encoder2_count = 0;
static uint32_t ThisPeriod_Encoder1=0;
static uint32_t ThisCapture_Encoder1=10;
static uint32_t LastCapture_Encoder1=0;
static uint32_t FlagOneShotTimeout_Encoder1=0;
static uint32_t ThisPeriod_Encoder2=0;
static uint32_t ThisCapture_Encoder2=10;
static uint32_t LastCapture_Encoder2=0;
static uint32_t FlagOneShotTimeout_Encoder2=0;
static float rpm_1 = 0;
static float rpm_2 = 0;
static float RPM_MULTIPLIER = 10704000;
static int Direction = 1; //set this to 1 for increase location, -1 for decrease location
static bool move_x_flag = false;
static bool move_y_flag = false;
static LocationState_t CurrentState;
static const uint32_t PERIOD_MULTIPLIER = 397245;  //from (40*(10^6)*60/(5.9*2*512))/(Period)
static float const iGain = 0.1;
static float const pGain = 2.1;
static float const dGain = 1;
static uint32_t PeriodicTimeout = 80000; //With 40MHz clock this is 2ms
//static uint32_t PeriodicTimeout = 40000; //With 40MHz clock this is 1ms
static int duty_mode = -1;
static uint8_t RequestedDuty = 0;
static float TRANSLATE_MULTIPLIER = 366.0/1600.0;
static bool use_location_checker = true;
static bool going_to_supply_flag = false;
/*------------------------------ Module Code ------------------------------*/

/* Initialize PWM, as well as input capture and one shot interrupt*/
void Init_Location(void){
  InitPeriodicInt();
  InitOwnPWM();
  OwnPWM_SetFreq(PWM_FREQ);
  Init_PWM_port();
  stopMotor();
  InitInputCapture_Encoder1();
  InitOneShotInt_Encoder1();
  InitInputCapture_Encoder2();
  InitOneShotInt_Encoder2();
  CurrentState = STOP;
}

/* Set Duty Cycle for PWM */
void set_PWM_Full_Duty(uint32_t input){
  FULL_DUTY = input;
}

/* Return current location x */
uint32_t get_X(void){
```

```c
  return location_x;
}

/* Return current location y */
uint32_t get_Y(void){
  return location_y;
}

/* To move in x direction*/
void move_X(uint32_t target_X){
  //update_Destination();
  target_x = target_X;
  encoder1_count = 0;
  encoder2_count = 0;
  move_x_flag = true;
  move_y_flag=  false;
  // compare target and current location to see which direction to move
  if(target_X >= location_x){
    Direction = 1; //tthis direction is EAST
    runMotor(WF);
  }
  else{
    Direction = -1; //this direction is WEST
    runMotor(EF);
  }
}

/* To move in y direction*/
void move_Y(uint32_t target_Y){
  //update_Destination();
  target_y = target_Y;
  encoder1_count = 0;
  encoder2_count = 0;
  move_x_flag = false;
  move_y_flag=  true;
  // compare target and current location to see which direction to move
  if(target_Y >= location_y){
    Direction = 1; //this direction is SOUTH
    runMotor(SF);
  }
  else{
    Direction = -1; //this direction is NORTH
    runMotor(NF);
  }
}

/* Return current state of the location service */
LocationState_t QueryLocationState ( void )
{
  return(CurrentState);
}


/****************************************************************************
private functions
****************************************************************************/

// For PWM digital control, we use we use port A 2,3
void Init_PWM_port(void){
  SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB);
  GPIOPinTypeGPIOOutput(GPIO_PORTB_BASE, GPIO_PIN_2);
  GPIOPinTypeGPIOOutput(GPIO_PORTB_BASE, GPIO_PIN_3);
}

void stopMotor (void){
  printf("\rStop motor\r\n");
  move_x_flag = false; //when we reach destination, clear flag
  move_y_flag = false;
  CurrentState = STOP;
  duty_mode = -1; // duty_mode for stopping is -1
  OwnPWM_SetDutyA(STOP_DUTY);
  OwnPWM_SetDutyB(STOP_DUTY);
```

```c
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_2, BIT2LO);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_3, BIT3LO);
  }

  /* Run motor fucntion set pwm duty cycle for channel A and B
  as well as the digital high and low for the two motor
  according to the direction it is given */
  void runMotor (uint8_t mode){
    if(mode == 0x08){ //Drive North Full speed
      duty_mode = 0;
      CurrentState = DRIVE_NORTH;
      OwnPWM_SetDutyA(MAX_DUTY-FULL_DUTY);
      OwnPWM_SetDutyB(MAX_DUTY-FULL_DUTY);
      GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_2, BIT2HI);
      GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_3, BIT3HI);
    }
    else if(mode == 0x09){ //Drive South full speed
      duty_mode = 1;
      CurrentState = DRIVE_SOUTH;
      OwnPWM_SetDutyA(FULL_DUTY);
      OwnPWM_SetDutyB(FULL_DUTY);
      GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_2, BIT2LO);
      GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_3, BIT3LO);
    }
    else if(mode == 0x10){ //Drive East full speed
      duty_mode = 2;
      CurrentState = DRIVE_EAST;
      OwnPWM_SetDutyA(FULL_DUTY);
      OwnPWM_SetDutyB(MAX_DUTY-FULL_DUTY);
      GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_2, BIT2LO);
      GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_3, BIT3HI);
    }
    else if(mode == 0x11){ // Drive West full speed
      duty_mode = 3;
      CurrentState = DRIVE_WEST;
      OwnPWM_SetDutyA(MAX_DUTY-FULL_DUTY);
      OwnPWM_SetDutyB(FULL_DUTY);
      GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_2, BIT2HI);
      GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_3, BIT3LO);
    }
    else{
      printf("\rInvalid Command Received for moving straight\r\n");
      }
  }

  void InitInputCapture_Encoder1( void ){
    // start by enabling the clock to the timer (Wide Timer 0)
    HWREG(SYSCTL_RCGCWTIMER) |= SYSCTL_RCGCWTIMER_R0;
    // enable the clock to Port C
    HWREG(SYSCTL_RCGCGPIO) |= SYSCTL_RCGCGPIO_R2;
    // since we added this Port C clock init, we can immediately start
    // into configuring the timer, no need for further delay
    // make sure that timer (Timer A) is disabled before configuring
    HWREG(WTIMER0_BASE+TIMER_O_CTL) &= ~TIMER_CTL_TAEN;
    // set it up in 32bit wide (individual, not concatenated) mode
    // the constant name derives from the 16/32 bit timer, but this is a 32/64
    // bit timer so we are setting the 32bit mode
    HWREG(WTIMER0_BASE+TIMER_O_CFG) = TIMER_CFG_16_BIT;
    // we want to use the full 32 bit count, so initialize the Interval Load
    // register to 0xffff.ffff (its default value :-)
    HWREG(WTIMER0_BASE+TIMER_O_TAILR) = 0xffffffff;
    // set up timer A in capture mode (TAMR=3, TAAMS = 0),
    // for edge time (TACMR = 1) and up-counting (TACDIR = 1)
    HWREG(WTIMER0_BASE+TIMER_O_TAMR) =
    (HWREG(WTIMER0_BASE+TIMER_O_TAMR) & ~TIMER_TAMR_TAAMS) |
    (TIMER_TAMR_TACDIR | TIMER_TAMR_TACMR | TIMER_TAMR_TAMR_CAP);
    // To set the event to rising edge, we need to modify the TAEVENT bits
    // in GPTMCTL. Rising edge = 00, so we clear the TAEVENT bits
    HWREG(WTIMER0_BASE+TIMER_O_CTL) &= ~TIMER_CTL_TAEVENT_M;
    // Now Set up the port to do the capture (clock was enabled earlier)
    // start by setting the alternate function for Port C bit 4 (WT0CCP0)
    HWREG(GPIO_PORTC_BASE+GPIO_O_AFSEL) |= BIT4HI;
```

```c
    // Then, map bit 4's alternate function to WT0CCP0
    // 7 is the mux value to select WT0CCP0, 16 to shift it over to the
    // right nibble for bit 4 (4 bits/nibble * 4 bits)
    HWREG(GPIO_PORTC_BASE+GPIO_O_PCTL) =
    (HWREG(GPIO_PORTC_BASE+GPIO_O_PCTL) & 0xfff0ffff) + (7<<16);
    // Enable pin on Port C for digital I/O
    HWREG(GPIO_PORTC_BASE+GPIO_O_DEN) |= BIT4HI;
    // make pin 4 on Port C into an input
    HWREG(GPIO_PORTC_BASE+GPIO_O_DIR) &= BIT4LO;
    // back to the timer to enable a local capture interrupt
    HWREG(WTIMER0_BASE+TIMER_O_IMR) |= TIMER_IMR_CAEIM;
    // enable the Timer A in Wide Timer 0 interrupt in the NVIC
    // it is interrupt number 94 so appears in EN2 at bit 30
    HWREG(NVIC_EN2) |= BIT30HI;
    // make sure interrupts are enabled globally
    __enable_irq();
    // now kick the timer off by enabling it and enabling the timer to
    // stall while stopped by the debugger
    HWREG(WTIMER0_BASE+TIMER_O_CTL) |= (TIMER_CTL_TAEN | TIMER_CTL_TASTALL);
  }


  void InputCaptureResponse_Encoder1( void ){
    //printf("In input capture1\r\n");
    #ifdef measurementMode
    togglePinStatePortA(3);//go measure PA3
    #endif
    //uint32_t ThisCapture;
    // start by clearing the source of the interrupt, the input capture event
    HWREG(WTIMER0_BASE+TIMER_O_ICR) = TIMER_ICR_CAECINT;
    //printf("In input capture ISR after clearing\n\r");
    //start the one shot timer
    StartOneShot_Encoder1();
    //clear the Flag for One shot because now we are at a new edge
    FlagOneShotTimeout_Encoder1=0;
    // now grab the captured value and calculate the period
    ThisCapture_Encoder1 = HWREG(WTIMER0_BASE+TIMER_O_TAR);
    ThisPeriod_Encoder1 = ThisCapture_Encoder1 - LastCapture_Encoder1;
    //printf("Period is %d\r\n", ThisPeriod_Encoder1);
    // update LastCapture to prepare for the next edge
    LastCapture_Encoder1 = ThisCapture_Encoder1;
    encoder1_count ++; //increase the encoder_count from the detected pulse
    rpm_1 = RPM_MULTIPLIER/ThisPeriod_Encoder1;
  }

  /* Return rpm of motor 1 */
  double get_RPM_Encoder1(void){
    return rpm_1;
  }

  /* Return rpm of motor 2 */
  double get_RPM_Encoder2(void){
    return rpm_2;
  }

  void InitOneShotInt_Encoder1( void ){
    // start by enabling the clock to the timer (Wide Timer 0)
    HWREG(SYSCTL_RCGCWTIMER) |= SYSCTL_RCGCWTIMER_R0;
    // kill a few cycles to let the clock get going
    while((HWREG(SYSCTL_PRWTIMER) & SYSCTL_PRWTIMER_R0) != SYSCTL_PRWTIMER_R0)
    {
    }
    // make sure that timer (Timer B) is disabled before configuring
    HWREG(WTIMER0_BASE+TIMER_O_CTL) &= ~TIMER_CTL_TBEN; //TBEN = Bit8
    // set it up in 32bit wide (individual, not concatenated) mode
    // the constant name derives from the 16/32 bit timer, but this is a 32/64
    // bit timer so we are setting the 32bit mode
    HWREG(WTIMER0_BASE+TIMER_O_CFG) = TIMER_CFG_16_BIT; //bits 0-2 = 0x04
    // set up timer B in 1-shot mode so that it disables timer on timeouts
    // first mask off the TAMR field (bits 0:1) then set the value for
    // 1-shot mode = 0x01
    HWREG(WTIMER0_BASE+TIMER_O_TBMR) =
```

```
    (HWREG(WTIMER0_BASE+TIMER_O_TBMR)& ~TIMER_TBMR_TBMR_M)|
    TIMER_TBMR_TBMR_1_SHOT;
    // set timeout
    HWREG(WTIMER0_BASE+TIMER_O_TBILR) = OneShotTimeout;
    // enable a local timeout interrupt. TBTOIM = bit 8
    HWREG(WTIMER0_BASE+TIMER_O_IMR) |= TIMER_IMR_TBTOIM; // 8
    // enable the Timer B in Wide Timer 0 interrupt in the NVIC
    // it is interrupt number 95 so appears in EN2 at bit 30
    HWREG(NVIC_EN2) |= BIT31HI;
    // make sure interrupts are enabled globally
    __enable_irq();
    //StartTime = ES_Timer_GetTime();
    // now kick the timer off by enabling it and enabling the timer to
    // stall while stopped by the debugger. TAEN = Bit0, TASTALL = bit1
    HWREG(WTIMER0_BASE+TIMER_O_CTL) |= (TIMER_CTL_TBEN | TIMER_CTL_TBSTALL);
  }

  void StartOneShot_Encoder1( void ){
    // start by grabbing the start time
    //StartTime = ES_Timer_GetTime();
    // now kick the timer off by enabling it and enabling the timer to
    // stall while stopped by the debugger
    HWREG(WTIMER0_BASE+TIMER_O_CTL) |= (TIMER_CTL_TBEN | TIMER_CTL_TBSTALL);
  }

  void OneShotIntResponse_Encoder1( void ){
    // start by clearing the source of the interrupt
    HWREG(WTIMER0_BASE+TIMER_O_ICR) = TIMER_ICR_TBTOCINT;
    //printf("In one shot ISR after clearing\n\r");
    FlagOneShotTimeout_Encoder1=1;
  }


  void InitInputCapture_Encoder2( void ){
    // start by enabling the clock to the timer (Wide Timer 1)
    HWREG(SYSCTL_RCGCWTIMER) |= SYSCTL_RCGCWTIMER_R1;
    // enable the clock to Port C
    HWREG(SYSCTL_RCGCGPIO) |= SYSCTL_RCGCGPIO_R2;
    // since we added this Port C clock init, we can immediately start
    // into configuring the timer, no need for further delay
    // make sure that timer (Timer A) is disabled before configuring
    HWREG(WTIMER1_BASE+TIMER_O_CTL) &= ~TIMER_CTL_TAEN;
    // set it up in 32bit wide (individual, not concatenated) mode
    // the constant name derives from the 16/32 bit timer, but this is a 32/64
    // bit timer so we are setting the 32bit mode
    HWREG(WTIMER1_BASE+TIMER_O_CFG) = TIMER_CFG_16_BIT;
    // we want to use the full 32 bit count, so initialize the Interval Load
    // register to 0xffff.ffff (its default value :-)
    HWREG(WTIMER1_BASE+TIMER_O_TAILR) = 0xffffffff;
    // set up timer A in capture mode (TAMR=3, TAAMS = 0),
    // for edge time (TACMR = 1) and up-counting (TACDIR = 1)
    HWREG(WTIMER1_BASE+TIMER_O_TAMR) =
    (HWREG(WTIMER1_BASE+TIMER_O_TAMR) & ~TIMER_TAMR_TAAMS) |
    (TIMER_TAMR_TACDIR | TIMER_TAMR_TACMR | TIMER_TAMR_TAMR_CAP);
    // To set the event to rising edge, we need to modify the TAEVENT bits
    // in GPTMCTL. Rising edge = 00, so we clear the TAEVENT bits
    HWREG(WTIMER1_BASE+TIMER_O_CTL) &= ~TIMER_CTL_TAEVENT_M;
    // Now Set up the port to do the capture (clock was enabled earlier)
    // start by setting the alternate function for Port C bit 6 (WT1CCP0)
    HWREG(GPIO_PORTC_BASE+GPIO_O_AFSEL) |= BIT6HI;
    // Then, map bit 4's alternate function to WT0CCP0
    // 7 is the mux value to select WT0CCP0, 16 to shift it over to the
    // right nibble for bit 6 (4 bits/nibble * 6 bits)
    HWREG(GPIO_PORTC_BASE+GPIO_O_PCTL) =
    (HWREG(GPIO_PORTC_BASE+GPIO_O_PCTL) & 0xf0ffffff) + (7<<24);
    // Enable pin on Port C for digital I/O
    HWREG(GPIO_PORTC_BASE+GPIO_O_DEN) |= BIT6HI;
    // make pin 4 on Port C into an input
    HWREG(GPIO_PORTC_BASE+GPIO_O_DIR) &= BIT6LO;
    // back to the timer to enable a local capture interrupt
    HWREG(WTIMER1_BASE+TIMER_O_IMR) |= TIMER_IMR_CAEIM;
    // enable the Timer A in Wide Timer 0 interrupt in the NVIC
```

```
  // it is interrupt number 94 so appears in EN2 at bit 30
  HWREG(NVIC_EN3) |= BIT0HI;
  // make sure interrupts are enabled globally
  __enable_irq();
  // now kick the timer off by enabling it and enabling the timer to
  // stall while stopped by the debugger
  HWREG(WTIMER1_BASE+TIMER_O_CTL) |= (TIMER_CTL_TAEN | TIMER_CTL_TASTALL);
}


void InputCaptureResponse_Encoder2( void ){
  //printf("In input capture2\r\n");
  #ifdef measurementMode
  togglePinStatePortA(3);//go measure PA3
  #endif
  //uint32_t ThisCapture;
  // start by clearing the source of the interrupt, the input capture event
  HWREG(WTIMER1_BASE+TIMER_O_ICR) = TIMER_ICR_CAECINT;
  //printf("In input capture ISR after clearing\n\r");
  //start the one shot timer
  StartOneShot_Encoder2();
  //clear the Flag for One shot because now we are at a new edge
  FlagOneShotTimeout_Encoder2=0;
  // now grab the captured value and calculate the period
  ThisCapture_Encoder2 = HWREG(WTIMER1_BASE+TIMER_O_TAR);
  ThisPeriod_Encoder2 = ThisCapture_Encoder2 - LastCapture_Encoder2;
  // update LastCapture to prepare for the next edge
  LastCapture_Encoder2 = ThisCapture_Encoder2;
  encoder2_count ++; //increase the encoder_count from the detected pulse
  rpm_2 = RPM_MULTIPLIER/ThisPeriod_Encoder2;
}

/* Return encoder value of motor 1 */
uint32_t get_encoder1_count(void){
  return encoder1_count;
}

/* Return encoder value of motor 2 */
uint32_t get_encoder2_count(void){
  return encoder2_count;
}

void InitOneShotInt_Encoder2( void ){
  // start by enabling the clock to the timer (Wide Timer 1)
  HWREG(SYSCTL_RCGCWTIMER) |= SYSCTL_RCGCWTIMER_R1;
  // kill a few cycles to let the clock get going
  while((HWREG(SYSCTL_PRWTIMER) & SYSCTL_PRWTIMER_R1) != SYSCTL_PRWTIMER_R1)
  {
  }
  // make sure that timer (Timer B) is disabled before configuring
  HWREG(WTIMER1_BASE+TIMER_O_CTL) &= ~TIMER_CTL_TBEN; //TBEN = Bit8
  // set it up in 32bit wide (individual, not concatenated) mode
  // the constant name derives from the 16/32 bit timer, but this is a 32/64
  // bit timer so we are setting the 32bit mode
  HWREG(WTIMER1_BASE+TIMER_O_CFG) = TIMER_CFG_16_BIT; //bits 0-2 = 0x04
  // set up timer B in 1-shot mode so that it disables timer on timeouts
  // first mask off the TAMR field (bits 0:1) then set the value for
  // 1-shot mode = 0x01
  HWREG(WTIMER1_BASE+TIMER_O_TBMR) =
  (HWREG(WTIMER1_BASE+TIMER_O_TBMR)& ~TIMER_TBMR_TBMR_M)|
  TIMER_TBMR_TBMR_1_SHOT;
  // set timeout
  HWREG(WTIMER1_BASE+TIMER_O_TBILR) = OneShotTimeout;
  // enable a local timeout interrupt. TBTOIM = bit 8
  HWREG(WTIMER1_BASE+TIMER_O_IMR) |= TIMER_IMR_TBTOIM; // 8
  // enable the Timer B in Wide Timer 0 interrupt in the NVIC
  // it is interrupt number 97 so appears in EN3 at bit 1
  HWREG(NVIC_EN3) |= BIT1HI;
  // make sure interrupts are enabled globally
  __enable_irq();
  //StartTime = ES_Timer_GetTime();
  // now kick the timer off by enabling it and enabling the timer to
```

```
  // stall while stopped by the debugger. TAEN = Bit0, TASTALL = bit1
  HWREG(WTIMER1_BASE+TIMER_O_CTL) |= (TIMER_CTL_TBEN | TIMER_CTL_TBSTALL);
}

void StartOneShot_Encoder2( void ){
  // start by grabbing the start time
  //StartTime = ES_Timer_GetTime();
  // now kick the timer off by enabling it and enabling the timer to
  // stall while stopped by the debugger
  HWREG(WTIMER1_BASE+TIMER_O_CTL) |= (TIMER_CTL_TBEN | TIMER_CTL_TBSTALL);
}

void OneShotIntResponse_Encoder2( void ){
  // start by clearing the source of the interrupt
  HWREG(WTIMER1_BASE+TIMER_O_ICR) = TIMER_ICR_TBTOCINT;
  //printf("In one shot ISR after clearing\n\r");
  FlagOneShotTimeout_Encoder2 = 1;
}

void InitPeriodicInt( void ){
  // start by enabling the clock to the timer (Wide Timer 3)
  HWREG(SYSCTL_RCGCWTIMER) |= SYSCTL_RCGCWTIMER_R3; // kill a few cycles to let the clock get going
  while((HWREG(SYSCTL_PRWTIMER) & SYSCTL_PRWTIMER_R3) != SYSCTL_PRWTIMER_R3){}
  // make sure that timer (Timer A) is disabled before configuring
  HWREG(WTIMER3_BASE+TIMER_O_CTL) &= ~TIMER_CTL_TAEN;
  // set it up in 32bit wide (individual, not concatenated) mode
  HWREG(WTIMER3_BASE+TIMER_O_CFG) = TIMER_CFG_16_BIT;
  // set up timer A in periodic mode so that it repeats the time-outs
  HWREG(WTIMER3_BASE+TIMER_O_TAMR) =
  (HWREG(WTIMER3_BASE+TIMER_O_TAMR)& ~TIMER_TAMR_TAMR_M)|
  TIMER_TAMR_TAMR_PERIOD;
  // set timeout to 2mS
  HWREG(WTIMER3_BASE+TIMER_O_TAILR) = PeriodicTimeout;
  // enable a local timeout interrupt
  HWREG(WTIMER3_BASE+TIMER_O_IMR) |= TIMER_IMR_TATOIM;
  // enable the Timer A in Wide Timer 0 interrupt in the NVIC
  // it is interrupt number 100 so appears in EN3 at bit 4
  HWREG(NVIC_EN3) = BIT4HI;
  HWREG(NVIC_PRI25)= ((HWREG(NVIC_PRI25)) & ~NVIC_PRI25_INTA_M) | NVIC_PRI25_INTA_M;
  // make sure interrupts are enabled globally
  __enable_irq();
  // now kick the timer off by enabling it and enabling the timer to
  // stall while stopped by the debugger
  HWREG(WTIMER3_BASE+TIMER_O_CTL) |= (TIMER_CTL_TAEN |TIMER_CTL_TASTALL);
}

/* This periodic response is where our control happens */
void PeriodicIntResponse( void ){
  static float IntegralTerm=0.0;
  /* integrator control effort */
  static float RPMError;
  /* make static for speed */
  static float LastError;
  /* for Derivative Control */
  //static uint32_t rpm_EN1;
  /* make static for speed */
  // start by clearing the source of the interrupt HWREG(WTIMER0_BASE+TIMER_O_ICR) = TIMER_ICR_TATOCINT;
  HWREG(WTIMER3_BASE+TIMER_O_ICR) = TIMER_ICR_TATOCINT;
  #ifdef RPM_TRACKING
  float RPM_1 = rpm_1;
  float RPM_2 = rpm_2;
  RPMError = RPM_1 - RPM_2;
  IntegralTerm += iGain * RPMError;
  //IntegralTerm = clamp(IntegralTerm, 30, MAX_DUTY);
  /* anti-windup */
  RequestedDuty =(pGain * ((RPMError)+IntegralTerm+((-1)*dGain * (RPMError-LastError))));
  RequestedDuty = clamp(RequestedDuty, MIN_DUTY, MAX_DUTY);

  //anti-wind up
  if (RequestedDuty == MAX_DUTY){
    IntegralTerm=IntegralTerm-iGain*RPMError;
  }
```

```c
    if (RequestedDuty == MIN_DUTY){
      IntegralTerm=IntegralTerm+iGain*RPMError;
    }

    // if we don't include control in our drive
    #ifdef OPEN_LOOP
    RequestedDuty = FULL_DUTY; //always set requested duty to full duty
    #endif

    LastError = RPMError; // update last error

    //check duty mode to make sure the motor is in running mode
    if(duty_mode != -1){
      if(duty_mode == 0 | duty_mode == 2){
        OwnPWM_SetDutyB(MAX_DUTY-RequestedDuty);
      }
      else{
        OwnPWM_SetDutyB((uint32_t) (RequestedDuty));
      }
    }
    #endif

    // in case we want to track encoder count instead
    #ifdef ENCODER_TRACKING
    int ENCODER1 = encoder1_count;
    int ENCODER2 = encoder2_count;
    RPMError = ENCODER1 - ENCODER2;
    IntegralTerm += iGain * RPMError;
    IntegralTerm = clamp(IntegralTerm, 0, MAX_DUTY);
    /* anti-windup */
    RequestedDuty =(pGain * ((RPMError)+IntegralTerm+(dGain * (RPMError-LastError))));
    RequestedDuty = clamp(RequestedDuty, 0, MAX_DUTY);
    LastError = RPMError; // update last error
    #ifdef OPEN_LOOP
    RequestedDuty = FULL_DUTY;
    #endif
    if(duty_mode != -1){
      if(duty_mode == 0 | duty_mode == 3){
        OwnPWM_SetDutyB(MAX_DUTY-RequestedDuty);
      }
      else{
        OwnPWM_SetDutyB((uint32_t) (RequestedDuty));
      }
    }
    #endif
  }

  /* Return requested duty value which is calculated in control loop */
  uint8_t get_Duty(void){
    return RequestedDuty;
  }

  // constrain val to be in range clampL to clampH
  float clamp(float val, float clampL, float clampH){
    if (val > clampH) { // if too high
      return clampH;
    }
    if (val < clampL) { // if too low
      return clampL;
    }
    return val; // if OK as-is
  }

  void updateLocation_from_Ultrasonic(void){
    prev_location_x = location_x;
    prev_location_y = location_y;
    location_x = get_Ultrasonic_X();
    uint32_t y_f = get_Ultrasonic_Y_F(); // get value for y front
    uint32_t y_b = get_Ultrasonic_Y_B(); // get value for y back

    // if using two sensors for y axis
    #ifdef USING_FRONT_BACK_ULTRASONIC
```

```c
    // the idea here is that we have two ultrasonic reading: one is in the front
    // of the robot and the other one is in the back. The assumption here is that
    // the closer one is to the wall, the more accurate the reading will be.
    // thus, when we have to reading, we will choose the get the value from the
    // reading that is lower (closer to the wall). We have to adjust the reading
    // accordingly if we use the one in the back.
    if( y_f <= y_b){
      location_y = y_f;
    }
    else{
      // when using back reading, we have to subtract it from the offset
      // to ge a uniform value with the one we get from the front reading
      location_y = Y_AXIS_OFFSET - y_b;
    }
    #endif

    // if using only one sensor in the front for y axis
    #ifdef NOT_USING_FRONT_BACK_ULTRASONIC
    location_y = y_f; // NOW WE USE THE LOCATION FROM THE FRONT
    #endif

    // here, we validate the reading value, if more than maximum limit, we reject it
    if(location_x > MAX_X){
      location_x = prev_location_x;
    }
    if(location_y >MAX_Y){
      printf("location_y more than max, reject this y\r\n");
      location_y = prev_location_y;
    }

    // here, we reject value that is too different from the reading before this
    // we assume the robot cannot move too far during the inveral of this reading
    if(prev_location_x != 0){
      if (location_x > prev_location_x){
        if(location_x - prev_location_x > REJECTION_TOLERANCE){
          location_x = prev_location_x;
        }
      }
      else if(prev_location_x > location_x){
        if(prev_location_x - location_x > REJECTION_TOLERANCE){
          location_x = prev_location_x;
        }
      }
    }
  }

  /* Return current location values */
  uint32_t get_current_X(void){
    return location_x;
  }

  uint32_t get_current_Y(void){
    return location_y;
  }

  /* Set this flag to use location checker and vice versa*/
  void set_location_checker_flag(bool input){
    use_location_checker = input;
  }

  /* This function check if the destination is reached */
  void Location_Checker( void){
    updateLocation_from_Ultrasonic(); // first update the current readings
    if(use_location_checker){
      if(move_x_flag){ //check location only when we are moving (x or y)
        if((target_x <= location_x + LOCATION_TOLERANCE) && (target_x >= location_x - LOCATION_TOLERANCE)){
          // if the location we are and the target is at the same place,
          // post REACH event to main
          stopMotor();
          ES_Event to_post1;
          to_post1.EventType = X_REACHED;
          move_x_flag = false; //when we reach destination, clear flag
```

```c
          move_y_flag = false;
          PostMasterVehicleSM(to_post1);
        }
        else{
          // do nothing, don't post anything
        }
      }
      else if(move_y_flag){
        //if the location we are and the target is at the same place,
        // post REACH event to main
        if((target_y <= location_y + LOCATION_TOLERANCE) && (target_y >= location_y - LOCATION_TOLERANCE)){
          stopMotor();
          ES_Event to_post2;
          to_post2.EventType = Y_REACHED;
          move_y_flag = false; //when we reach destination, clear flag
          move_x_flag = false;
          PostMasterVehicleSM(to_post2);
        }
        else{
          // do nothing, don't post anything
        }
      }
    }
    // for this special flag, when the robot is arriving at the wall, we reduce the speed to avoid hard collision
    if(going_to_supply_flag){
      if(location_y < WALL_BREAK_DISTANCE){
        set_PWM_Full_Duty(CORRECTION_FULL_DUTY);
        going_to_supply_flag = false;
      }
    }
  }

  /* set going to supply flag */
  void set_going_to_supply_flag(void){
    going_to_supply_flag = true;
  }

  /* clear going to supply flag */
  void clear_going_to_supply_flag(void){
    going_to_supply_flag = false;
  }

  /* functions for checking whether target location is reached or not */
  bool verify_x_location(void){
    printf("Running verify X location\r\n");
    if((target_x <= location_x + LOCATION_TOLERANCE) &&
    (target_x >= location_x - LOCATION_TOLERANCE)){
      return true;
    }
    else{
      return false;
    }
  }

  bool verify_y_location(void){
    printf("Running verify Y location\r\n");
    if((target_y <= location_y + LOCATION_TOLERANCE) &&
    (target_y >= location_y - LOCATION_TOLERANCE)){
      return true;
    }
    else{
      return false;
    }
  }

```