Location pseudo code:

Location.c

Purpose:

The location service keeps track of the robot's location, and also handles motor driving. It interacts with the ultrasonic service to obtain the robot's position. It handles encoder interrupts to compute the robot's speed, and apply PID. The way the PID is set up, one motor tracks the speed of the other one.

void Init_Location(void)

Initialize period interrupt

Initialize PWM subsystem

Set the PWM frequency

Initialize the PWM ports

Stop the motors

Initialize the input capture for encoder 1

Initialize a one shot timer for encoder 1

Initialize the input capture for encoder 2

Initialize a one shot timer for encoder 2

Set the current FSM state as STOP

void set_PWM_Full_Duty(uint32_t input)

Set PWM input to full duty

uint32_t get_X(void)

return the current X location

uint32_t get_Y(void)

return the current Y location

void move_X(uint32_t target_X)

update the target X location to the target X passed to the function

set encoder 1 count to 0

set encoder 2 count to 0

set move X flag to true

set move Y flag to false

compare the current X location and the target X location, and determine the movement direction

run the motors to east or west accordingly

void move_Y(uint32_t target_Y)

update the target Y location to the target Y passed to the function

set encoder 1 count to 0

set encoder 2 count to 0

set move X flag to false

set move Y flag to true

compare the current Y location and the target Y location, and determine the movement direction

run the motors to north or south accordingly

LocationState_t QueryLocationState (void)

Return the current state

void Init_PWM_port(void)

Enable port B

Set PB2 as output

Set PB3 as output

void stopMotor (void)

Set move X flag to falsae

Set move Y flag to false

Set current FSM state as stop

Set the duty cycle mode to stop

Write Stop Duty Cycle to Motor A

Write Stop Duty Cycle to Motor B

Write low to PB2

Write low to PB3

void runMotor (uint8_t mode)

If Drive Mode == Drive North:

Set duty cycle mode to 0

Set current FSM state to DRIVE_NORTH

Write max duty cycle – full duty cycle to Motor A

Write max duty cycle – full duty cycle to Motor B

Write high to PB2

Write high to PB3

Else If Drive Mode == Drive South:

Set duty cycle mode to 1

Set current FSM state to DRIVE_SOUTH

Write full duty cycle to Motor A

Write full duty cycle to Motor B

Write low to PB2

Write low to PB3

Else If Drive Mode == Drive East:

Set duty cycle mode to 2

Set current FSM state to DRIVE_EAST

Write full duty cycle to Motor A

Write max duty - full duty cycle to Motor B

Write low to PB2

Write high to PB3

Else If Drive Mode == Drive West:

Set duty cycle mode to 3

Set current FSM state to DRIVE_WEST

Write max duty - full duty cycle to Motor A

Write full duty cycle to Motor B

Write high to PB2

Write low to PB3

Else

Print invalid command error

void InitInputCapture_Encoder1(void)

start by enabling the clock to the timer (Wide Timer 0)

enable the clock to Port C

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

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

we want to use the full 32 bit count, so initialize the Interval Load

register to 0xffff.ffff (its default value :-)

set up timer A in capture mode (TAMR=3, TAAMS = 0),

for edge time (TACMR = 1) and up-counting (TACDIR = 1)

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

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)

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)

Enable pin on Port C for digital I/O

make pin 4 on Port C into an input

back to the timer to enable a local capture interrupt

enable the Timer A in Wide Timer 0 interrupt in the NVIC

it is interrupt number 94 so appears in EN2 at bit 30

make sure interrupts are enabled globally

now kick the timer off by enabling it and enabling the timer to

stall while stopped by the debugger

void InputCaptureResponse_Encoder1(void)

start by clearing the source of the interrupt, the input capture event
start the one shot timer

clear the Flag for One shot because now we are at a new edge

now grab the captured value and calculate the period

update LastCapture to prepare for the next edge

increase the encoder_count from the detected pulse

compute the rpm

double get_RPM_Encoder1(void)

return RPM of motor A

double get_RPM_Encoder2(void)

return RPM of motor B

void InitOneShotInt_Encoder1(void)

start by enabling the clock to the timer (Wide Timer 0)

kill a few cycles to let the clock get going

make sure that timer (Timer B) is disabled before configuring

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

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

set timeout

enable a local timeout interrupt. TBTOIM = bit 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

make sure interrupts are enabled globally

now kick the timer off by enabling it and enabling the timer to

stall while stopped by the debugger. TAEN = Bit0, TASTALL = bit1

void StartOneShot_Encoder1(void)

start by grabbing the start time

now kick the timer off by enabling it and enabling the timer to

stall while stopped by the debugger

void OneShotIntResponse_Encoder1(void)

start by clearing the source of the interrupt

set encoder 1 one shot timeout flag to high

void InitInputCapture_Encoder2(void)

start by enabling the clock to the timer (Wide Timer 1)

enable the clock to Port C

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

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

we want to use the full 32 bit count, so initialize the Interval Load

register to 0xffff.ffff (its default value :-)

set up timer A in capture mode (TAMR=3, TAAMS = 0),

for edge time (TACMR = 1) and up-counting (TACDIR = 1)

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

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)

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)

Enable pin on Port C for digital I/O

make pin 4 on Port C into an input

back to the timer to enable a local capture interrupt

enable the Timer A in Wide Timer 0 interrupt in the NVIC

it is interrupt number 94 so appears in EN2 at bit 30

make sure interrupts are enabled globally

now kick the timer off by enabling it and enabling the timer to

stall while stopped by the debugger

void InputCaptureResponse_Encoder2(void)

clear the source of interrupt

start the one shot timer for encoder 2

since we got a new encoder edge, clear the timer for the encoder 2 one shot

compute the encoder time period from the new edge

update the last encoder capture variable

increase the encoder count

update the rpm

uint32_t get_encoder1_count(void)

return the encoder count for motor 1

uint32_t get_encoder1_count(void)

return the encoder count for motor 2

void InitOneShotInt_Encoder2(void)

start by enabling the clock to the timer (Wide Timer 1)

kill a few cycles to let the clock get going

make sure that timer (Timer B) is disabled before configuring

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

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

set timeout

enable a local timeout interrupt. TBTOIM = bit 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

make sure interrupts are enabled globally

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

void StartOneShot_Encoder2(void)

start by grabbing the start time

now kick the timer off by enabling it and enabling the timer to

stall while stopped by the debugger

void OneShotIntResponse_Encoder2(void)

clear the source of the interrupt

raise the encoder 2 oneshot timer timeout flag high

void InitPeriodicInt(void)

start by enabling the clock to the timer (Wide Timer 3)

kill a few cycles to let the clock get going

make sure that timer (Timer A) is disabled before configuring

set it up in 32bit wide (individual, not concatenated) mode

set up timer A in periodic mode so that it repeats the time-outs

set timeout to 2mS

enable a local timeout interrupt

enable the Timer A in Wide Timer 0 interrupt in the NVIC

it is interrupt number 100 so appears in EN3 at bit 4

make sure interrupts are enabled globally

now kick the timer off by enabling it and enabling the timer to

stall while stopped by the debugger

// this function performs PID control

void PeriodicIntResponse(void)

clear the source of the interrupt

define the variables for the integral term, RPM error and the last error

set the integral term to zero

ifdef -> RPM tracking mode (i.e. one motor's RPM is tracking the other)

create a local RPM variable for RPM of motor 1 (RPM_1) and assign its value as the current rpm reading

create a local RPM variable for RPM of motor 2 (RPM_2) and assign its value as the current rpm reading

RPM error is RPM_1 – RPM_2

Integral term = integral gain * rpm error

Requested duty = p gain * RPM error + integral term – differential gain * (rpm error – last error)

Clamp requested duty between the minimum duty and the maximum duty

Perform anti wind-up of the integral term

Ifdef open loop mode ->

Requested duty = full duty

Update the last rpm error

If the motors are running:

Write (either max-duty – requested duty) or (requested duty) to the tracking motor depending on its direction

uint8_t get_Duty(void)

return the requested duty cycle

float clamp(float val, float clampL, float clampH)

if value > clampH, return clampH

if value < clampL , return clampL

if the value is within the range, return the value

void updateLocation_from_Ultrasonic(void)

set previous x location as the current X location

set previous y location as the current Y location

set the current x location by getting an X reading from the ultrasonic service

set the current y front reading from the ultrasonic service

set the current y back reading from the ultrasonic service

ifdef -> using both back and the front y ultrasonic:

// 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 front reading < y back reading

Location y = y front reading

Else

Location y = y axis offset – y back

Ifdef -> using only one sensor

Location y = y front

If x reading > max x

//must have been a bad reading, ignore

Location x = previous location x

If location y > max y

//must have been a bad reading, ignore

Location y = previous location y

//code to ignore readings that are very different from the last reading (another indication of a //bad reading …)

If prev_location_x is not zero

if location x > previous location x

if the difference between the new and previous x location is bigger than the tolerated value:

location x = previous location x

else if previous location x > location x)

if(previous x location – x location > maximum tolerated difference)

location x = previous location x

uint32_t get_current_X(void)

return location x

uint32_t get_current_Y(void)

return location y

//function to set the flag to use the location checker

void set_location_checker_flag(bool input)

use location checker = input

//This function checks if the destination is reached

void Location_Checker(void)

get the current location reading from the ultrasonic sensors

if using the location checker:

if moving in x:

if the current location is within the tolerance bound to the target location

stop the motors

create an X_REACHED event

set move x and y flags to false

post the X_REACHED event to the master vehicle state machine

else, do nothing

if moving in y:

if the current location is within the tolerance bound to the target location

stop the motors

create an Y_REACHED event

set move x and y flags to false

post the Y_REACHED event to the master vehicle state machine

else, do nothing

if resupply flag is high:

if the y_location is less than the wall break distance

reduce the duty cycle applied to the motors

set going to supply flag false

void set_going_to_supply_flag(void)

going to supply flag is true

void clear_going_to_supply_flag(void)

going to supply flag is false

void clear_going_to_supply_flag(void)

if the x location is within the tolerance bound to the target x location

return true

else

return false

void verify_y_location(void)

if the y location is within the tolerance bound to the target y location

return true

else

return false