

```

1  /*****
2  Module
3  Location.c
4
5  Description
6  This is the location control module.
7
8  *****/
9  /*----- Include Files -----*/
10 /* include header files for the framework and this service
11 */
12 #include "ES_Configure.h"
13 #include "ES_Framework.h"
14 #include "ES_DeferRecall.h"
15 #include "OwnPWM.h"
16 #include "inc/hw_memmap.h"
17 #include "inc/hw_types.h"
18 #include "inc/hw_gpio.h"
19 #include "inc/hw_sysctl.h"
20 #include "driverlib/sysctl.h"
21 #include "driverlib/pin_map.h" // Define PART_TM4C123GH6PM in project
22 #include "driverlib/gpio.h"
23 #include "inc/hw_timer.h"
24 #include "inc/hw_nvic.h"
25 #include "MasterVehicle.h"
26 #include "Location.h"
27 #include "Ultrasonic.h"
28 #include "PWMTiva.h"
29 /*----- Module Defines -----*/
30 // these times assume a 1.000mS/tick timing
31 #define ONE_SEC 976
32 #define HALF_SEC (ONE_SEC /2)
33 #define TWO_SEC (ONE_SEC *2)
34 #define FIVE_SEC (ONE_SEC *5)
35 #define TEN_SEC (ONE_SEC *10)
36 #define FORTY_FIVE_DEGREE_ROTATE_TIME 600
37 #define PWM_FREQ 1000
38 // #define FULL_DUTY 70 //Set for testing
39 #define RIGHT_FULL_DUTY 90 //Adjusted pwm for right wheel to ensure the robot runs straight
40 #define HALF_DUTY 60
41 #define SMALL_DUTY 40
42 #define STOP_DUTY 0
43 #define NF 0x08 //North Full Speed command
44 #define SF 0x09 //West Full Speed command
45 #define EF 0x10 //East Full Speed command
46 #define WF 0x11 //WestFull speed command
47 #define OneShotTimeout 5000000 //empirically determined, and quite arbitrary to be honest
48 #define CLOCK_RATE 40000000
49 #define test_tapedetect false
50 #define test_rotate_90_CW false
51 #define test_rotate_45_CW false
52 #define MAX_X 90 // Maximum location for x
53 #define MAX_Y 225 // Maximum location for y
54 #define DISTANCE_PER_PULSE 1
55 #define MAX_DUTY 99
56 #define MIN_DUTY 0
57 static uint32_t FULL_DUTY = 90;
58
59 // #define ENCODER_TRACKING
60 #define RPM_TRACKING
61 #define OPEN_LOOP //no control
62 #define LOCATION_TOLERANCE 1
63 #define REJECTION_TOLERANCE 100
64 #define Y_AXIS_OFFSET 222
65 #define USING_FRONT_BACK_ULTRASONIC
66 // #define NOT_USING_FRONT_BACK_ULTRASONIC
67 #define WALL_BREAK_DISTANCE 10
68 #define COMPETITION_FULL_DUTY 75
69 #define CORRECTION_FULL_DUTY 55
70 /*----- Module Functions -----*/
71 /* prototypes for private functions for this service.They should be functions
72 relevant to the behavior of this service

```

```

73  */
74  void Init_PWM_port(void);
75  void runMotor(uint8_t mode);
76  static void rotateMotor_CW(void);
77  static void rotateMotor_CCW(void);
78  void stopMotor(void);
79  void set_PWM_Full_Duty(uint32_t input);
80  float clamp(float val, float clampL, float clampH);
81
82  /*----- Module Variables -----*/
83  // with the introduction of Gen2, we need a module level Priority variable
84  static uint32_t location_x = 0;
85  static uint32_t location_y = 0;
86  static uint32_t location_ultrasonic_x = 0;
87  static uint32_t location_ultrasonic_y = 0;
88  static uint32_t prev_location_x = 0;
89  static uint32_t prev_location_y = 0;
90  static uint32_t target_x = 0;
91  static uint32_t target_y = 0;
92  static uint32_t target_ultrasonic_x = 0;
93  static uint32_t target_ultrasonic_y = 0;
94  static uint32_t encoder1_count = 0;
95  static uint32_t encoder2_count = 0;
96  static uint32_t ThisPeriod_Encoder1=0;
97  static uint32_t ThisCapture_Encoder1=10;
98  static uint32_t LastCapture_Encoder1=0;
99  static uint32_t FlagOneShotTimeout_Encoder1=0;
100 static uint32_t ThisPeriod_Encoder2=0;
101 static uint32_t ThisCapture_Encoder2=10;
102 static uint32_t LastCapture_Encoder2=0;
103 static uint32_t FlagOneShotTimeout_Encoder2=0;
104 static float rpm_1 = 0;
105 static float rpm_2 = 0;
106 static float RPM_MULTIPLIER = 10704000;
107 static int Direction = 1; //set this to 1 for increase location, -1 for decrease location
108 static bool move_x_flag = false;
109 static bool move_y_flag = false;
110 static LocationState_t CurrentState;
111 static const uint32_t PERIOD_MULTIPLIER = 397245; //from (40*(10^6)*60/(5.9*2*512))/(Period)
112 static float const iGain = 0.1;
113 static float const pGain = 2.1;
114 static float const dGain = 1;
115 static uint32_t PeriodicTimeout = 80000; //With 40MHz clock this is 2ms
116 //static uint32_t PeriodicTimeout = 40000; //With 40MHz clock this is 1ms
117 static int duty_mode = -1;
118 static uint8_t RequestedDuty = 0;
119 static float TRANSLATE_MULTIPLIER = 366.0/1600.0;
120 static bool use_location_checker = true;
121 static bool going_to_supply_flag = false;
122 /*----- Module Code -----*/
123
124 /* Initialize PWM, as well as input capture and one shot interrupt*/
125 void Init_Location(void){
126     InitPeriodicInt();
127     InitOwnPWM();
128     OwnPWM_SetFreq(PWM_FREQ);
129     Init_PWM_port();
130     stopMotor();
131     InitInputCapture_Encoder1();
132     InitOneShotInt_Encoder1();
133     InitInputCapture_Encoder2();
134     InitOneShotInt_Encoder2();
135     CurrentState = STOP;
136 }
137
138 /* Set Duty Cycle for PWM */
139 void set_PWM_Full_Duty(uint32_t input){
140     FULL_DUTY = input;
141 }
142
143 /* Return current location x */
144 uint32_t get_X(void){

```

```

145     return location_x;
146 }
147
148 /* Return current location y */
149 uint32_t get_Y(void){
150     return location_y;
151 }
152
153 /* To move in x direction*/
154 void move_X(uint32_t target_X){
155     //update_Destination();
156     target_x = target_X;
157     encoder1_count = 0;
158     encoder2_count = 0;
159     move_x_flag = true;
160     move_y_flag= false;
161     // compare target and current location to see which direction to move
162     if(target_X >= location_x){
163         Direction = 1; //this direction is EAST
164         runMotor(WF);
165     }
166     else{
167         Direction = -1; //this direction is WEST
168         runMotor(EF);
169     }
170 }
171
172 /* To move in y direction*/
173 void move_Y(uint32_t target_Y){
174     //update_Destination();
175     target_y = target_Y;
176     encoder1_count = 0;
177     encoder2_count = 0;
178     move_x_flag = false;
179     move_y_flag= true;
180     // compare target and current location to see which direction to move
181     if(target_Y >= location_y){
182         Direction = 1; //this direction is SOUTH
183         runMotor(SF);
184     }
185     else{
186         Direction = -1; //this direction is NORTH
187         runMotor(NF);
188     }
189 }
190
191 /* Return current state of the location service */
192 LocationState_t QueryLocationState ( void )
193 {
194     return(CurrentState);
195 }
196
197
198 /*****
199 private functions
200 *****/
201
202 // For PWM digital control, we use we use port A 2,3
203 void Init_PWM_port(void){
204     SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB);
205     GPIOPinTypeGPIOOutput(GPIO_PORTB_BASE, GPIO_PIN_2);
206     GPIOPinTypeGPIOOutput(GPIO_PORTB_BASE, GPIO_PIN_3);
207 }
208
209 void stopMotor (void){
210     printf("\rStop motor\r\n");
211     move_x_flag = false; //when we reach destination, clear flag
212     move_y_flag = false;
213     CurrentState = STOP;
214     duty_mode = -1; // duty_mode for stopping is -1
215     OwnPWM_SetDutyA(STOP_DUTY);
216     OwnPWM_SetDutyB(STOP_DUTY);

```

```

217     GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_2, BIT2LO);
218     GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_3, BIT3LO);
219 }
220
221 /* Run motor fucntion set pwm duty cycle for channel A and B
222 as well as the digital high and low for the two motor
223 according to the direction it is given */
224 void runMotor (uint8_t mode){
225     if(mode == 0x08){ //Drive North Full speed
226         duty_mode = 0;
227         CurrentState = DRIVE_NORTH;
228         OwnPWM_SetDutyA(MAX_DUTY-FULL_DUTY);
229         OwnPWM_SetDutyB(MAX_DUTY-FULL_DUTY);
230         GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_2, BIT2HI);
231         GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_3, BIT3HI);
232     }
233     else if(mode == 0x09){ //Drive South full speed
234         duty_mode = 1;
235         CurrentState = DRIVE_SOUTH;
236         OwnPWM_SetDutyA(FULL_DUTY);
237         OwnPWM_SetDutyB(FULL_DUTY);
238         GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_2, BIT2LO);
239         GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_3, BIT3LO);
240     }
241     else if(mode == 0x10){ //Drive East full speed
242         duty_mode = 2;
243         CurrentState = DRIVE_EAST;
244         OwnPWM_SetDutyA(FULL_DUTY);
245         OwnPWM_SetDutyB(MAX_DUTY-FULL_DUTY);
246         GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_2, BIT2LO);
247         GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_3, BIT3HI);
248     }
249     else if(mode == 0x11){ // Drive West full speed
250         duty_mode = 3;
251         CurrentState = DRIVE_WEST;
252         OwnPWM_SetDutyA(MAX_DUTY-FULL_DUTY);
253         OwnPWM_SetDutyB(FULL_DUTY);
254         GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_2, BIT2HI);
255         GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_3, BIT3LO);
256     }
257     else{
258         printf("\rInvalid Command Received for moving straight\r\n");
259     }
260 }
261
262 void InitInputCapture_Encoder1( void ){
263     // start by enabling the clock to the timer (Wide Timer 0)
264     HWREG(SYSCTL_RCGCWTIMER) |= SYSCTL_RCGCWTIMER_R0;
265     // enable the clock to Port C
266     HWREG(SYSCTL_RCGCGPIO) |= SYSCTL_RCGCGPIO_R2;
267     // since we added this Port C clock init, we can immediately start
268     // into configuring the timer, no need for further delay
269     // make sure that timer (Timer A) is disabled before configuring
270     HWREG(WTIMER0_BASE+TIMER_O_CTL) &= ~TIMER_CTL_TAEN;
271     // set it up in 32bit wide (individual, not concatenated) mode
272     // the constant name derives from the 16/32 bit timer, but this is a 32/64
273     // bit timer so we are setting the 32bit mode
274     HWREG(WTIMER0_BASE+TIMER_O_CFG) = TIMER_CFG_16_BIT;
275     // we want to use the full 32 bit count, so initialize the Interval Load
276     // register to 0xffff.ffff (its default value :-))
277     HWREG(WTIMER0_BASE+TIMER_O_TAILR) = 0xffffffff;
278     // set up timer A in capture mode (TAMR=3, TAAMS = 0),
279     // for edge time (TACMR = 1) and up-counting (TACDIR = 1)
280     HWREG(WTIMER0_BASE+TIMER_O_TAMR) =
281     (HWREG(WTIMER0_BASE+TIMER_O_TAMR) & ~TIMER_TAMR_TAAMS) |
282     (TIMER_TAMR_TACDIR | TIMER_TAMR_TACMR | TIMER_TAMR_TAMR_CAP);
283     // To set the event to rising edge, we need to modify the TAEVENT bits
284     // in GPTMCTL. Rising edge = 00, so we clear the TAEVENT bits
285     HWREG(WTIMER0_BASE+TIMER_O_CTL) &= ~TIMER_CTL_TAEVENT_M;
286     // Now Set up the port to do the capture (clock was enabled earlier)
287     // start by setting the alternate function for Port C bit 4 (WTOCCP0)
288     HWREG(GPIO_PORTC_BASE+GPIO_O_AFSEL) |= BIT4HI;

```

```

289 // Then, map bit 4's alternate function to WtOCCP0
290 // 7 is the mux value to select WtOCCP0, 16 to shift it over to the
291 // right nibble for bit 4 (4 bits/nibble * 4 bits)
292 HWREG(GPIO_PORTC_BASE+GPIO_O_PCTL) =
293 (HWREG(GPIO_PORTC_BASE+GPIO_O_PCTL) & 0xffff0fff) + (7<<16);
294 // Enable pin on Port C for digital I/O
295 HWREG(GPIO_PORTC_BASE+GPIO_O_DEN) |= BIT4HI;
296 // make pin 4 on Port C into an input
297 HWREG(GPIO_PORTC_BASE+GPIO_O_DIR) &= BIT4LO;
298 // back to the timer to enable a local capture interrupt
299 HWREG(WTIMER0_BASE+TIMER_O_IMR) |= TIMER_IMR_CAEIM;
300 // enable the Timer A in Wide Timer 0 interrupt in the NVIC
301 // it is interrupt number 94 so appears in EN2 at bit 30
302 HWREG(NVIC_EN2) |= BIT30HI;
303 // make sure interrupts are enabled globally
304 __enable_irq();
305 // now kick the timer off by enabling it and enabling the timer to
306 // stall while stopped by the debugger
307 HWREG(WTIMER0_BASE+TIMER_O_CTL) |= (TIMER_CTL_TAEN | TIMER_CTL_TASTALL);
308 }
309
310
311 void InputCaptureResponse_Encoder1( void ){
312 //printf("In input capture1\r\n");
313 #ifdef measurementMode
314 togglePinStatePortA(3); //go measure PA3
315 #endif
316 //uint32_t ThisCapture;
317 // start by clearing the source of the interrupt, the input capture event
318 HWREG(WTIMER0_BASE+TIMER_O_ICR) = TIMER_ICR_CAECINT;
319 //printf("In input capture ISR after clearing\r\n");
320 //start the one shot timer
321 StartOneShot_Encoder1();
322 //clear the Flag for One shot because now we are at a new edge
323 FlagOneShotTimeout_Encoder1=0;
324 // now grab the captured value and calculate the period
325 ThisCapture_Encoder1 = HWREG(WTIMER0_BASE+TIMER_O_TAR);
326 ThisPeriod_Encoder1 = ThisCapture_Encoder1 - LastCapture_Encoder1;
327 //printf("Period is %d\r\n", ThisPeriod_Encoder1);
328 // update LastCapture to prepare for the next edge
329 LastCapture_Encoder1 = ThisCapture_Encoder1;
330 encoder1_count ++; //increase the encoder_count from the detected pulse
331 rpm_1 = RPM_MULTIPLIER/ThisPeriod_Encoder1;
332 }
333
334 /* Return rpm of motor 1 */
335 double get_RPM_Encoder1(void){
336 return rpm_1;
337 }
338
339 /* Return rpm of motor 2 */
340 double get_RPM_Encoder2(void){
341 return rpm_2;
342 }
343
344 void InitOneShotInt_Encoder1( void ){
345 // start by enabling the clock to the timer (Wide Timer 0)
346 HWREG(SYSCTL_RCGCWTIMER) |= SYSCTL_RCGCWTIMER_R0;
347 // kill a few cycles to let the clock get going
348 while((HWREG(SYSCTL_PRWTIMER) & SYSCTL_PRWTIMER_R0) != SYSCTL_PRWTIMER_R0)
349 {
350 }
351 // make sure that timer (Timer B) is disabled before configuring
352 HWREG(WTIMER0_BASE+TIMER_O_CTL) &= ~TIMER_CTL_TBEN; //TBEN = Bit8
353 // set it up in 32bit wide (individual, not concatenated) mode
354 // the constant name derives from the 16/32 bit timer, but this is a 32/64
355 // bit timer so we are setting the 32bit mode
356 HWREG(WTIMER0_BASE+TIMER_O_CFG) = TIMER_CFG_16_BIT; //bits 0-2 = 0x04
357 // set up timer B in 1-shot mode so that it disables timer on timeouts
358 // first mask off the TAMR field (bits 0:1) then set the value for
359 // 1-shot mode = 0x01
360 HWREG(WTIMER0_BASE+TIMER_O_TBMR) =

```

```

361     (HWREG(WTIMER0_BASE+TIMER_O_TBMR) & ~TIMER_TBMR_TBMR_M) |
362     TIMER_TBMR_TBMR_1_SHOT;
363     // set timeout
364     HWREG(WTIMER0_BASE+TIMER_O_TBILR) = OneShotTimeout;
365     // enable a local timeout interrupt. TBTOIM = bit 8
366     HWREG(WTIMER0_BASE+TIMER_O_IMR) |= TIMER_IMR_TBTOIM; // 8
367     // enable the Timer B in Wide Timer 0 interrupt in the NVIC
368     // it is interrupt number 95 so appears in EN2 at bit 30
369     HWREG(NVIC_EN2) |= BIT31HI;
370     // make sure interrupts are enabled globally
371     __enable_irq();
372     //StartTime = ES_Timer_GetTime();
373     // now kick the timer off by enabling it and enabling the timer to
374     // stall while stopped by the debugger. TAEN = Bit0, TASTALL = bit1
375     HWREG(WTIMER0_BASE+TIMER_O_CTL) |= (TIMER_CTL_TBEN | TIMER_CTL_TBSTALL);
376 }
377
378 void StartOneShot_Encoder1( void ){
379     // start by grabbing the start time
380     //StartTime = ES_Timer_GetTime();
381     // now kick the timer off by enabling it and enabling the timer to
382     // stall while stopped by the debugger
383     HWREG(WTIMER0_BASE+TIMER_O_CTL) |= (TIMER_CTL_TBEN | TIMER_CTL_TBSTALL);
384 }
385
386 void OneShotIntResponse_Encoder1( void ){
387     // start by clearing the source of the interrupt
388     HWREG(WTIMER0_BASE+TIMER_O_ICR) = TIMER_ICR_TBTOCINT;
389     //printf("In one shot ISR after clearing\n\r");
390     FlagOneShotTimeout_Encoder1=1;
391 }
392
393
394 void InitInputCapture_Encoder2( void ){
395     // start by enabling the clock to the timer (Wide Timer 1)
396     HWREG(SYSCTL_RCGCWTIMER) |= SYSCTL_RCGCWTIMER_R1;
397     // enable the clock to Port C
398     HWREG(SYSCTL_RCGCGPIO) |= SYSCTL_RCGCGPIO_R2;
399     // since we added this Port C clock init, we can immediately start
400     // into configuring the timer, no need for further delay
401     // make sure that timer (Timer A) is disabled before configuring
402     HWREG(WTIMER1_BASE+TIMER_O_CTL) &= ~TIMER_CTL_TAEN;
403     // set it up in 32bit wide (individual, not concatenated) mode
404     // the constant name derives from the 16/32 bit timer, but this is a 32/64
405     // bit timer so we are setting the 32bit mode
406     HWREG(WTIMER1_BASE+TIMER_O_CFG) = TIMER_CFG_16_BIT;
407     // we want to use the full 32 bit count, so initialize the Interval Load
408     // register to 0xffff.ffff (its default value :-))
409     HWREG(WTIMER1_BASE+TIMER_O_TAILR) = 0xffffffff;
410     // set up timer A in capture mode (TAMR=3, TAAMS = 0),
411     // for edge time (TACMR = 1) and up-counting (TACDIR = 1)
412     HWREG(WTIMER1_BASE+TIMER_O_TAMR) =
413     (HWREG(WTIMER1_BASE+TIMER_O_TAMR) & ~TIMER_TAMR_TAAMS) |
414     (TIMER_TAMR_TACDIR | TIMER_TAMR_TACMR | TIMER_TAMR_TAMR_CAP);
415     // To set the event to rising edge, we need to modify the TAEVENT bits
416     // in GPTMCTL. Rising edge = 00, so we clear the TAEVENT bits
417     HWREG(WTIMER1_BASE+TIMER_O_CTL) &= ~TIMER_CTL_TAEVENT_M;
418     // Now Set up the port to do the capture (clock was enabled earlier)
419     // start by setting the alternate function for Port C bit 6 (WT1CCP0)
420     HWREG(GPIO_PORTC_BASE+GPIO_O_AFSEL) |= BIT6HI;
421     // Then, map bit 4's alternate function to WT0CCP0
422     // 7 is the mux value to select WT0CCP0, 16 to shift it over to the
423     // right nibble for bit 6 (4 bits/nibble * 6 bits)
424     HWREG(GPIO_PORTC_BASE+GPIO_O_PCTL) =
425     (HWREG(GPIO_PORTC_BASE+GPIO_O_PCTL) & 0xf0ffffff) + (7<<24);
426     // Enable pin on Port C for digital I/O
427     HWREG(GPIO_PORTC_BASE+GPIO_O_DEN) |= BIT6HI;
428     // make pin 4 on Port C into an input
429     HWREG(GPIO_PORTC_BASE+GPIO_O_DIR) &= BIT6LO;
430     // back to the timer to enable a local capture interrupt
431     HWREG(WTIMER1_BASE+TIMER_O_IMR) |= TIMER_IMR_CAEIM;
432     // enable the Timer A in Wide Timer 0 interrupt in the NVIC

```

```

433 // it is interrupt number 94 so appears in EN2 at bit 30
434 HWREG(NVIC_EN3) |= BIT0HI;
435 // make sure interrupts are enabled globally
436 __enable_irq();
437 // now kick the timer off by enabling it and enabling the timer to
438 // stall while stopped by the debugger
439 HWREG(WTIMER1_BASE+TIMER_O_CTL) |= (TIMER_CTL_TAEN | TIMER_CTL_TASTALL);
440 }
441
442
443 void InputCaptureResponse_Encoder2( void ){
444 //printf("In input capture2\r\n");
445 #ifdef measurementMode
446 togglePinStatePortA(3); //go measure PA3
447 #endif
448 //uint32_t ThisCapture;
449 // start by clearing the source of the interrupt, the input capture event
450 HWREG(WTIMER1_BASE+TIMER_O_ICR) = TIMER_ICR_CAECINT;
451 //printf("In input capture ISR after clearing\n\r");
452 //start the one shot timer
453 StartOneShot_Encoder2();
454 //clear the Flag for One shot because now we are at a new edge
455 FlagOneShotTimeout_Encoder2=0;
456 // now grab the captured value and calculate the period
457 ThisCapture_Encoder2 = HWREG(WTIMER1_BASE+TIMER_O_TAR);
458 ThisPeriod_Encoder2 = ThisCapture_Encoder2 - LastCapture_Encoder2;
459 // update LastCapture to prepare for the next edge
460 LastCapture_Encoder2 = ThisCapture_Encoder2;
461 encoder2_count ++; //increase the encoder_count from the detected pulse
462 rpm_2 = RPM_MULTIPLIER/ThisPeriod_Encoder2;
463 }
464
465 /* Return encoder value of motor 1 */
466 uint32_t get_encoder1_count(void){
467     return encoder1_count;
468 }
469
470 /* Return encoder value of motor 2 */
471 uint32_t get_encoder2_count(void){
472     return encoder2_count;
473 }
474
475 void InitOneShotInt_Encoder2( void ){
476 // start by enabling the clock to the timer (Wide Timer 1)
477 HWREG(SYSCTL_RCGCTIMER) |= SYSCTL_RCGCTIMER_R1;
478 // kill a few cycles to let the clock get going
479 while((HWREG(SYSCTL_PRWTIMER) & SYSCTL_PRWTIMER_R1) != SYSCTL_PRWTIMER_R1)
480 {
481 }
482 // make sure that timer (Timer B) is disabled before configuring
483 HWREG(WTIMER1_BASE+TIMER_O_CTL) &= ~TIMER_CTL_TBEN; //TBEN = Bit8
484 // set it up in 32bit wide (individual, not concatenated) mode
485 // the constant name derives from the 16/32 bit timer, but this is a 32/64
486 // bit timer so we are setting the 32bit mode
487 HWREG(WTIMER1_BASE+TIMER_O_CFG) = TIMER_CFG_16_BIT; //bits 0-2 = 0x04
488 // set up timer B in 1-shot mode so that it disables timer on timeouts
489 // first mask off the TAMR field (bits 0:1) then set the value for
490 // 1-shot mode = 0x01
491 HWREG(WTIMER1_BASE+TIMER_O_TBMR) =
492 (HWREG(WTIMER1_BASE+TIMER_O_TBMR) & ~TIMER_TBMR_TBMR_M) |
493 TIMER_TBMR_TBMR_1_SHOT;
494 // set timeout
495 HWREG(WTIMER1_BASE+TIMER_O_TBILR) = OneShotTimeout;
496 // enable a local timeout interrupt. TBTOIM = bit 8
497 HWREG(WTIMER1_BASE+TIMER_O_IMR) |= TIMER_IMR_TBTOIM; // 8
498 // enable the Timer B in Wide Timer 0 interrupt in the NVIC
499 // it is interrupt number 97 so appears in EN3 at bit 1
500 HWREG(NVIC_EN3) |= BIT1HI;
501 // make sure interrupts are enabled globally
502 __enable_irq();
503 //StartTime = ES_Timer_GetTime();
504 // now kick the timer off by enabling it and enabling the timer to

```



```

505 // stall while stopped by the debugger. TAEN = Bit0, TASTALL = bit1
506 HWREG(WTIMER1_BASE+TIMER_O_CTL) |= (TIMER_CTL_TBEN | TIMER_CTL_TBSTALL);
507 }
508
509 void StartOneShot_Encoder2( void ){
510 // start by grabbing the start time
511 //StartTime = ES_Timer_GetTime();
512 // now kick the timer off by enabling it and enabling the timer to
513 // stall while stopped by the debugger
514 HWREG(WTIMER1_BASE+TIMER_O_CTL) |= (TIMER_CTL_TBEN | TIMER_CTL_TBSTALL);
515 }
516
517 void OneShotIntResponse_Encoder2( void ){
518 // start by clearing the source of the interrupt
519 HWREG(WTIMER1_BASE+TIMER_O_ICR) = TIMER_ICR_TBTOCINT;
520 //printf("In one shot ISR after clearing\n\r");
521 FlagOneShotTimeout_Encoder2 = 1;
522 }
523
524 void InitPeriodicInt( void ){
525 // start by enabling the clock to the timer (Wide Timer 3)
526 HWREG(SYSCTL_RCGCWTIMER) |= SYSCTL_RCGCWTIMER_R3; // kill a few cycles to let the clock get going
527 while((HWREG(SYSCTL_PRWTIMER) & SYSCTL_PRWTIMER_R3) != SYSCTL_PRWTIMER_R3){}
528 // make sure that timer (Timer A) is disabled before configuring
529 HWREG(WTIMER3_BASE+TIMER_O_CTL) &= ~TIMER_CTL_TAEN;
530 // set it up in 32bit wide (individual, not concatenated) mode
531 HWREG(WTIMER3_BASE+TIMER_O_CFG) = TIMER_CFG_16_BIT;
532 // set up timer A in periodic mode so that it repeats the time-outs
533 HWREG(WTIMER3_BASE+TIMER_O_TAMR) =
534 (HWREG(WTIMER3_BASE+TIMER_O_TAMR) & ~TIMER_TAMR_TAMR_M) |
535 TIMER_TAMR_TAMR_PERIOD;
536 // set timeout to 2mS
537 HWREG(WTIMER3_BASE+TIMER_O_TAILR) = PeriodicTimeout;
538 // enable a local timeout interrupt
539 HWREG(WTIMER3_BASE+TIMER_O_IMR) |= TIMER_IMR_TATOIM;
540 // enable the Timer A in Wide Timer 0 interrupt in the NVIC
541 // it is interrupt number 100 so appears in EN3 at bit 4
542 HWREG(NVIC_EN3) = BIT4HI;
543 HWREG(NVIC_PRI25) = ((HWREG(NVIC_PRI25)) & ~NVIC_PRI25_INTA_M) | NVIC_PRI25_INTA_M;
544 // make sure interrupts are enabled globally
545 __enable_irq();
546 // now kick the timer off by enabling it and enabling the timer to
547 // stall while stopped by the debugger
548 HWREG(WTIMER3_BASE+TIMER_O_CTL) |= (TIMER_CTL_TAEN | TIMER_CTL_TASTALL);
549 }
550
551 /* This periodic response is where our control happens */
552 void PeriodicIntResponse( void ){
553 static float IntegralTerm=0.0;
554 /* integrator control effort */
555 static float RPMEError;
556 /* make static for speed */
557 static float LastError;
558 /* for Derivative Control */
559 //static uint32_t rpm_EN1;
560 /* make static for speed */
561 // start by clearing the source of the interrupt HWREG(WTIMER0_BASE+TIMER_O_ICR) = TIMER_ICR_TATOCINT;
562 HWREG(WTIMER3_BASE+TIMER_O_ICR) = TIMER_ICR_TATOCINT;
563 #ifdef RPM_TRACKING
564 float RPM_1 = rpm_1;
565 float RPM_2 = rpm_2;
566 RPMEError = RPM_1 - RPM_2;
567 IntegralTerm += iGain * RPMEError;
568 //IntegralTerm = clamp(IntegralTerm, 30, MAX_DUTY);
569 /* anti-windup */
570 RequestedDuty = (pGain * ((RPMEError)+IntegralTerm+((-1)*dGain * (RPMEError-LastError))));
571 RequestedDuty = clamp(RequestedDuty, MIN_DUTY, MAX_DUTY);
572
573 //anti-wind up
574 if (RequestedDuty == MAX_DUTY){
575 IntegralTerm=IntegralTerm-iGain*RPMEError;
576 }

```



```

577     if (RequestedDuty == MIN_DUTY){
578         IntegralTerm=IntegralTerm+iGain*RPMEError;
579     }
580
581     // if we don't include control in our drive
582     #ifdef OPEN_LOOP
583     RequestedDuty = FULL_DUTY; //always set requested duty to full duty
584     #endif
585
586     LastError = RPMEError; // update last error
587
588     //check duty mode to make sure the motor is in running mode
589     if(duty_mode != -1){
590         if(duty_mode == 0 | duty_mode == 2){
591             OwnPWM_SetDutyB(MAX_DUTY-RequestedDuty);
592         }
593         else{
594             OwnPWM_SetDutyB((uint32_t) (RequestedDuty));
595         }
596     }
597     #endif
598
599     // in case we want to track encoder count instead
600     #ifdef ENCODER_TRACKING
601     int ENCODER1 = encoder1_count;
602     int ENCODER2 = encoder2_count;
603     RPMEError = ENCODER1 - ENCODER2;
604     IntegralTerm += iGain * RPMEError;
605     IntegralTerm = clamp(IntegralTerm, 0, MAX_DUTY);
606     /* anti-windup */
607     RequestedDuty =(pGain * ((RPMEError)+IntegralTerm+(dGain * (RPMEError-LastError))));
608     RequestedDuty = clamp(RequestedDuty, 0, MAX_DUTY);
609     LastError = RPMEError; // update last error
610     #ifdef OPEN_LOOP
611     RequestedDuty = FULL_DUTY;
612     #endif
613     if(duty_mode != -1){
614         if(duty_mode == 0 | duty_mode == 3){
615             OwnPWM_SetDutyB(MAX_DUTY-RequestedDuty);
616         }
617         else{
618             OwnPWM_SetDutyB((uint32_t) (RequestedDuty));
619         }
620     }
621     #endif
622 }
623
624 /* Return requested duty value which is calculated in control loop */
625 uint8_t get_Duty(void){
626     return RequestedDuty;
627 }
628
629 // constrain val to be in range clampL to clampH
630 float clamp(float val, float clampL, float clampH){
631     if (val > clampH) { // if too high
632         return clampH;
633     }
634     if (val < clampL) { // if too low
635         return clampL;
636     }
637     return val; // if OK as-is
638 }
639
640 void updateLocation_from_Ultrasonic(void){
641     prev_location_x = location_x;
642     prev_location_y = location_y;
643     location_x = get_Ultrasonic_X();
644     uint32_t y_f = get_Ultrasonic_Y_F(); // get value for y front
645     uint32_t y_b = get_Ultrasonic_Y_B(); // get value for y back
646
647     // if using two sensors for y axis
648     #ifdef USING_FRONT_BACK_ULTRASONIC

```

```

649 // the idea here is that we have two ultrasonic reading: one is in the front
650 // of the robot and the other one is in the back. The assumption here is that
651 // the closer one is to the wall, the more accurate the reading will be.
652 // thus, when we have to reading, we will choose the get the value from the
653 // reading that is lower (closer to the wall). We have to adjust the reading
654 // accordingly if we use the one in the back.
655 if( y_f <= y_b){
656     location_y = y_f;
657 }
658 else{
659     // when using back reading, we have to subtract it from the offset
660     // to ge a uniform value with the one we get from the front reading
661     location_y = Y_AXIS_OFFSET - y_b;
662 }
663 #endif
664
665 // if using only one sensor in the front for y axis
666 #ifdef NOT_USING_FRONT_BACK_ULTRASONIC
667 location_y = y_f; // NOW WE USE THE LOCATION FROM THE FRONT
668 #endif
669
670 // here, we validate the reading value, if more than maximum limit, we reject it
671 if(location_x > MAX_X){
672     location_x = prev_location_x;
673 }
674 if(location_y >MAX_Y){
675     printf("location_y more than max, reject this y\r\n");
676     location_y = prev_location_y;
677 }
678
679 // here, we reject value that is too different from the reading before this
680 // we assume the robot cannot move too far during the inveral of this reading
681 if(prev_location_x != 0){
682     if (location_x > prev_location_x){
683         if(location_x - prev_location_x > REJECTION_TOLERANCE){
684             location_x = prev_location_x;
685         }
686     }
687     else if(prev_location_x > location_x){
688         if(prev_location_x - location_x > REJECTION_TOLERANCE){
689             location_x = prev_location_x;
690         }
691     }
692 }
693 }
694
695 /* Return current location values */
696 uint32_t get_current_X(void){
697     return location_x;
698 }
699
700 uint32_t get_current_Y(void){
701     return location_y;
702 }
703
704 /* Set this flag to use location checker and vice versa*/
705 void set_location_checker_flag(bool input){
706     use_location_checker = input;
707 }
708
709 /* This function check if the destination is reached */
710 void Location_Checker( void){
711     updateLocation_from_Ultrasonic(); // first update the current readings
712     if(use_location_checker){
713         if(move_x_flag){ //check location only when we are moving (x or y)
714             if((target_x <= location_x + LOCATION_TOLERANCE) && (target_x >= location_x - LOCATION_TOLERANCE)){
715                 // if the location we are and the target is at the same place,
716                 // post REACH event to main
717                 stopMotor();
718                 ES_Event to_post1;
719                 to_post1.EventType = X_REACHED;
720                 move_x_flag = false; //when we reach destination, clear flag

```

```

721         move_y_flag = false;
722         PostMasterVehicleSM(to_post1);
723     }
724     else{
725         // do nothing, don't post anything
726     }
727 }
728 else if(move_y_flag){
729     //if the location we are and the target is at the same place,
730     // post REACH event to main
731     if((target_y <= location_y + LOCATION_TOLERANCE) && (target_y >= location_y - LOCATION_TOLERANCE)){
732         stopMotor();
733         ES_Event to_post2;
734         to_post2.EventType = Y_REACHED;
735         move_y_flag = false; //when we reach destination, clear flag
736         move_x_flag = false;
737         PostMasterVehicleSM(to_post2);
738     }
739     else{
740         // do nothing, don't post anything
741     }
742 }
743 }
744 // for this special flag, when the robot is arriving at the wall, we reduce the speed to avoid hard
collision
745 if(going_to_supply_flag){
746     if(location_y < WALL_BREAK_DISTANCE){
747         set_PWM_Full_Duty(CORRECTION_FULL_DUTY);
748         going_to_supply_flag = false;
749     }
750 }
751 }
752
753 /* set going to supply flag */
754 void set_going_to_supply_flag(void){
755     going_to_supply_flag = true;
756 }
757
758 /* clear going to supply flag */
759 void clear_going_to_supply_flag(void){
760     going_to_supply_flag = false;
761 }
762
763 /* functions for checking whether target location is reached or not */
764 bool verify_x_location(void){
765     printf("Running verify X location\r\n");
766     if((target_x <= location_x + LOCATION_TOLERANCE) &&
767        (target_x >= location_x - LOCATION_TOLERANCE)){
768         return true;
769     }
770     else{
771         return false;
772     }
773 }
774
775 bool verify_y_location(void){
776     printf("Running verify Y location\r\n");
777     if((target_y <= location_y + LOCATION_TOLERANCE) &&
778        (target_y >= location_y - LOCATION_TOLERANCE)){
779         return true;
780     }
781     else{
782         return false;
783     }
784 }
785

```