UNIVERSITY OF HOUSTON

Embedded Microcomputer Systems

Fall 2021

Professor

Han Le

**Team 22**

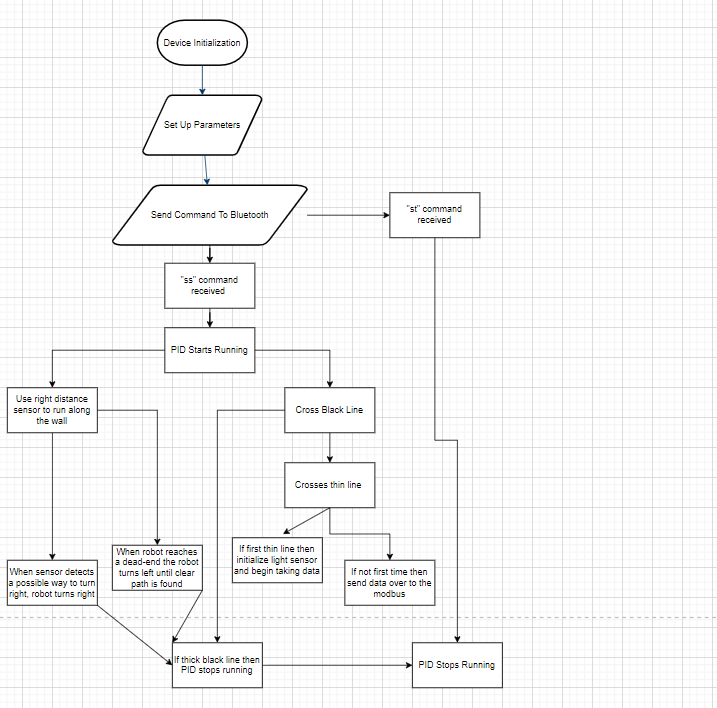
Damon Spencer Dat Tran

Nurul Saiym Christopher Montealvo

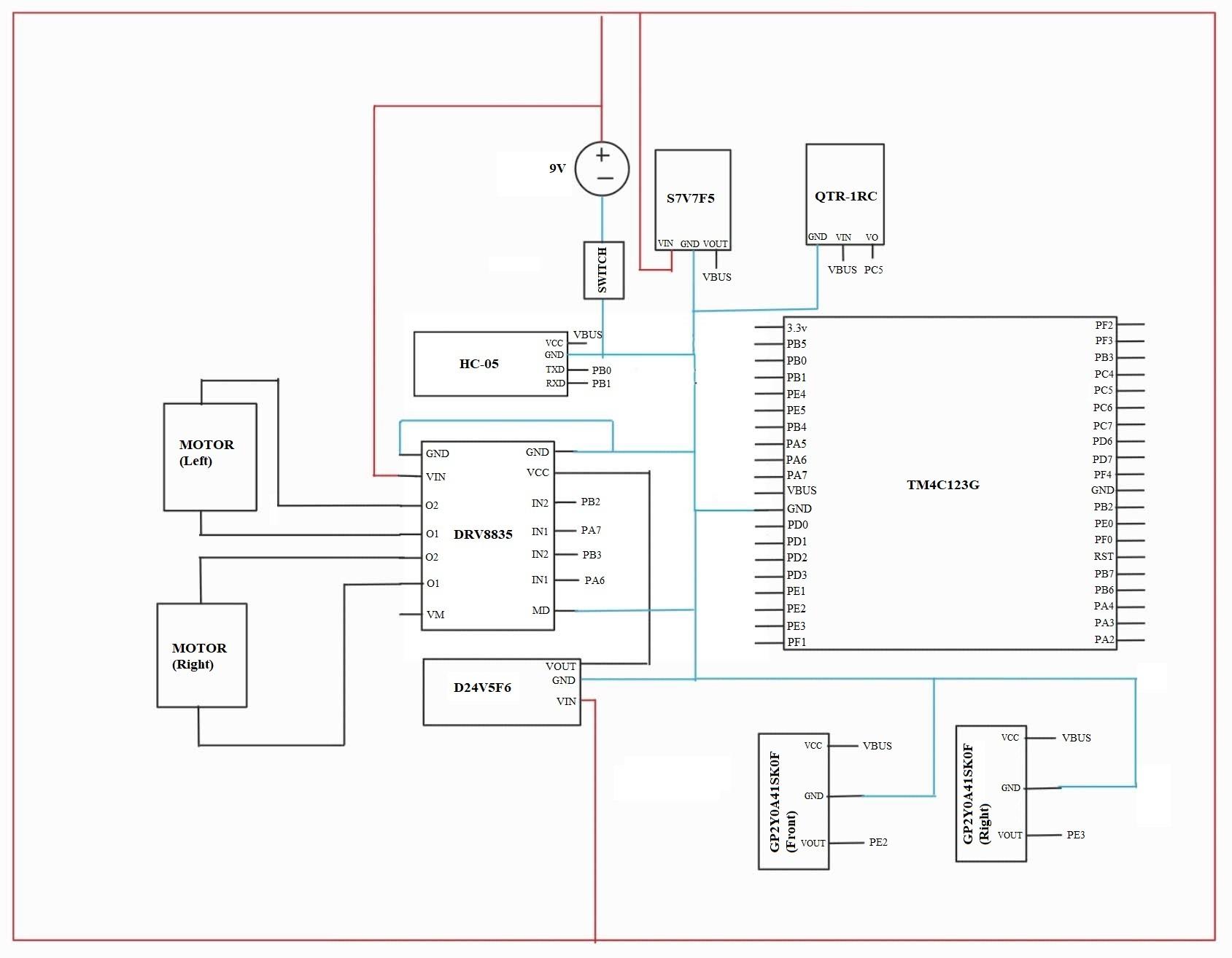
**Table of RTOS and functions required:**

| **Function** | **Filename** | **Functions / Notes** |
| --- | --- | --- |
| PID | RTOSPID.c | void RTOSPIDCompute() |
| Turns (Uturns and right turns) | RTOSPID.c | void RTOSPIDRightTurn()  bool RTOSPIDTurnAround() |
| Task | RTOSBluetooth.c  RTOSFlashing.c | void RTOSBTReadTaskFunc()  void RTOSBTWriteTaskFunc()  void RTOSFlashingTaskFunc() |
| HWI | Empty.cfg  Main.c  RTOSPID.c | Interrupt Number 51  Int main()  void RTOSPIDRun() |
| LED | RTOSFlashing.c  RTOSLightsensor.c | void RTOSFlashingSet(int flashingCode)  void RTOSFlashingSetColor(uint8\_t color)  void RTOSLightSensorTick() |
| Ping Pong buffers | RTOSModbus.c | bool RTOSModbusPushData(uint32\_t data) |
| Semaphores | RTOSBluetooth.c  RTOSFlashing.c | Semaphore\_pend(RTOSBT\_TXSem, BIOS\_WAIT\_FOREVER);  Semaphore\_pend(RTOSBT\_RXSem, BIOS\_WAIT\_FOREVER);  Semaphore\_pend(RTOSFlashingSem, BIOS\_WAIT\_FOREVER); |
| Modbus | RTOSModbus.c | void RTOSModbusPrint()  void RTOSModbusPrintExisting() |
| Clock | RTOSBluetooth.c  RTOSFlashing.c | void RTOSBluetooth\_ClockHandler()  void RTOSFlashing\_ClockHandler() |

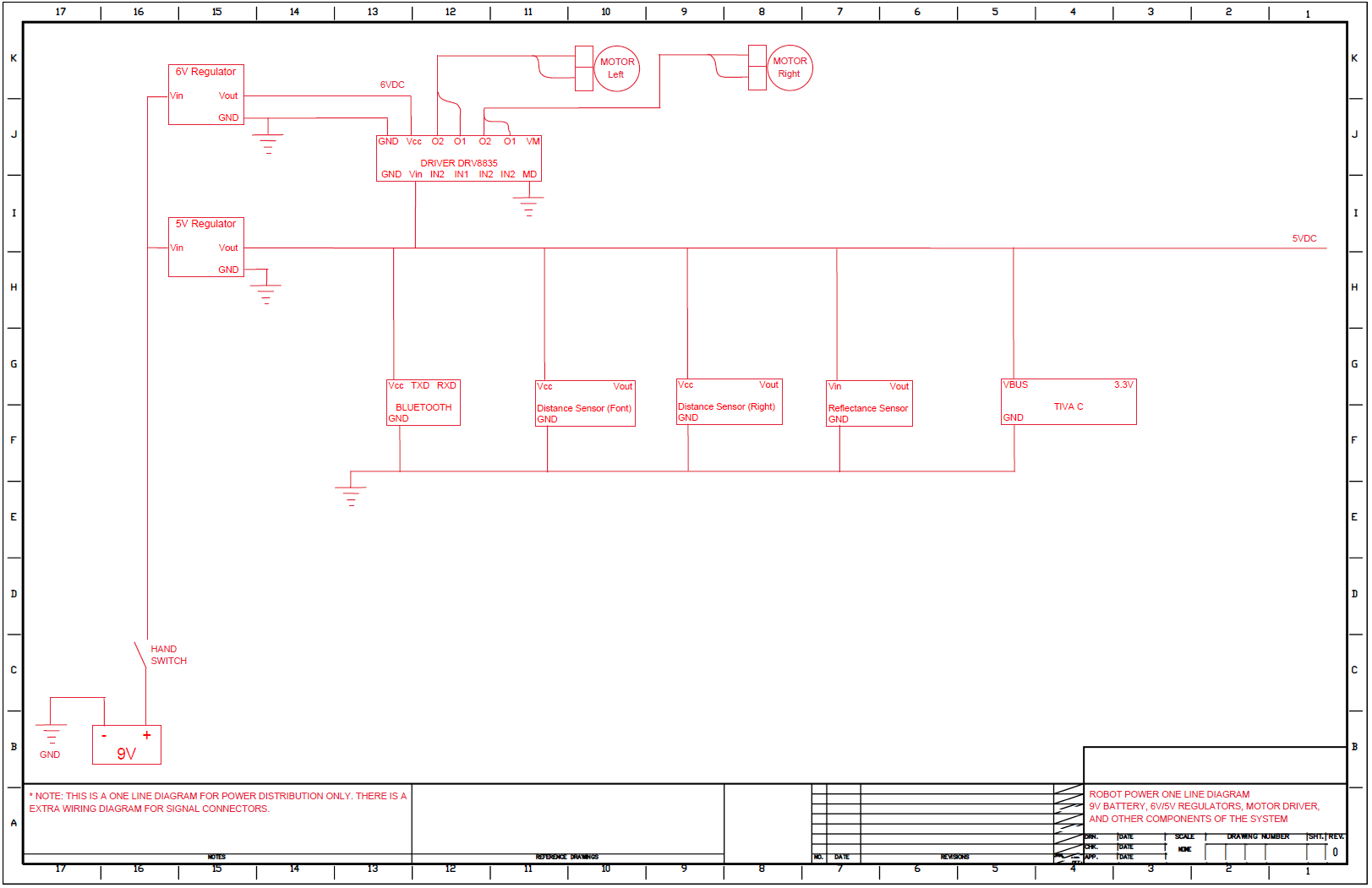
**Block Diagram:**



**Wiring Details:**

****

**Power Distribution One line Diagram:**

****

**Pseudo code:**

int main(void){

Timer 3 Hardware Interrupt Setup

BIOS Setting in empty.cfg: 80000000 = 80 MHz

enable Timer 3 periph clks

cfg Timer 3 mode - periodic

period = CPU clk div 2 (50ms)

set Timer 2 period

IntEnable(INT\_TIMER3A);

enable Timer 2 to interrupt CPU

enable Timer 2

call Board\_Init() then start the RTOS

}

void Board\_Init() {

Set CPU Clock to 40MHz. 400MHz PLL/2 = 200 DIV 5 = 40MHz

Call RTOSFlashingInit(), RTOSBTInit(), RTOSDistanceInit(), RTOSPIDInit(), RTOSMotorInit(),

RTOSLightSensorInit(), and RTOSModbusInit()

}

* Board\_Init():
  + Set the CPU Clock to 40 MHz.
  + Call RTOSFlashingInit(), RTOSBTInit(), RTOSDistanceInit(), RTOSPIDInit(), RTOSMotorInit(), RTOSLightSensorInit(), and RTOSModbusInit()

void RTOSBTInit() {

UART Init

Wait for the UART module to be ready

Configure task

Configure semaphore

Configure the UART Module

Create a UART with data processing off.

If UART fails to be set up (is null), Use RTOSFlashing to update status

}

void RTOSBTWriteTaskFunc(UArg arg0, UArg arg1) {

Initialize a pointer to a character c and integer count

While true:

Post semaphore

Set the pointer to a string

count = 0;

while the current character does not equal the null terminator and the current character is less than the string length:

UARTCharPut

Increment c and count

}

void RTOSWriteString(char\* cs) {

Set RTOSBT\_WriteString to cs

Post the semaphore RTOSBT\_TXSem

}

void RTOSBTReadTaskFunc(UArg arg0, UArg arg1) {

Make the rxBuffer, readSize, and temp

While true {

Release the RTOSBT\_RXSem, BIOS\_WAIT\_FOREVER semaphore

Get the read size

Rewrite rxBuffer to the UART

skip when no data

if c is not \n, append the command

otherwise, execute command

reset the string length

}

void RTOSBluetooth\_ClockHandler(UArg arg) {

Post the semaphore RTOSBT\_RXSem

}

void RTOSCommanderInit() {

Empty function

}

int RTOSRunCommand(char \*cmd) {

copy raw command

split command intto args

execute command:

if RTOSCommander\_token equals “d” {

get 1st argument

if RTOSCommander\_token equals NULL) {

Print error and return

}

if RTOSCommander\_token equals normal {

RTOSFlashingSet(0)

} else if RTOSCommander\_token equals error {

RTOSFlashingSet(1);

} else if RTOSCommander\_token equals bt {

RTOSFlashingSet(2);

} else if RTOSCommander\_token equals mr {

send command: debug motor\_right

get 2nd argument

} else if RTOSCommander\_token equals ml {

send command: debug motor\_left

get 2nd argument

}else if RTOSCommander\_token equals d {

Get the distance

}else if RTOSCommander\_token equals clk{

Get the system clock and send it to UART

}else if RTOSCommander\_token equals t {

Get the light timer count and send it to UART

}else if RTOSCommander\_token equals timer // get 2nd argument

Get the light timer for the corresponding sensor and send it to the UART

}

else {

Send error message to UART

}

}

// setting

else if RTOSCommander\_tokenequals pid {

// get 1st argument

if RTOSCommander\_token equals NULL {

Send error to UART

}

if RTOSCommander\_token equals get {

Get PID settings and print it to UART

} else if RTOSCommander\_token equals set {

get 2nd, 3rd, 4th values as PID

} else if RTOSCommander\_token equals setright {

get 2nd argument and set the right PID with it

} else if RTOSCommander\_token equals setfront {

get 2nd argument and set the front PID with it

} else if RTOSCommander\_token equals setpwmright {

get 2nd argument and set the right pwm with it

} else if RTOSCommander\_token equals setpwmleft{

get 2nd argument and set the right pwm with it

}

else if RTOSCommander\_token equals start {

Enable the pid

}

else if RTOSCommander\_token equals stop {

Disable the PID

}

else {

Print error message

}

}

else if RTOSCommander\_token equals t { // tune values in PID

get 1st argument

if RTOSCommander\_token equals NULL {

Print error to UART

}

if RTOSCommander\_token equals p+ {

RTOSPIDTuneP(1);

}

else if RTOSCommander\_token equals p- {

RTOSPIDTuneP(-1);

}

else if RTOSCommander\_token equals i+ {

RTOSPIDTuneI(1);

}

else if RTOSCommander\_token equals i- {

RTOSPIDTuneI(-1);

}

else if RTOSCommander\_token equals d+ {

RTOSPIDTuneD(1);

}

else if RTOSCommander\_token equals d- {

RTOSPIDTuneD(-1);

}

print PID values

}

else if RTOSCommander\_token equals ss {

RTOSPIDPIDEnable(1);

}

else if RTOSCommander\_token st {

RTOSPIDPIDEnable(0);

}

else {

invalid command, r = 0

}

Increment commandCount

return r

}

void RTOSDistanceInit(void) {

Setup ADC PE3 for front distance sensor - PE2 for right distance sensor

Enable ADC0 module

configure PE3 for input

Configure sample sequencer

}

void RTOSDistanceGet(uint32\_t\* front, uint32\_t\* right) {

RTOSDistanceRightTemp = 0;

RTOSDistanceFrontTemp = 0;

for (RTOSILoop = 0; RTOSILoop < 5; RTOSILoop++) {

clear ADC interrupt

trigger ADC sampling

read voltage

}

RTOSDistanceRightTemp = RTOSDistanceRightTemp / 5;

RTOSDistanceFrontTemp = RTOSDistanceFrontTemp / 5;

If distance right less than minimum of 4cm, \*right = 400

else if right greater than maximum of 40cm, \* right = 4000

else \*right = RTOSDistanceRightTemp;

if distance front less than minimum of 4cm

else if distance front is greater than maximum of 30cm, \*right = 3000

else distance front = RTOSDistanceFrontTemp;

}

uint32\_t VoltageToCmRight(uint32\_t voltage) {

Calibration to get the voltage equation to 10x[mm]

power factor 0.984 < 1 should cause error

replace with 1

}

uint32\_t VoltageToCmFront(uint32\_t voltage) {

Calibration to get the voltage equation to 10x[mm]

}

void RTOSFlashingInit() {

Config LED on board

Configure task

Configure semaphore

It's optional to store the handle, but we store it

}

void RTOSFlashingTaskFunc(UArg arg0, UArg arg1)

{

Set up Local variables. Variables here go onto the task stack!!

Run one-time code when task starts

Run loop forever (unless terminated)

{

Wait foe semaphore

Semaphore\_pend(RTOSFlashingSem, BIOS\_WAIT\_FOREVER);

Blink the led if it is not flashing

create the correct flashing codes

Write the flashing code to the LEDs

}

}

void RTOSFlashingSet(int flashingCode) {

Set isLEDFlashing to true

Set currentFlashingCode to flashingCode

}

int RTOSFlashingGet() {

return the currentFlashingCode

}

void RTOSFlashing\_ClockHandler(UArg arg) {

Post the semaphore RTOSFlashingSem

}

void RTOSFlashingSetColor(uint8\_t color) {

Set isLEDFlashing to false;

Set blinkingColor to color

}

void RTOSLightSensorInit() {

configure gpio

Configure timer for charging/discharging light sensor

Configure Timer for crossing line

Create a timer that is used for calculating how long it takes to cross a black line

}

void RTOSLightSensorTick() {

set timer to 0 to start timing

write 0xFF into pin C5

Wait 6000 clock cycles for it to finish charging

Run an idle loop 1000 times

Capture startTime

Make PortB pin1 input to time the decaying

Assign the value when sensor is in fully charged state

Loop to compare to the current state to fully-charged state

Capture endTime when fully decayed

if the robot crosses the white line

Note: LED values - 2=RED, 4=BLUE, 8=GREEN

RTOSFlashingSetColor(8);

if RTOSLightWhiteLine equals false {

stop the timer and store it

Capture startTime

print out when crossline

if RTOSLightSensorLineCount equals 1{

start timing

Capture startTime

} else {

cross the thin line, write remaining data

else if crossed thick line {

Output that the robot crossed the thick line

output the timing

choose 26 for delay of coding

} else {

Print invalid range of line

}

}

}

}

Else, the robot crosses the black line

Note: LED values - 2=RED, 4=BLUE, 8=GREEN

RTOSFlashingSetColor(4)

If RTOSLightWhiteLine equals true {

start timer

}

}

Increment RTOSLightTimeCount

}

void RTOSModbusInit() {

initialize dynamic array

store 20 8 bit numbers in hexadecimal

modbusID = 22

}

bool RTOSModbusPushData(uint32\_t data) {

flag equals false

if isPing equals true {

Add the data to the ping array

} else {

Add the data to the pong array

}

If modbusIndex is greater than or equal to modbusArrNum {

flag equals true

modbusIndex equals 0

isPing equals !isPing

}

return flag

}

void RTOSModbusPrint() {

get the string of mudbus format

start of frame ":", Team number (slave address), 20 bytes of data in ASCII, Team Number (checksum), CR, LF

store 20 bytes of data into modbusformat

print it out to input array

}

void RTOSModbusPrintExisting() {

swap ping vs ping in RTOSModbusPrint()

print it out to input array

}

void RTOSMotorDriveRight(int32\_t freq) {

if freq equals 0 {

Set the right pwm pulse width to 1 and the right output state to false

return

}

if freq is less than 0 {

Run the motor in reverse

note: in reverse mode, the lower value will be run higher

set freq to pwmMax + freq;

}

else {

Put the right motor in forward mode

}

Set the frequency

}

void RTOSMotorDriveLeft(int32\_t freq) {

if freq equals 0 {

Set the left pwm pulse width to 1 and the left output state to false

return

}

if freq is less than 0 {

Run the motor in reverse

note: in reverse mode, the lower value will be run higher

set freq to pwmMax + freq;

}

else {

Put the left motor in forward mode

}

Set the frequency, add 300 to the frequency due to balance issues

}

void RTOSMotorStop() {

Stop the motors

}

int RTOSMotorGetMaxPWM() {

return the pwmMax variable

}

void RTOSPIDInit(void){

Initialize Values of Settings

Set the min to 2000 and the max to 4500

RTOSPIDSetting[PIDPWMRight] = RTOSPIDSetting[PIDPWMMin];

RTOSPIDSetting[PIDPWMLeft] = RTOSPIDSetting[PIDPWMMin];

Set the offset to 800

Set p to 1, i to 1, and d to 3

Ki should be smallest, Kd and Ki is smaller than Kp

default robot state is stopped

}

void RTOSPIDGetSettings(char\* s) {

Print PID settings

}

void RTOSPIDSetSetting(int32\_t type, int32\_t value) {

Set RTOSPIDSetting at type to value

}

void RTOSPIDTuneP(int8\_t v) {

Add v to RTOSPIDSetting at PIDKp

}

void RTOSPIDTuneD(int8\_t v) {

Add v to RTOSPIDSetting at PIDKd

}

void RTOSPIDTuneI(int8\_t v) {

Add v to RTOSPIDSetting at PIDKi

}

void RTOSPIDGet(int\* p, int\* i, int\* d) {

Set \*p to RTOSPIDSetting at PIDKp

Set \*i to RTOSPIDSetting at PIDKi

Set \*d to RTOSPIDSetting at PIDKd

}

void RTOSPIDRunAlongWall() {

BEGIN PID CONTROL

allow error within 2mm or 20 [10xmm]

PIDError equals Right - Offset: > 0 when it is far away from wall

if (left wheel > right wheel) {

choose left wheel max, and right wheel decreasing

}

else if ( rightwheel > left wheel) {

PIDError < 0 when it is closed to the wall

choose right wheel max, adjust left wheel

}

END PID CONTROL

range is between PWM Min and PWM Max

}

void RTOSPIDRightTurn() {

Set RTOSPIDSetting at PIDPWMRight equal to 800

Set RTOSPIDSetting at PIDPWMLeft equal to 8500

}

// the robot keep turning left until find a path (u-turn or turn left)

// return true when the new path is found

bool RTOSPIDTurnAround() {

the robot keeps turning left until it finds a path (u-turn or turn left)

return true when the new path is found

perform turn left

slowly uturn until front greater than 2000 (20cm)

use d ml and d mr to find out the value for the motors

RTOSPIDSetting at PIDPWMRight equal to 2000

RTOSPIDSetting at PIDPWMLeft equal to -6000

if RTOSDistanceFront > 1500 and RTOSDistanceRight > 500 {

new path is found

return true

}

return false;

}

void RTOSPIDCompute() {

get distance sensor

PIDError < 0: temp\_dr < PIDRightOffset: when it is far away from the wall

PIDError > 0: temp\_dr > PIDRightOffset: when it is too closed from the wall

Temp\_p is negative when far away

temp\_i equals RTOSPIDSetting at PIDKi \* (PIDError + PIDPreError)

temp\_d equals RTOSPIDSetting at PIDKd \* (PIDError - PIDPreError)

PIDPWMAdjust equals temp\_p + temp\_i + temp\_d

send error to modbus

only send data when the first line is crossed

LED values - 2=RED, 4=BLUE, 8=GREEN

recalculate every 5ms

get error data each 100ms => 100 / 5 = 20

update Error

SET STATE OF ROBOT

switch (PIDState) {

case PIDSTATEALONGWALL:

The robot uses the right sensor to keep a constant distance to the wall

case PIDSTATETURNRIGHT:

when sensor detects a possible way to turn right, the robot will make a turn right

case PIDSTATETURNAROUND:

The robot keeps turning left to detect clear path

If new path is still not found, return to prevent the state change

}

change state

if RTOSDistanceFront < 1000 and RTOSDistanceRight < 1200 (10\*60m need to stop)

{

Stop the motor

Set PIDState to PIDSTATETURNAROUND

}

else if RTOSDistanceRight > 1200 {

change to right turn mode when right distance > 15cm

Set PIDState to PIDSTATETURNRIGHT

}

else {

Set PIDState to PIDSTATEALONGWALL

}

increase the loop count

}

void RTOSPIDPIDEnable(int8\_t flag) {

if flag equals 1 {

Enable the robot

} else {

stop the robot

}

}

void RTOSPIDRun() {

clear timer interrupt

if PIDState does not equal PIDSTATESTOPALL {

RTOSPIDCompute()

RTOSMotorDriveRight(RTOSPIDSetting at PIDPWMRight);

RTOSMotorDriveLeft(RTOSPIDSetting at PIDPWMLeft);

}

}

**Function Description:**

*Main.c functions*

* void Board\_init(): initializes board components for future use

*RTOSBluetooth Functions*

* void RTOSBTInit(): Bluetooth initialization along with UART initialization and using semaphores
* void RTOSBTWriteTaskFunc(UArg arg0, UArg arg1): writes commands to the bluetooth
* void RTOSBTReadTaskFunc(UArg arg0, UArg arg1): reads and executes bluetooth commands
* void RTOSWriteString(char\* cs): writing string and sending into semaphores

*RTOSCommander Functions*

* void RTOSCommanderInit(): for advance only, use queue interrupt to push commands
* int RTOSRunCommand(char \*cmd): runs various commands
* void Float2Str(char \*c, float f): converts variables from floats to strings

*RTOSDistance Functions*

* void RTOSDistanceInit(void): initializing Distance sensor using ADC0 module
* void RTOSDistanceGet(uint32\_t\* front, uint32\_t\* right): gets the distance from the right and front distance sensor
* uint32\_t VoltageToCmRight(uint32\_t voltage): calibrating right distance sensor
* uint32\_t VoltageToCmFront(uint32\_t voltage): calibrating front distance sensor

*RTOSFlashing Functions*

* void RTOSFlashingInit(): initializes LED on board
* void RTOSFlashingSet(int flashingCode): sets LED status
* int RTOSFlashingGet(): gets current LED status
* void RTOSFlashingTaskFunc(UArg arg0, UArg arg1): flashes LED depending on status
* void RTOSFlashing\_ClockHandler(UArg arg): handles clock semaphore
* void RTOSFlashingSetColor(uint8\_t color): sets LED color

*RTOSLightSensor Functions*

* void RTOSLightSensorInit(): initializes light sensor and configures timer
* void RTOSLightSensorTick(): collects light sensor data and determines which lines if any are crossed
* void RTOSLightSensorResetTick(): resets light counter tick
* uint32\_t RTOSLightSensorCount(): returns light sensor value
* uint32\_t RTOSLightSensorGet(int16\_t index): return indexed light sensor value
* int16\_t RTOSLightGetLightCount(): return number of lines crossed

*RTOSModbus Functions*

* void RTOSModbusInit(): initializes modbus
* void RTOSModbusSetID(uint16\_t tm): sets modbus ID
* void RTOSModbusPrint(): prints out modbus data
* bool RTOSModbusPushData(uint32\_t data): pushes data to ping or pong depending isPing and returns true if the index is reach to max array
* void RTOSModbusPrintExisting(): print existing value, use to print remaining values when robot is stopped

*RTOSMotor Functions*

* void RTOSMotorInit(void): initialized motor using PWM
* void RTOSMotorDriveLeft(int32\_t freq): drives the left motor
* void RTOSMotorDriveRight(int32\_t freq): drives the right motor
* int RTOSMotorGetMaxPWM(): gets the max PWM value
* void RTOSMotorStop(): stop both the left and right motor

*RTOSPID Functions*

* void RTOSPIDInit(void): initializes PID settings
* void RTOSPIDGetSettings(char\* s): gets PID settings and returns to user
* void RTOSPIDSetSetting(int32\_t type, int32\_t value): sets PID settings
* void RTOSPIDCompute(): computes PID values and error values
* void RTOSPID\_ClockHandler(UArg arg): PID clock handler
* void RTOSPIDRun(): runs the PID using the other functions
* void RTOSPIDTuneP(int8\_t v): tunes PID Kp value
* void RTOSPIDTuneD(int8\_t v): tunes PID Kd value
* void RTOSPIDTuneI(int8\_t v): tunes PID Ki value
* void RTOSPIDGet(int\* p, int\* i, int\* d): gets PID Kp, Ki, and Kd values
* void RTOSPIDPIDEnable(int8\_t flag): enables PID by setting flag to 1
* void RTOSPIDRunAlongWall(): this state is for the robot run a long a wall with a distance
* void RTOSPIDRightTurn(): the robot will make a right turn
* bool RTOSPIDTurnAround(): the robot keep turning left until find a path (u-turn or turn left) and returns true when the new path is found

**Commented code:**

main.c :

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

/\* Standard libs \*/

#include <stdio.h>

#include <stdint.h>

#include <stdbool.h>

/\* Constants \*/

#include <inc/hw\_ints.h>

#include <inc/hw\_memmap.h>

#include <inc/hw\_types.h>

#include <inc/hw\_gpio.h>

/\* XDC \*/

#include <xdc/runtime/System.h>

#include <xdc/runtime/Log.h>

#include <xdc/cfg/global.h>

/\* DriverLib \*/

#include <driverlib/sysctl.h>

#include <driverlib/pin\_map.h>

#include <driverlib/gpio.h>

#include <driverlib/uart.h>

#include <driverlib/pwm.h>

#include <driverlib/timer.h>

#include <driverlib/interrupt.h>

/\* TI-RTOS BIOS \*/

#include <ti/sysbios/BIOS.h>

#include <ti/sysbios/knl/Task.h>

///\* TI-RTOS Peripherals \*/

#include <ti/drivers/GPIO.h>

#include <ti/drivers/UART.h>

#include <ti/drivers/uart/UARTTiva.h>

//

#include <RTOS/RTOSFlashing.h>

#include <RTOS/RTOSBluetooth.h>

#include <RTOS/RTOSDistance.h>

#include <RTOS/RTOSPID.h>

#include <RTOS/RTOSMotor.h>

#include <RTOS/RTOSLightSensor.h>

#include <RTOS/RTOSModbus.h>

/\*

\* ======== Hardware Configuration ========

\*/

//void Float2Str(char \*c, float f);

uint32\_t ui32Period;

// \*\*\* Board Initialization Function \*\*\*

void Board\_Init() {

//Set CPU Clock to 40MHz. 400MHz PLL/2 = 200 DIV 5 = 40MHz

//SysCtlClockSet(SYSCTL\_SYSDIV\_5|SYSCTL\_USE\_PLL|SYSCTL\_XTAL\_16MHZ|SYSCTL\_OSC\_MAIN);

RTOSFlashingInit();

RTOSBTInit();

RTOSDistanceInit();

RTOSPIDInit();

RTOSMotorInit();

RTOSLightSensorInit();

RTOSModbusInit();

}

// ======== main ========

int main(void){

// Timer 3 Hardware Interrupt Setup

// BIOS Setting in empty.cfg: 80000000 = 80 MHz

SysCtlPeripheralEnable(SYSCTL\_PERIPH\_TIMER3); // enable Timer 3 periph clks

TimerConfigure(TIMER3\_BASE, TIMER\_CFG\_PERIODIC); // cfg Timer 3 mode - periodic

ui32Period = (SysCtlClockGet()/100); // period = CPU clk div 2 (50ms)

TimerLoadSet(TIMER3\_BASE, TIMER\_A, ui32Period); // set Timer 2 period

IntEnable(INT\_TIMER3A);

TimerIntEnable(TIMER3\_BASE, TIMER\_TIMA\_TIMEOUT); // enables Timer 2 to interrupt CPU

IntMasterEnable();

TimerEnable(TIMER3\_BASE, TIMER\_A); // enable Timer 2

Board\_Init();

BIOS\_start();

return (0);

}

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Empty.cfg :

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

/\* ================ Clock configuration ================ \*/

var Clock = xdc.useModule('ti.sysbios.knl.Clock');

var LoggingSetup = xdc.useModule('ti.uia.sysbios.LoggingSetup');

var Hwi = xdc.useModule('ti.sysbios.hal.Hwi');

var Timer = xdc.useModule('ti.sysbios.hal.Timer');

/\*

\* Default value is family dependent. For example, Linux systems often only

\* support a minimum period of 10000 us and multiples of 10000 us.

\* TI platforms have a default of 1000 us.

\*/

Clock.tickPeriod = 1000;

/\* ================ Defaults (module) configuration ================ \*/

var Defaults = xdc.useModule('xdc.runtime.Defaults');

//Defaults.common$.namedModule = false;

/\* ================ Error configuration ================ \*/

var Error = xdc.useModule('xdc.runtime.Error');

Error.policyFxn = Error.policyDefault;

//Error.policyFxn = Error.policySpin;

Error.raiseHook = Error.print;

//Error.raiseHook = null;

//Error.raiseHook = "&myErrorFxn";

Error.maxDepth = 2;

//m3Hwi.nvicCCR.UNALIGN\_TRP = 1;

/\* ================ Idle configuration ================ \*/

var Idle = xdc.useModule('ti.sysbios.knl.Idle');

//Idle.addFunc("&myIdleFunc");

/\* ================ Kernel (SYS/BIOS) configuration ================ \*/

var BIOS = xdc.useModule('ti.sysbios.BIOS');

BIOS.assertsEnabled = true;

//BIOS.assertsEnabled = false;

BIOS.heapSize = 768;

BIOS.includeXdcRuntime = false;

//BIOS.includeXdcRuntime = true;

BIOS.libType = BIOS.LibType\_Custom;

//BIOS.libType = BIOS.LibType\_Debug;

BIOS.runtimeCreatesEnabled = true;

//BIOS.runtimeCreatesEnabled = false;

//BIOS.logsEnabled = true;

BIOS.logsEnabled = true;

/\* ================ Memory configuration ================ \*/

var Memory = xdc.useModule('xdc.runtime.Memory');

/\* ================ Program configuration ================ \*/

if (!Program.build.target.$name.match(/iar/)) {

/\*

\* Reducing the system stack size (used by ISRs and Swis) to reduce

\* RAM usage.

\*/

Program.stack = 1024;

}

if (Program.build.target.$name.match(/gnu/)) {

var SemiHost = xdc.useModule('ti.sysbios.rts.gnu.SemiHostSupport');

}

/\* ================ Semaphore configuration ================ \*/

var Semaphore = xdc.useModule('ti.sysbios.knl.Semaphore');

//Semaphore.supportsPriority = true;

Semaphore.supportsPriority = false;

//Semaphore.supportsEvents = true;

Semaphore.supportsEvents = false;

/\* ================ Swi configuration ================ \*/

var Swi = xdc.useModule('ti.sysbios.knl.Swi');

/\* ================ System configuration ================ \*/

var System = xdc.useModule('xdc.runtime.System');

System.abortFxn = System.abortStd;

//System.abortFxn = System.abortSpin;

//System.abortFxn = "&myAbortSystem";

System.exitFxn = System.exitStd;

//System.exitFxn = System.exitSpin;

//System.exitFxn = "&myExitSystem";

System.maxAtexitHandlers = 2;

var SysMin = xdc.useModule('xdc.runtime.SysMin');

SysMin.bufSize = 128;

System.SupportProxy = SysMin;

//var SysCallback = xdc.useModule('xdc.runtime.SysCallback');

//System.SupportProxy = SysCallback;

//SysCallback.abortFxn = "&myUserAbort";

//SysCallback.exitFxn = "&myUserExit";

//SysCallback.flushFxn = "&myUserFlush";

//SysCallback.putchFxn = "&myUserPutch";

//SysCallback.readyFxn = "&myUserReady";

/\* ================ Task configuration ================ \*/

var Task = xdc.useModule('ti.sysbios.knl.Task');

Task.checkStackFlag = true;

//Task.checkStackFlag = false;

Task.defaultStackSize = 512;

Task.enableIdleTask = true;

//Task.enableIdleTask = false;

//Task.allBlockedFunc = Idle.run;

Task.idleTaskStackSize = 512;

Task.numPriorities = 16;

/\* ================ Text configuration ================ \*/

var Text = xdc.useModule('xdc.runtime.Text');

Text.isLoaded = true;

//Text.isLoaded = false;

/\* ================ Types configuration ================ \*/

var Types = xdc.useModule('xdc.runtime.Types');

/\*

\* This module defines basic constants and types used throughout the

\* xdc.runtime package.

\*/

/\* ================ TI-RTOS middleware configuration ================ \*/

var mwConfig = xdc.useModule('ti.mw.Config');

/\*

\* Include TI-RTOS middleware libraries

\*/

/\* ================ TI-RTOS drivers' configuration ================ \*/

var driversConfig = xdc.useModule('ti.drivers.Config');

/\*

\* Include TI-RTOS drivers

\*

\* Pick one:

\* - driversConfig.LibType\_NonInstrumented (default)

\* Use TI-RTOS drivers library optimized for footprint and performance

\* without asserts or logs.

\* - driversConfig.LibType\_Instrumented

\* Use TI-RTOS drivers library for debugging with asserts and logs enabled.

\*/

driversConfig.libType = driversConfig.LibType\_NonInstrumented;

LoggingSetup.loadHwiLogging = false;

LoggingSetup.sysbiosHwiLogging = true;

LoggingSetup.loggerType = LoggingSetup.LoggerType\_MIN;

LoggingSetup.sysbiosSwiLogging = false;

LoggingSetup.loadTaskLogging = false;

LoggingSetup.loadSwiLogging = false;

LoggingSetup.sysbiosSemaphoreLogging = false;

LoggingSetup.enableTaskProfiler = false;

LoggingSetup.sysbiosTaskLogging = true;

LoggingSetup.loadLogging = false;

Clock.tickMode = Clock.TickMode\_PERIODIC;

var clock1Params = new Clock.Params();

clock1Params.instance.name = "RTOSFlashingHandle";

clock1Params.period = 5000;

clock1Params.startFlag = true;

Program.global.RTOSFlashingHandle = Clock.create("&RTOSFlashing\_ClockHandler", 5000, clock1Params);

var clock2Params = new Clock.Params();

clock2Params.instance.name = "RTOSBluetoothHandler";

clock2Params.period = 100;

clock2Params.startFlag = true;

Program.global.RTOSBluetoothHandler = Clock.create("&RTOSBluetooth\_ClockHandler", 1500, clock2Params);

var hwi0Params = new Hwi.Params();

hwi0Params.instance.name = "hwi0";

Program.global.hwi0 = Hwi.create(51, "&RTOSPIDRun", hwi0Params);

var clock3Params = new Clock.Params();

clock3Params.instance.name = "RTOSLightHandle";

clock3Params.period = 25;

clock3Params.startFlag = true;

Program.global.RTOSLightHandle = Clock.create("&RTOSLightSensorTick", 1000, clock3Params);

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

RTOSBluetooth.c :

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

/\*

\* RTOSBluetooth.c

\*

\* Created on: Oct 2, 2021

\* Author: dattr

\*/

#include "RTOSBluetooth.h"

#include "RTOSCommander.h"

#include "RTOSFlashing.h"

/\* Standard libs \*/

#include <stdio.h>

#include <stdint.h>

#include <stdbool.h>

#include <string.h>

/\* Constants \*/

#include <inc/hw\_ints.h>

#include <inc/hw\_memmap.h>

#include <inc/hw\_types.h>

#include <inc/hw\_gpio.h>

/\* XDC \*/

#include <xdc/runtime/System.h>

#include <xdc/runtime/Log.h>

#include <xdc/cfg/global.h>

#include <ti/sysbios/BIOS.h>

#include <ti/sysbios/knl/Task.h>

#include <ti/sysbios/knl/Clock.h>

#include <ti/sysbios/knl/Semaphore.h>

#include <xdc/std.h>

#include <xdc/runtime/System.h>

/\* TI-RTOS Peripherals \*/

#include <ti/drivers/GPIO.h>

#include <ti/drivers/UART.h>

#include <ti/drivers/uart/UARTTiva.h>

/\* DriverLib \*/

#include <driverlib/sysctl.h>

#include <driverlib/pin\_map.h>

#include <driverlib/gpio.h>

#include <driverlib/uart.h>

// Configure Bluetooth Port Here

typedef struct {

const uint32\_t periph\_gpio;

const uint32\_t periph\_uart;

const uint32\_t gpio\_rx;

const uint32\_t gpio\_tx;

const uint32\_t pin\_rx;

const uint32\_t pin\_tx;

const uint32\_t gpio\_base;

const uint32\_t uart\_base;

const uint32\_t baudRate;

} RTOSTConfig;

const RTOSTConfig rtosbtconfig = {

// setup the port here

.periph\_gpio = SYSCTL\_PERIPH\_GPIOB,// SYSCTL\_PERIPH\_GPIOE,

.periph\_uart = SYSCTL\_PERIPH\_UART1, //SYSCTL\_PERIPH\_UART5,

.gpio\_tx = GPIO\_PB1\_U1TX,//GPIO\_PE5\_U5TX,

.gpio\_rx = GPIO\_PB0\_U1RX,//GPIO\_PE4\_U5RX,

.pin\_tx = GPIO\_PIN\_1,//GPIO\_PIN\_5,

.pin\_rx = GPIO\_PIN\_0,//GPIO\_PIN\_4,

.gpio\_base = GPIO\_PORTB\_BASE,// GPIO\_PORTE\_BASE,

.uart\_base = UART1\_BASE,//UART5\_BASE,

.baudRate = 9600

};

// \*\*\* UART Configuration \*\*\*

UARTTiva\_Object uartTivaObjects[1];

unsigned char uartTivaRingBuffer[1][32];

// UART configuration structures

const UARTTiva\_HWAttrs uartTivaHWAttrs[1] = {

{

.baseAddr = UART1\_BASE,// UART5\_BASE,

.intNum = INT\_UART1,//INT\_UART5,

.intPriority = (~0),

.flowControl = UART\_FLOWCONTROL\_NONE,

.ringBufPtr = uartTivaRingBuffer[0],

.ringBufSize = sizeof(uartTivaRingBuffer[0])

}

};

const UART\_Config UART\_config[] = {

{

.fxnTablePtr = &UARTTiva\_fxnTable,

.object = &uartTivaObjects[0],

.hwAttrs = &uartTivaHWAttrs[0]

},

{NULL, NULL, NULL}

};

// Task's stack

uint8\_t RTOSBT\_TXTaskStack[512];

uint8\_t RTOSBT\_RXTaskStack[512];

// Task object (to be constructed)

Task\_Struct RTOSBT\_TXTaskStruct;

Task\_Struct RTOSBT\_RXTaskStruct;

// Configure task

Task\_Params RTOSBT\_TXTaskParams;

Task\_Params RTOSBT\_RXTaskParams;

// Constructing a Semaphore

Semaphore\_Handle RTOSBT\_TXSem;

Semaphore\_Handle RTOSBT\_RXSem;

Semaphore\_Struct RTOSBT\_TXSemStruct; // Memory allocated at build time

Semaphore\_Struct RTOSBT\_RXSemStruct; // Memory allocated at build time

// Bluetooth UART

UART\_Handle RTOSBT\_Uart;

// Command storage

static char RTOSBT\_Command[32];

static int RTOSBT\_CommandLen = 0;

char\* RTOSBT\_WriteString;

int RTOSBT\_WriteStringMaxLen = 256;

void RTOSBTInit() {

// \*\*\*\*\*\* UART Init \*\*\*\*\*\*

SysCtlPeripheralEnable(rtosbtconfig.periph\_gpio);

SysCtlPeripheralEnable(rtosbtconfig.periph\_uart);

GPIOPinConfigure(rtosbtconfig.gpio\_tx);

GPIOPinConfigure(rtosbtconfig.gpio\_rx);

GPIOPinTypeUART(rtosbtconfig.gpio\_base, rtosbtconfig.pin\_tx | rtosbtconfig.pin\_rx);

UART\_init();

// Wait for the UART module to be ready

//while(!SysCtlPeripheralReady(SYSCTL\_PERIPH\_UART5)) { }

// Configure task

// Task\_Params RTOSBT\_TXTaskParams;

// Task\_Params RTOSBT\_RXTaskParams;

Task\_Params\_init(&RTOSBT\_TXTaskParams);

RTOSBT\_TXTaskParams.stack = RTOSBT\_TXTaskStack;

RTOSBT\_TXTaskParams.stackSize = 512;

RTOSBT\_TXTaskParams.priority = 11;

Task\_Params\_init(&RTOSBT\_RXTaskParams);

RTOSBT\_RXTaskParams.stack = RTOSBT\_RXTaskStack;

RTOSBT\_RXTaskParams.stackSize = 512;

RTOSBT\_RXTaskParams.priority = 10;

Task\_construct(&RTOSBT\_TXTaskStruct, RTOSBTWriteTaskFunc, &RTOSBT\_TXTaskParams, NULL);

Task\_construct(&RTOSBT\_RXTaskStruct, RTOSBTReadTaskFunc, &RTOSBT\_RXTaskParams, NULL);

// Configure semaphore

Semaphore\_Params RTOSBT\_TXSemParams;

Semaphore\_Params RTOSBT\_RXSemParams;

Semaphore\_Params\_init(&RTOSBT\_TXSemParams);

Semaphore\_Params\_init(&RTOSBT\_RXSemParams);

Semaphore\_construct(&RTOSBT\_TXSemStruct, 0, &RTOSBT\_TXSemParams);

Semaphore\_construct(&RTOSBT\_RXSemStruct, 0, &RTOSBT\_RXSemParams);

//It's optional to store the handle

RTOSBT\_TXSem = Semaphore\_handle(&RTOSBT\_TXSemStruct);

RTOSBT\_RXSem = Semaphore\_handle(&RTOSBT\_RXSemStruct);

//Configure the UART Module

UART\_Params uartParams;

// Create a UART with data processing off.

UART\_Params\_init(&uartParams);

uartParams.writeDataMode = UART\_DATA\_BINARY;

uartParams.readDataMode = UART\_DATA\_BINARY;

uartParams.readReturnMode = UART\_RETURN\_FULL;

uartParams.readEcho = UART\_ECHO\_ON;

uartParams.baudRate = rtosbtconfig.baudRate;

//uartParams.baudRate = 115200;

RTOSBT\_Uart = UART\_open(0, &uartParams);

if (RTOSBT\_Uart == NULL) { // If UART fails

// Use RTOSFlashing to update status

RTOSFlashingSet(2);

System\_abort("\nRTOSBluetooth UART Failed to open.");

return;

}

}

void RTOSBTWriteTaskFunc(UArg arg0, UArg arg1) {

char \*c;

int count = 0;

while(1) {

Semaphore\_pend(RTOSBT\_TXSem, BIOS\_WAIT\_FOREVER);

c = RTOSBT\_WriteString;

count = 0;

while (\*c != '\0' | count > RTOSBT\_WriteStringMaxLen) {

UARTCharPut(rtosbtconfig.uart\_base, \*c);

c++;

count++; // insurance for infinite loop

}

}

}

void RTOSWriteString(char\* cs) {

RTOSBT\_WriteString = cs;

Semaphore\_post(RTOSBT\_TXSem);

}

void RTOSBTReadTaskFunc(UArg arg0, UArg arg1) {

unsigned char rxBuffer[1];

int readSize;

int temp;

while(1) {

Semaphore\_pend(RTOSBT\_RXSem, BIOS\_WAIT\_FOREVER);

readSize = UART\_read(RTOSBT\_Uart, rxBuffer, sizeof(rxBuffer));

UART\_write(RTOSBT\_Uart, rxBuffer, sizeof(rxBuffer));

if (readSize == 0) continue; // skip when no data

// if c is not \n, append the command

// otherwise, execute command

if (rxBuffer[0] != '\n' && rxBuffer[0] != '\r') {

// append the command

RTOSBT\_Command[RTOSBT\_CommandLen++] = rxBuffer[0];

} else {

RTOSBT\_Command[RTOSBT\_CommandLen] = '\0'; // end of string

// execute command

temp = RTOSRunCommand(RTOSBT\_Command);

if (temp == 1) {

RTOSWriteString("\n\rCommand Executed.\n\rEnter command: ");

} else {

// invalid command

RTOSWriteString("\n\rInvalid command.\n\rEnter command: ");

}

RTOSBT\_CommandLen = 0; // reset string length

RTOSBT\_Command[0] = '\0';

}

}

}

void RTOSBluetooth\_ClockHandler(UArg arg) {

Semaphore\_post(RTOSBT\_RXSem);

}

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

RTOSBluetooth.h :

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

/\*

\* RTOSBluetooth.h

\*

\* Created on: Oct 2, 2021

\* Author: dattr

\* Summary: This RTOSBluetooth is used to initialize and run bluetooth UART by using Swi, Sophomore, and Task

\* Interrupt Clock need to create in .cfg as below:

\* Clock Handle: RTOSBluetoothHandler

\* Clock Function: RTOSBluetooth\_ClockHandler

\* Initial Timeout: 500 (as 0.5 seconds)

\* Clock period: 100 (as 100 ms) recommend minimum value at 10 for Bluetooth Low Energy

\*/

#ifndef RTOS\_RTOSBLUETOOTH\_H\_

#define RTOS\_RTOSBLUETOOTH\_H\_

#include <xdc/std.h>

void RTOSBTInit();

void RTOSBTWriteLine(char \*c);

void RTOSBTReadLine(char \*arr);

void RTOSBTWriteTaskFunc(UArg arg0, UArg arg1);

void RTOSBTReadTaskFunc(UArg arg0, UArg arg1);

void RTOSWriteString(char\* cs);

#endif /\* RTOS\_RTOSBLUETOOTH\_H\_ \*/

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

RTOSCommander.c :

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

/\*

\* RTOSCommander.c

\*

\* Created on: Oct 3, 2021

\* Author: dattr

\*/

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <stdbool.h>

#include <RTOS/RTOSFlashing.h>

#include <RTOS/RTOSBluetooth.h>

#include <math.h>

#include <xdc/runtime/System.h>

#include <RTOS/RTOSMotor.h>

#include <RTOS/RTOSPID.h>

#include <RTOS/RTOSCommander.h>

#include <RTOS/RTOSDistance.h>

#include "driverlib/sysctl.h"

#include <RTOS/RTOSLightSensor.h>

char RTOSCommander\_Raw[32]; // store command before proceed

char \*RTOSCommander\_token;

// temperory output command

char RTOSCommanderOutString[256];

unsigned int commandCount = 0;

int RTOSCommanderTemp[3];

uint32\_t RTOSCommanderTempU[3];

char RTOSChar1[10];

char RTOSChar2[10];

void RTOSCommanderInit() {

// For advance only

// Use queue interrupt to push command

}

// return 1 when executed

// return 0 when invalid command

// for advance: this one should put in queue

int RTOSRunCommand(char \*cmd) {

int r = 1;

// copy raw command

strcpy(RTOSCommander\_Raw, cmd);

// split command to args

RTOSCommander\_token = strtok(RTOSCommander\_Raw, " ");

// execute command

if (strcmp(RTOSCommander\_token, "d") == 0) {

// get 1st argument

RTOSCommander\_token = strtok(NULL, " ");

if (RTOSCommander\_token == NULL) {

RTOSWriteString("\n\rAn arg is required for [debug] command.");

return 0;

}

if (strcmp(RTOSCommander\_token, "normal") == 0) {

RTOSFlashingSet(0);

} else if (strcmp(RTOSCommander\_token, "error") == 0) {

RTOSFlashingSet(1);

} else if (strcmp(RTOSCommander\_token, "bt") == 0) {

RTOSFlashingSet(2);

} else if (strcmp(RTOSCommander\_token, "mr") == 0) {

// send command: debug motor\_right

// get 2nd argument

RTOSCommander\_token = strtok(NULL, " ");

RTOSMotorDriveRight(atoi(RTOSCommander\_token));

} else if (strcmp(RTOSCommander\_token, "ml") == 0) {

// send command: debug motor\_left

// get 2nd argument

RTOSCommander\_token = strtok(NULL, " ");

RTOSMotorDriveLeft(atoi(RTOSCommander\_token));

}else if (strcmp(RTOSCommander\_token, "d") == 0) {

RTOSDistanceGet(&RTOSCommanderTempU[0], &RTOSCommanderTempU[1]);

// Float2Str(&RTOSChar1[0], RTOSCommanderTemp[0]);

// Float2Str(&RTOSChar2[0], RTOSCommanderTemp[1]);

System\_sprintf(RTOSCommanderOutString, "\n\rDistance to front: %d mm \n\rDistance to right: %d mm\n\r",

RTOSCommanderTempU[0]/10, RTOSCommanderTempU[1]/10);

RTOSWriteString(RTOSCommanderOutString);

}

else if (strcmp(RTOSCommander\_token, "clk") == 0) {

System\_sprintf(RTOSCommanderOutString, "\n\rSystem Clock: %d \n\r", SysCtlClockGet());

RTOSWriteString(RTOSCommanderOutString);

}

else if (strcmp(RTOSCommander\_token, "t") == 0) {

System\_sprintf(RTOSCommanderOutString, "\n\rLight Timer Count: %d \n\r", RTOSLightSensorCount());

RTOSWriteString(RTOSCommanderOutString);

}

else if (strcmp(RTOSCommander\_token, "timer") == 0) {

// get 2nd argument

RTOSCommander\_token = strtok(NULL, " ");

RTOSCommanderTempU[0] = atoi(RTOSCommander\_token);

System\_sprintf(RTOSCommanderOutString, "\n\rLight Timer for %d line is: %d \n\r", RTOSCommanderTempU[0],

RTOSLightSensorGet(RTOSCommanderTempU[0]));

RTOSWriteString(RTOSCommanderOutString);

}

else {

r = 0;

System\_sprintf(RTOSCommanderOutString, "\n\rArg=%s is not valid in [debug] command.", RTOSCommander\_token);

RTOSWriteString(RTOSCommanderOutString);

}

}

// setting

else if(strcmp(RTOSCommander\_token, "pid") == 0) {

// get 1st argument

RTOSCommander\_token = strtok(NULL, " ");

if (RTOSCommander\_token == NULL) {

RTOSWriteString("\n\rAn arg is required for [pid] command.");

return 0;

}

if (strcmp(RTOSCommander\_token, "get") == 0) {

RTOSPIDGetSettings(RTOSCommanderOutString);

RTOSWriteString(RTOSCommanderOutString);

} else if (strcmp(RTOSCommander\_token, "set") == 0) {

// get 2nd, 3rd, 4th values as PID

RTOSCommander\_token = strtok(NULL, " ");

RTOSPIDSetSetting(0, atoi(RTOSCommander\_token));

RTOSCommander\_token = strtok(NULL, " ");

RTOSPIDSetSetting(1, atoi(RTOSCommander\_token));

RTOSCommander\_token = strtok(NULL, " ");

RTOSPIDSetSetting(2, atoi(RTOSCommander\_token));

} else if (strcmp(RTOSCommander\_token, "setright") == 0) {

// get 2nd argument

RTOSCommander\_token = strtok(NULL, " ");

RTOSPIDSetSetting(3, atoi(RTOSCommander\_token));

} else if (strcmp(RTOSCommander\_token, "setfront") == 0) {

// get 2nd argument

RTOSCommander\_token = strtok(NULL, " ");

RTOSPIDSetSetting(4, atoi(RTOSCommander\_token));

} else if (strcmp(RTOSCommander\_token, "setpwmright") == 0) {

// get 2nd argument

RTOSCommander\_token = strtok(NULL, " ");

RTOSPIDSetSetting(5, atoi(RTOSCommander\_token));

} else if (strcmp(RTOSCommander\_token, "setpwmleft") == 0) {

// get 2nd argument

RTOSCommander\_token = strtok(NULL, " ");

RTOSPIDSetSetting(6, atoi(RTOSCommander\_token));

}

else if (strcmp(RTOSCommander\_token, "start") == 0) {

RTOSPIDPIDEnable(1);

}

else if (strcmp(RTOSCommander\_token, "stop") == 0) {

RTOSPIDPIDEnable(0);

}

else {

r = 0;

System\_sprintf(RTOSCommanderOutString, "\n\rArg=%s is not valid in [pid] command.", RTOSCommander\_token);

RTOSWriteString(RTOSCommanderOutString);

}

}

else if(strcmp(RTOSCommander\_token, "t") == 0) { // tune values in PID

// get 1st argument

RTOSCommander\_token = strtok(NULL, " ");

if (RTOSCommander\_token == NULL) {

RTOSWriteString("\n\rAn arg is required for [tune pid] command.");

return 0;

}

if (strcmp(RTOSCommander\_token, "p+") == 0) {

RTOSPIDTuneP(1);

}

else if (strcmp(RTOSCommander\_token, "p-") == 0) {

RTOSPIDTuneP(-1);

}

else if (strcmp(RTOSCommander\_token, "i+") == 0) {

RTOSPIDTuneI(1);

}

else if (strcmp(RTOSCommander\_token, "i-") == 0) {

RTOSPIDTuneI(-1);

}

else if (strcmp(RTOSCommander\_token, "d+") == 0) {

RTOSPIDTuneD(1);

}

else if (strcmp(RTOSCommander\_token, "d-") == 0) {

RTOSPIDTuneD(-1);

}

// print PID values

RTOSPIDGet(&RTOSCommanderTemp[0], &RTOSCommanderTemp[1], &RTOSCommanderTemp[2]);

System\_sprintf(RTOSCommanderOutString, "\n\rPID Values: %d %d %d\n\r",

RTOSCommanderTemp[0], RTOSCommanderTemp[1], RTOSCommanderTemp[2]);

RTOSWriteString(RTOSCommanderOutString);

}

else if(strcmp(RTOSCommander\_token, "ss") == 0) {

RTOSPIDPIDEnable(1);

}

else if(strcmp(RTOSCommander\_token, "st") == 0) {

RTOSPIDPIDEnable(0);

}

else {

// invalid command

r = 0;

}

commandCount++;

return r;

}

int temp1;

int temp2;

void Float2Str(char \*c, float f) {

temp1 = f;

f = f - temp1;

temp2 = f \* 10000;

System\_sprintf(c, "%d.%04d", temp1, temp2);

}

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

RTOSCommander.h :

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

/\*

\* RTOSCommander.h

\*

\* Created on: Oct 3, 2021

\* Author: dattr

\*/

#ifndef RTOS\_RTOSCOMMANDER\_H\_

#define RTOS\_RTOSCOMMANDER\_H\_

#include <xdc/std.h>

// init commander

void RTOSCommanderInit();

// return 1 when executed

// return 0 when invalid command

int RTOSRunCommand(char \*cmd);

void Float2Str(char \*c, float f);

#endif /\* RTOS\_RTOSCOMMANDER\_H\_ \*/

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

RTOSDistance.c :

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

/\*

\* RTOSDistance.c

\*

\* Created on: Oct 26, 2021

\* Author: dattr

\*/

#include <stdint.h>

#include <stdbool.h>

#include <stdio.h>

#include <math.h>

#include <xdc/runtime/System.h>

#include "inc/hw\_types.h"

#include "inc/hw\_memmap.h"

#include "inc/tm4c123gh6pm.h"

#include "inc/hw\_gpio.h"

#include "inc/hw\_timer.h"

#include <ti/sysbios/BIOS.h>

#include <ti/sysbios/knl/Task.h>

#include "driverlib/sysctl.h"// to enable ports

#include "driverlib/interrupt.h"

#include "driverlib/pin\_map.h"

#include "driverlib/gpio.h"

#include "driverlib/timer.h"

#include "driverlib/adc.h"

#include <time.h>

#include <RTOS/RTOSDistance.h>

uint32\_t RTOSDistanceFrontVol, RTOSDistanceRightVol;

//float RTOSDistanceFrontCm, RTOSDistanceRightCm;

uint32\_t RTOSDistanceFrontTemp, RTOSDistanceRightTemp;

uint16\_t RTOSILoop;

void RTOSDistanceInit(void) {

// RTOSDistanceFrontVol = RTOSDistanceRightVol = 0;

// RTOSDistanceFrontCm = RTOSDistanceRightCm = 0.0;

// Setup ADC PE3 for front distance sensor - PE2 for right distance sensor

// Enable ADC0 module

SysCtlPeripheralEnable(SYSCTL\_PERIPH\_ADC0);

//SysCtlPeripheralEnable(SYSCTL\_PERIPH\_ADC1);

SysCtlPeripheralEnable(SYSCTL\_PERIPH\_GPIOE);

// configure PE3 for input

GPIOPinTypeADC(GPIO\_PORTE\_BASE, GPIO\_PIN\_2 | GPIO\_PIN\_3);

// Configure sample sequencer

ADCSequenceDisable(ADC0\_BASE, 0);

ADCSequenceDisable(ADC0\_BASE, 1);

ADCSequenceConfigure(ADC0\_BASE, 0, ADC\_TRIGGER\_PROCESSOR, 0);

ADCSequenceStepConfigure(ADC0\_BASE, 0, 0, ADC\_CTL\_IE | ADC\_CTL\_END | ADC\_CTL\_CH0);

ADCSequenceStepConfigure(ADC0\_BASE, 0, 1, ADC\_CTL\_IE | ADC\_CTL\_END | ADC\_CTL\_CH0);

ADCSequenceConfigure(ADC0\_BASE, 1, ADC\_TRIGGER\_PROCESSOR, 0);

ADCSequenceStepConfigure(ADC0\_BASE, 1, 0, ADC\_CTL\_IE | ADC\_CTL\_END | ADC\_CTL\_CH1);

ADCSequenceStepConfigure(ADC0\_BASE, 1, 1, ADC\_CTL\_IE | ADC\_CTL\_END | ADC\_CTL\_CH1);

ADCSequenceEnable(ADC0\_BASE, 0);

ADCSequenceEnable(ADC0\_BASE, 1);

}

void RTOSDistanceGet(uint32\_t\* front, uint32\_t\* right) {

RTOSDistanceRightTemp = 0;

RTOSDistanceFrontTemp = 0;

for (RTOSILoop = 0; RTOSILoop < 5; RTOSILoop++) {

// clear ADC interrupt

ADCIntClear(ADC0\_BASE, 0);

ADCIntClear(ADC0\_BASE, 1);

// trigger ADC sampling

ADCProcessorTrigger(ADC0\_BASE, 0);

ADCProcessorTrigger(ADC0\_BASE, 1);

// read voltage

ADCSequenceDataGet(ADC0\_BASE, 0, &RTOSDistanceRightVol);

ADCSequenceDataGet(ADC0\_BASE, 1, &RTOSDistanceFrontVol);

RTOSDistanceRightTemp += VoltageToCmRight(RTOSDistanceRightVol);

RTOSDistanceFrontTemp += VoltageToCmFront(RTOSDistanceFrontVol);

//RTOSDistanceRightTemp += RTOSDistanceRightVol;

//RTOSDistanceFrontTemp += RTOSDistanceFrontVol;

}

RTOSDistanceRightTemp = RTOSDistanceRightTemp / 5;

RTOSDistanceFrontTemp = RTOSDistanceFrontTemp / 5;

if (RTOSDistanceRightTemp < 400) \*right = 400; // minimum of 4cm

else if (RTOSDistanceRightTemp > 4000) \*right = 4000; // maximum of 40cm

else \*right = RTOSDistanceRightTemp;

//\*right = RTOSDistanceRightTemp;

if (RTOSDistanceFrontTemp < 400) \*front = 400; // minimum of 4cm

else if (RTOSDistanceFrontTemp > 3000) \*front = 3000; // maximum of 40cm

else \*front = RTOSDistanceFrontTemp;

//\*front = RTOSDistanceFrontTemp;

//printf("%d",RTOSDistanceFrontTemp);

}

// return [mm]

uint32\_t VoltageToCmRight(uint32\_t voltage) {

// Calibration to get the voltage equation to 10x[mm]

// power factor 0.984 < 1 should cause error

// replace with 1

return (13764.0 / pow((int)voltage, 0.984)) \* 100;

//return (31165.0 / pow((int)voltage, 1.101)) \* 100;

}

uint32\_t VoltageToCmFront(uint32\_t voltage) {

// Calibration to get the voltage equation to 10x[mm]

return (31165.0 / pow((int)voltage, 1.101)) \* 100;

}

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

RTOSDistance.h :

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

/\*

\* RTOSDistance.h

\*

\* Created on: Oct 26, 2021

\* Author: dattr

\*/

#ifndef RTOS\_RTOSDISTANCE\_H\_

#define RTOS\_RTOSDISTANCE\_H\_

void RTOSDistanceInit(void);

// Get raw data from Distance sensor on the front and on the right

// We can update the calibration distance

void RTOSDistanceGet(uint32\_t\* front, uint32\_t\* right); // 10x[mm]

uint32\_t VoltageToCmRight(uint32\_t voltage); // 10x[mm]

uint32\_t VoltageToCmFront(uint32\_t voltage); // 10x[mm]

#endif /\* RTOS\_RTOSDISTANCE\_H\_ \*/

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

RTOSFlashing.c :

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

/\*

\* RTOSFlashing.c

\*

\* Created on: Oct 2, 2021

\* Author: dattr

\* THIS FILE USED TO SEE THE STATUS OF DEVICE BASED ON LED FLASHING ON BOARD

\* Interrupt Clock need to create in .cfg as below:

\* Clock Handle: RTOSFlashingHandle

\* Clock Function: RTOSFlashing\_ClockHandler

\* Initial Timeout: 5000 (as 5 seconds)

\* Clock period: 5000 (as 5 seconds)

\*

\*/

#include "RTOSFlashing.h"

#include <xdc/std.h>

#include <stdbool.h>

#include <inc/hw\_memmap.h>

#include <ti/sysbios/BIOS.h>

#include <ti/sysbios/knl/Task.h>

#include <ti/sysbios/knl/Semaphore.h>

#include <ti/drivers/uart/UARTTiva.h>

#include <driverlib/gpio.h>

#include <driverlib/sysctl.h>

int currentFlashingCode = 0;

bool isLEDFlashing = true;

// LED values - 2=RED, 4=BLUE, 8=GREEN

//uint8\_t ledCode[6] = {8, 0, 8, 0, 8, 0};

typedef struct {

uint8\_t ledCode[6];

} RTOSFlashingLEDCode;

RTOSFlashingLEDCode flashingCode = {8, 0, 8, 0, 8, 0};

uint8\_t blinkingColor = 0;

/\* Task's stack \*/

uint8\_t RTOSFlashingTaskStack[512];

/\* Task object (to be constructed) \*/

Task\_Struct RTOSFlashingtask0;

// Constructing a Semaphore

Semaphore\_Handle RTOSFlashingSem;

Semaphore\_Struct RTOSFlashingStructSem; /\* Memory allocated at build time \*/

void RTOSFlashingInit() {

// Config LED on board

SysCtlPeripheralEnable(SYSCTL\_PERIPH\_GPIOF);

GPIOPinTypeGPIOOutput(GPIO\_PORTF\_BASE, GPIO\_PIN\_1|GPIO\_PIN\_2|GPIO\_PIN\_3);

Task\_Params RTOSFlashingTaskParams;

Semaphore\_Params RTOSFlashingSemParams;

// Configure task

Task\_Params\_init(&RTOSFlashingTaskParams);

RTOSFlashingTaskParams.stack = RTOSFlashingTaskStack;

RTOSFlashingTaskParams.stackSize = 512;

RTOSFlashingTaskParams.priority = 1;

Task\_construct(&RTOSFlashingtask0, RTOSFlashingTaskFunc, &RTOSFlashingTaskParams, NULL);

// Configure semaphore

Semaphore\_Params\_init(&RTOSFlashingSemParams);

Semaphore\_construct(&RTOSFlashingStructSem, 0, &RTOSFlashingSemParams);

//It's optional to store the handle

RTOSFlashingSem = Semaphore\_handle(&RTOSFlashingStructSem);

}

/\* Task function \*/

void RTOSFlashingTaskFunc(UArg arg0, UArg arg1)

{

/\* Local variables. Variables here go onto task stack!! \*/

Uint32 ms = 50;

int i = 0;

/\* Run one-time code when task starts \*/

while (1) /\* Run loop forever (unless terminated) \*/

{

Semaphore\_pend(RTOSFlashingSem, BIOS\_WAIT\_FOREVER);

if (isLEDFlashing == false) {

GPIOPinWrite(GPIO\_PORTF\_BASE, GPIO\_PIN\_1|GPIO\_PIN\_2|GPIO\_PIN\_3, blinkingColor);

Task\_sleep(50);

GPIOPinWrite(GPIO\_PORTF\_BASE, GPIO\_PIN\_1|GPIO\_PIN\_2|GPIO\_PIN\_3, 0);

blinkingColor = 0;

continue;

}

switch(currentFlashingCode) {

// LED values - 2=RED, 4=BLUE, 8=GREEN

case 1: // general error

flashingCode = (RTOSFlashingLEDCode) {2, 0, 2, 0, 2, 0};

break;

case 2: // No bluetooth connection

flashingCode = (RTOSFlashingLEDCode) {2, 0, 4, 0, 8, 0};

break;

case 3: // green

flashingCode = (RTOSFlashingLEDCode) {8, 8, 8, 8, 8, 8};

break;

case 4: //blue

flashingCode = (RTOSFlashingLEDCode) {4, 4, 4, 4, 4, 4};

break;

default: // set to 0 - normal status

flashingCode = (RTOSFlashingLEDCode) {8, 0, 8, 0, 8, 0};

}

for (i = 0; i < 6; i++) {

GPIOPinWrite(GPIO\_PORTF\_BASE, GPIO\_PIN\_1|GPIO\_PIN\_2|GPIO\_PIN\_3, flashingCode.ledCode[i]);

Task\_sleep(ms);

}

}

}

void RTOSFlashingSet(int flashingCode) {

isLEDFlashing = true;

currentFlashingCode = flashingCode;

}

int RTOSFlashingGet() {

return currentFlashingCode;

}

void RTOSFlashing\_ClockHandler(UArg arg) {

Semaphore\_post(RTOSFlashingSem);

}

void RTOSFlashingSetColor(uint8\_t color) {

isLEDFlashing = false;

blinkingColor = color;

}

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

RTOSFlashing.h :

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

/\*

\* RTOSFlashing.h

\*

\* Created on: Oct 2, 2021

\* Author: dattr

\* THIS FILE USED TO SEE THE STATUS OF DEVICE BASED ON LED FLASHING ON BOARD

\* Interrupt Clock need to create in .cfg as below:

\* Clock Handle: RTOSFlashingHandle

\* Clock Function: RTOSFlashing\_ClockHandler

\* Initial Timeout: 5000 (as 5 seconds)

\* Clock period: 5000 (as 5 seconds)

\* Code Status:

\* case 1: // general error {2, 0, 2, 0, 2, 0};

case 2: // No bluetooth connection {2, 0, 4, 0, 8, 0};

default: // normal {8, 0, 8, 0, 8, 0};

\*/

#ifndef RTOS\_RTOSFLASHING\_H\_

#define RTOS\_RTOSFLASHING\_H\_

#include <xdc/std.h>

void RTOSFlashingInit();

void RTOSFlashingRun();

void RTOSFlashingSet(int flashingCode);

int RTOSFlashingGet();

void RTOSFlashingTaskFunc(UArg arg0, UArg arg1);

void RTOSFlashing\_ClockHandler(UArg arg);

// LED values - 2=RED, 4=BLUE, 8=GREEN

void RTOSFlashingSetColor(uint8\_t color);

#endif /\* RTOS\_RTOSFLASHING\_H\_ \*/

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

RTOSLightSensor.c :

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

/\*

\* RTOSLightSensor.c

\*

\* Created on: Nov 8, 2021

\* Author: dattr

\*/

#include <stdint.h>

#include <stdbool.h>

#include <stdio.h>

#include "inc/hw\_memmap.h"

#include "driverlib/sysctl.h"

#include "driverlib/gpio.h"

#include "inc/hw\_gpio.h"

#include "driverlib/timer.h"

#include "inc/hw\_types.h"

#include "inc/hw\_timer.h"

#include <xdc/runtime/System.h>

#include <RTOS/RTOSLightSensor.h>

#include <RTOS/RTOSBluetooth.h>

#include <RTOS/RTOSCommander.h>

#include <RTOS/RTOSFlashing.h>

#include <RTOS/RTOSModbus.h>

uint32\_t RTOSLightDef = 20000; // thredhold value between white vs black, white ~ 8000, Black ~ 40,000

uint32\_t RTOSLightValue = 0;

uint32\_t RTOSLightCount = 0;

uint32\_t RTOSLightSensorHolder[20]; // hold number values of crossing the line, 0th index is the initialize

int16\_t RTOSLightSensorLineCount = 0; // count number of crossing line

uint32\_t RTOSLightTemp1, RTOSLightTemp2, RTOSLightTemp3;

bool RTOSLightWhiteLine = false;

char RTOSLightSensorWriteStr[50];

// timing

uint32\_t RTOSLightTimeCount = 0; // based on each

void RTOSLightSensorInit() {

// configure gpio

SysCtlPeripheralEnable(SYSCTL\_PERIPH\_GPIOC);

GPIOPinTypeGPIOOutput(GPIO\_PORTC\_BASE, GPIO\_PIN\_5);

// Timer for charging/discharging light sensor

SysCtlPeripheralEnable(SYSCTL\_PERIPH\_TIMER0); // enable Timer 2 periph clks

TimerConfigure(TIMER0\_BASE, TIMER\_CFG\_A\_ONE\_SHOT\_UP);

TimerLoadSet(TIMER0\_BASE, TIMER\_A, 0xFFFFFFFF);

TimerEnable(TIMER0\_BASE, TIMER\_A);

// Timer for crossing line

// This timer is used for calculate how long it takes to cross a black line

SysCtlPeripheralEnable(SYSCTL\_PERIPH\_TIMER1); // enable Timer 2 periph clks

TimerConfigure(TIMER1\_BASE, TIMER\_CFG\_A\_ONE\_SHOT\_UP);

TimerLoadSet(TIMER1\_BASE, TIMER\_A, 0xFFFFFFFF);

TimerEnable(TIMER1\_BASE, TIMER\_A);

RTOSLightValue = 0;

RTOSLightWhiteLine = true;

}

void RTOSLightSensorTick() {

uint32\_t startTime, endTime, pinValue;

HWREG(TIMER0\_BASE + TIMER\_O\_TAV) = 0; // set to 0 to start timing

GPIOPinTypeGPIOOutput(GPIO\_PORTC\_BASE, GPIO\_PIN\_5);

GPIOPinWrite(GPIO\_PORTC\_BASE, GPIO\_PIN\_5, 0xFF); //write into pin C5

//SysCtlDelay(6000); // Wait to finish charging

RTOSLightCount = 0;

while (RTOSLightCount < 1000) {

RTOSLightCount++;

}

startTime = TimerValueGet(TIMER0\_BASE, TIMER\_A); // Capture startTime

GPIOPinTypeGPIOInput(GPIO\_PORTC\_BASE, GPIO\_PIN\_5); // Make PortB pin1 input to time the decaying

pinValue = GPIOPinRead(GPIO\_PORTC\_BASE, GPIO\_PIN\_5); // Assign the value when sensor is in fully charged state

while (pinValue && GPIO\_PIN\_5) { // Loop to compare to the current state to fully-charged state

pinValue = GPIOPinRead(GPIO\_PORTC\_BASE, GPIO\_PIN\_5);

}

endTime = TimerValueGet(TIMER0\_BASE, TIMER\_A); // Capture endTime when fully decayed

RTOSLightValue = endTime - startTime;

if (RTOSLightValue < RTOSLightDef) { // robot cross white line

// LED values - 2=RED, 4=BLUE, 8=GREEN

RTOSFlashingSetColor(8);

if (RTOSLightWhiteLine == false) {

// stop the timer and store it

RTOSLightTemp2 = TimerValueGet(TIMER1\_BASE, TIMER\_A); // Capture startTime

RTOSLightSensorHolder[RTOSLightSensorLineCount++] = RTOSLightTemp2 - RTOSLightTemp1;

RTOSLightWhiteLine = true;

// print out when crossline

if (RTOSLightSensorLineCount == 1) {

RTOSWriteString("\n\r1st line initialized.");

// start timing

RTOSLightTimeCount = 0;

//HWREG(TIMER2\_BASE + TIMER\_O\_TAV) = 0; // set to 0 to start timing

//RTOSLightTimeFrom = TimerValueGet(TIMER2\_BASE, TIMER\_A); // Capture startTime

} else {

if (RTOSLightSensorHolder[RTOSLightSensorLineCount-1] > RTOSLightSensorHolder[0] \* 0.2 &&

RTOSLightSensorHolder[RTOSLightSensorLineCount-1] < RTOSLightSensorHolder[0] \* 1.5) {

// cross the thin line, write remaining data

RTOSModbusPrintExisting();

RTOSWriteString("\n\rRobot crossed thin line.");

} else if (RTOSLightSensorHolder[RTOSLightSensorLineCount-1] > RTOSLightSensorHolder[0] \* 1.5) {

//RTOSWriteString("\n\rRobot crossed thick line.");

RTOSRunCommand("st");

// output the timing

//RTOSLightTimeTo = TimerValueGet(TIMER2\_BASE, TIMER\_A);

// 1 stick = 25ms

//n tick = n \* 25 [ms]

System\_sprintf(RTOSLightSensorWriteStr, "\n\rRobot crossed thick line.\r\n Total run time: %d [ms].",

RTOSLightTimeCount \* 25); // choose 26 for delay of coding

RTOSWriteString(RTOSLightSensorWriteStr);

} else {

System\_sprintf(RTOSLightSensorWriteStr, "\n\rInvalid range of line: %d%%.",

RTOSLightSensorHolder[RTOSLightSensorLineCount-1] \* 100 / RTOSLightSensorHolder[0] );

RTOSWriteString(RTOSLightSensorWriteStr);

}

}

}

}

else { // robot cross blackline

// LED values - 2=RED, 4=BLUE, 8=GREEN

RTOSFlashingSetColor(4);

if (RTOSLightWhiteLine == true) {

// need to start timer

HWREG(TIMER1\_BASE + TIMER\_O\_TAV) = 0; // set to 0 to start timing

RTOSLightTemp1 = TimerValueGet(TIMER1\_BASE, TIMER\_A); // Capture startTime

RTOSLightWhiteLine = false;

}

}

RTOSLightTimeCount++;

}

void RTOSLightSensorResetTick() {

RTOSLightCount = 0;

}

uint32\_t RTOSLightSensorCount() {

return RTOSLightValue;

}

uint32\_t RTOSLightSensorGet(int16\_t index) {

return RTOSLightSensorHolder[index];

}

int16\_t RTOSLightGetLightCount() {

return RTOSLightSensorLineCount;

}

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

RTOSLightSensor.h :

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

/\*

\* RTOSLightSensor.h

\*

\* Created on: Nov 8, 2021

\* Author: dattr

\*/

#ifndef RTOS\_RTOSLIGHTSENSOR\_H\_

#define RTOS\_RTOSLIGHTSENSOR\_H\_

void RTOSLightSensorInit();

void RTOSLightSensorTick(); // keep increase 1ms when the function is called

void RTOSLightSensorResetTick();

uint32\_t RTOSLightSensorCount();

uint32\_t RTOSLightSensorGet(int16\_t index);

int16\_t RTOSLightGetLightCount();

#endif /\* RTOS\_RTOSLIGHTSENSOR\_H\_ \*/

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

RTOSMasterInclude.h :

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

/\*

\* RTOSMasterInclude.h

\*

\* Created on: Oct 2, 2021

\* Author: dattr

\* NOT RECEOMMEND TO USE THIS FILE, WILL CAUSE SLOW BUILD

\*/

#ifndef RTOS\_RTOSMASTERINCLUDE\_H\_

#define RTOS\_RTOSMASTERINCLUDE\_H\_

/\* Standard libs \*/

#include <stdio.h>

#include <stdint.h>

#include <stdbool.h>

/\* Constants \*/

#include <inc/hw\_ints.h>

#include <inc/hw\_memmap.h>

#include <inc/hw\_types.h>

#include <inc/hw\_gpio.h>

/\* XDC \*/

#include <xdc/runtime/System.h>

#include <xdc/runtime/Log.h>

#include <xdc/cfg/global.h>

#include <ti/sysbios/BIOS.h>

#include <ti/sysbios/knl/Task.h>

#include <ti/sysbios/knl/Clock.h>

#include <ti/sysbios/knl/Semaphore.h>

#include <xdc/std.h>

#include <xdc/runtime/System.h>

/\* TI-RTOS Peripherals \*/

#include <ti/drivers/GPIO.h>

#include <ti/drivers/UART.h>

#include <ti/drivers/uart/UARTTiva.h>

/\* DriverLib \*/

#include <driverlib/sysctl.h>

#include <driverlib/pin\_map.h>

#include <driverlib/gpio.h>

#include <driverlib/uart.h>

#endif /\* RTOS\_RTOSMASTERINCLUDE\_H\_ \*/

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

RTOSModbus.c :

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

/\*

\* RTOSModbus.c

\*

\* Created on: Nov 15, 2021

\* Author: dattr

\*/

#include <stdbool.h>

#include <stdint.h>

#include <stdio.h>

#include <stdlib.h>

#include <RTOS/RTOSBluetooth.h>

char mudbusformat[41]; // store 40 char to print

char modbustouart[60];

uint32\_t modbusping[20];

uint32\_t modbuspong[20];

uint16\_t modbusID; // use team member for this ID

uint16\_t modbusIndex; // store index of ping/pong arr

uint16\_t modbusArrNum; // total number in array

bool isPing; // isPing = true will send modbusping, otherwise

void RTOSModbusInit() {

// initialize dynamic array

modbusArrNum = 20; // store 20 of 2 hex value

// modbusping = (uint32\_t\*) malloc(modbusArrNum \* sizeof(uint32\_t));

// modbuspong = (uint32\_t\*) malloc(modbusArrNum \* sizeof(uint32\_t));

// mudbusformat = (char\*) malloc(2 \* modbusArrNum \* sizeof(char));

mudbusformat[40] = '\0'; // end string

modbusID = 22; // change this one based on our team member

modbusIndex = 0;

isPing = true;

int i = 0;

for(i = 0; i < 41; i++) {

mudbusformat[i] = '\0';

}

}

void RTOSModbusSetID(uint16\_t tm) {

modbusID = tm;

}

// Push data to ping or pong depend on isPing

// return true if the index is reach to max array

bool RTOSModbusPushData(uint32\_t data) {

bool flag = false;

if (isPing) {

modbusping[modbusIndex++] = data;

} else {

modbuspong[modbusIndex++] = data;

}

if(modbusIndex >= modbusArrNum) {

flag = true;

modbusIndex = 0;

isPing = !isPing;

}

return flag;

}

void RTOSModbusPrint() {

// get the string of mudbus format

// start of frame ":", Team number (slave address), 20 bytes of data in ASCII, Team Number (checksum), CR, LF

// store 20 bytes of data into mudbusformat

uint16\_t i, j, k;

uint32\_t\* arr = isPing ? modbuspong : modbusping;

for (i = 0; i < modbusArrNum; i++) {

// [jk] is number of hex

j = (arr[i] / 16) % 15;

k = arr[i] % 15;

// store it into mudbusformat

j = j < 10 ? j + 48 : j + 65 - 10;

k = k < 10 ? k + 48 : k + 65 - 10;

mudbusformat[2\*i] = j;

mudbusformat[2\*i+1] = k;

}

// print it out to input array

sprintf(modbustouart, "\r\n:%d%s%dCRLF", modbusID, mudbusformat, modbusID);

RTOSWriteString(modbustouart);

}

void RTOSModbusPrintExisting() {

uint16\_t i, j, k;

// swap ping vs ping in RTOSModbusPrint()

uint32\_t\* arr = isPing ? modbusping : modbuspong;

for (i = 0; i < modbusArrNum; i++) {

if (i < modbusIndex) {

// [jk] is number of hex

j = (arr[i] / 16) % 15;

k = arr[i] % 15;

} else {

// write 0 since we do not want to write old value

j = k = 0;

}

// store it into mudbusformat

j = j < 10 ? j + 48 : j + 65 - 10;

k = k < 10 ? k + 48 : k + 65 - 10;

mudbusformat[2\*i] = j;

mudbusformat[2\*i+1] = k;

}

// print it out to input array

sprintf(modbustouart, "\r\n:%d%s%dCRLF", modbusID, mudbusformat, modbusID);

RTOSWriteString(modbustouart);

}

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

RTOSModbus.h :

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

/\*

\* RTOSModbus.h

\*

\* Created on: Nov 15, 2021

\* Author: dattr

\*/

#ifndef RTOS\_RTOSMODBUS\_H\_

#define RTOS\_RTOSMODBUS\_H\_

// Use ping pong as requirement

void RTOSModbusInit();

void RTOSModbusSetID(uint16\_t tm);

void RTOSModbusPrint(); // print completed value

void RTOSModbusPrintExisting(); // print existing value, use to print remaining values when robot is stopped

bool RTOSModbusPushData(uint32\_t data);

#endif /\* RTOS\_RTOSMODBUS\_H\_ \*/

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

RTOSMotor.c :

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

/\*

\* RTOSMotor.c

\*

\* Created on: Oct 13, 2021

\* Author: dattr

\*/

#include <stdint.h>

#include <stdbool.h>

#include "inc/hw\_memmap.h"

#include "driverlib/sysctl.h"

#include "driverlib/pwm.h"

#include "driverlib/pin\_map.h"

#include "driverlib/gpio.h"

#include "driverlib/uart.h"

#include "inc/hw\_types.h"

#include "inc/hw\_gpio.h"

#include "inc/hw\_memmap.h"

int pwmMax = 0;

int pwmAdjust = 0;

void RTOSMotorInit(void) {

// using PWM Generator 1

SysCtlPeripheralEnable(SYSCTL\_PERIPH\_PWM1);

SysCtlPWMClockSet(SYSCTL\_PWMDIV\_64);

// using port A and B

SysCtlPeripheralEnable(SYSCTL\_PERIPH\_GPIOA);

SysCtlPeripheralEnable(SYSCTL\_PERIPH\_GPIOB);

// PWM output PA6 and PA7

GPIOPinTypePWM(GPIO\_PORTA\_BASE, GPIO\_PIN\_6 | GPIO\_PIN\_7); //PWM output

//PB2 and PB3 for xPhase, PB6 for mode

GPIOPinTypeGPIOOutput(GPIO\_PORTB\_BASE, GPIO\_PIN\_2 | GPIO\_PIN\_3 | GPIO\_PIN\_6);

GPIOPinConfigure(GPIO\_PA7\_M1PWM3);

GPIOPinConfigure(GPIO\_PA6\_M1PWM2);

GPIOPinWrite(GPIO\_PORTB\_BASE, GPIO\_PIN\_2, 0xFF); // Mode 0x00 to reverse

GPIOPinWrite(GPIO\_PORTB\_BASE, GPIO\_PIN\_3, 0xFF); // Phase

//setxPhase(1);

PWMGenConfigure(PWM1\_BASE, PWM\_GEN\_1, PWM\_GEN\_MODE\_DOWN | PWM\_GEN\_MODE\_NO\_SYNC);

// get PWM clock rate

//uint32\_t PWMclockRate = SysCtlClockGet() / 64 ;

// PWM clock rate / PWM base frequency = pwm max

//pwmMax = (PWMclockRate / 100) - 1;

pwmMax = 11362;

pwmAdjust = 350;

// set period for PWM

PWMGenPeriodSet(PWM1\_BASE, PWM\_GEN\_1, pwmMax);

//PWMGenPeriodSet(PWM1\_BASE, PWM\_GEN\_1, 1000);

PWMPulseWidthSet(PWM1\_BASE, PWM\_OUT\_2, 0); //right

PWMPulseWidthSet(PWM1\_BASE, PWM\_OUT\_3, 0); //left

PWMOutputState(PWM1\_BASE, PWM\_OUT\_2\_BIT, true);

PWMOutputState(PWM1\_BASE, PWM\_OUT\_3\_BIT, true);

PWMGenEnable(PWM1\_BASE, PWM\_GEN\_1);

}

void RTOSMotorDriveRight(int32\_t freq) {

if (freq == 0) {

PWMPulseWidthSet(PWM1\_BASE, PWM\_OUT\_2, 1);

PWMOutputState(PWM1\_BASE, PWM\_OUT\_2\_BIT, false);

return;

}

if (freq < 0) {

GPIOPinWrite(GPIO\_PORTB\_BASE, GPIO\_PIN\_2, 0xFF);// reverse

// in reverse mode, the lower value will be run higher

freq = pwmMax + freq;

}

else {

GPIOPinWrite(GPIO\_PORTB\_BASE, GPIO\_PIN\_2, 0x00);// forward

}

//PWMPulseWidthSet(PWM1\_BASE, PWM\_OUT\_2, abs(freq) \* pwmMax / 1000); // right

PWMPulseWidthSet(PWM1\_BASE, PWM\_OUT\_2, abs(freq)); // right

PWMOutputState(PWM1\_BASE, PWM\_OUT\_2\_BIT, true);

}

void RTOSMotorDriveLeft(int32\_t freq) {

if (freq == 0) {

PWMPulseWidthSet(PWM1\_BASE, PWM\_OUT\_3, 1);

PWMOutputState(PWM1\_BASE, PWM\_OUT\_3\_BIT, false);

return;

}

if (freq < 0) {

GPIOPinWrite(GPIO\_PORTB\_BASE, GPIO\_PIN\_3, 0xFF);// reverse

// in reverse mode, the lower value will be run higher

freq = pwmMax + freq;

}

else {

GPIOPinWrite(GPIO\_PORTB\_BASE, GPIO\_PIN\_3, 0x00);// forward

}

// adjust motor left +300 due to unbalance

freq = abs(freq) + 300;

//PWMPulseWidthSet(PWM1\_BASE, PWM\_OUT\_3, abs(freq) \* pwmMax / 1000); //left

PWMPulseWidthSet(PWM1\_BASE, PWM\_OUT\_3, abs(freq)); // right

PWMOutputState(PWM1\_BASE, PWM\_OUT\_3\_BIT, true);

}

void RTOSMotorStop() { // stop the motor

RTOSMotorDriveRight(500);

RTOSMotorDriveLeft(500);

RTOSMotorDriveRight(1);

RTOSMotorDriveLeft(1);

}

int RTOSMotorGetMaxPWM() {

return pwmMax;

}

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

RTOSMotor.h :

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

/\*

\* RTOSMotor.h

\*

\* Created on: Oct 13, 2021

\* Author: dattr

\*/

#ifndef RTOS\_RTOSMOTOR\_H\_

#define RTOS\_RTOSMOTOR\_H\_

void RTOSMotorInit(void);

// negative for going backward

void RTOSMotorDriveLeft(int32\_t freq);

void RTOSMotorDriveRight(int32\_t freq);

int RTOSMotorGetMaxPWM();

void RTOSMotorStop();

#endif /\* RTOS\_RTOSMOTOR\_H\_ \*/

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

RTOSPID.c :

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

/\*

\* RTOSPID.c

\*

\* Created on: Oct 11, 2021

\* Author: dattr

\*/

#include <stdint.h>

#include <stdbool.h>

#include <stdio.h>

#include <xdc/runtime/System.h>

#include "inc/hw\_types.h"

#include "inc/hw\_memmap.h"

#include "inc/tm4c123gh6pm.h"

#include "inc/hw\_gpio.h"

#include "inc/hw\_timer.h"

#include <ti/sysbios/BIOS.h>

#include <ti/sysbios/knl/Task.h>

#include "driverlib/sysctl.h"// to enable ports

#include "driverlib/interrupt.h"

#include "driverlib/pin\_map.h"

#include "driverlib/gpio.h"

#include "driverlib/timer.h"

#include "driverlib/adc.h"

#include <time.h>

#include <xdc/std.h>

#include <RTOS/RTOSPID.h>

#include <RTOS/RTOSBluetooth.h>

#include <RTOS/RTOSMotor.h>

#include <RTOS/RTOSDistance.h>

#include <RTOS/RTOSLightSensor.h>

#include <RTOS/RTOSModbus.h>

#include <RTOS/RTOSFlashing.h>

int32\_t RTOSPIDSetting[20];

enum PIDEnum {PIDKp, PIDKi, PIDKd, PIDRightOffset, PIDFrontOffset, PIDPWMRight, PIDPWMLeft, PIDPWMMin, PIDPWMMax};

int32\_t PIDPreError, PIDError, PIDtotalerror;

int32\_t temp\_p, temp\_i, temp\_d;

int32\_t temp\_dr, temp\_df; // use to store error of PID

uint32\_t RTOSDistanceFront, RTOSDistanceRight; // store data of distance, unit 10x[mm]

// temp\_df = store distance to front

// temp\_dr = store distance to right

int PIDPWMAdjust;

int8\_t RTOSPIDFlag = 0;

enum PIDStateConst { PIDSTATESTOPALL, PIDSTATEALONGWALL, PIDSTATETURNRIGHT, PIDSTATETURNAROUND };

uint8\_t PIDState; // set the state of robot

uint16\_t iloop = 0;

void RTOSPIDInit(void){

// Init Values of Settings

int i = 0;

for (i = 0; i < 20; i++) {

RTOSPIDSetting[i] = 0;

}

RTOSPIDSetting[PIDPWMMin] = 2000;

RTOSPIDSetting[PIDPWMMax] = 4500; //5500

RTOSPIDSetting[PIDPWMRight] = RTOSPIDSetting[PIDPWMMin];

RTOSPIDSetting[PIDPWMLeft] = RTOSPIDSetting[PIDPWMMin];

RTOSPIDSetting[PIDRightOffset] = 800; // 10x[milimeter]

RTOSPIDSetting[PIDKp] = 6; RTOSPIDSetting[PIDKi] = 1; RTOSPIDSetting[PIDKd] = 3;

// Ki should be smallest, Kd and Ki is smaller than Kp

PIDPreError = PIDError = PIDtotalerror = 0;

RTOSDistanceFront = RTOSDistanceRight = 0;

// default robot state is stopped

PIDState = PIDSTATESTOPALL;

}

// enum PIDEnum {PIDPropotional, PIDIntegral, PIDDerivative, PIDRightOffset, PIDFrontOffset, PIDPWMRight, PIDPWMLeft};

void RTOSPIDGetSettings(char\* s) {

System\_sprintf(s, "\n\rPID values: %d %d %d \n\rRightoffset: %d\n\rFrontOffset: %d \n\rPWM Right: %d\n\rPWM Left: %d\n\rMin PWM: %d \n\rMax PWM: %d\n\rPID Error: %d \n\rPID Adj: %d \n\r",

RTOSPIDSetting[PIDKp], RTOSPIDSetting[PIDKi], RTOSPIDSetting[PIDKd],

RTOSPIDSetting[PIDRightOffset], RTOSPIDSetting[PIDFrontOffset],

RTOSPIDSetting[PIDPWMRight],RTOSPIDSetting[PIDPWMLeft], RTOSPIDSetting[PIDPWMMin], RTOSPIDSetting[PIDPWMMax],

PIDError, PIDPWMAdjust);

}

void RTOSPIDSetSetting(int32\_t type, int32\_t value) {

RTOSPIDSetting[type] = value;

}

void RTOSPIDTuneP(int8\_t v) {

RTOSPIDSetting[PIDKp] += v;

}

void RTOSPIDTuneD(int8\_t v) {

RTOSPIDSetting[PIDKd] += v;

}

void RTOSPIDTuneI(int8\_t v) {

RTOSPIDSetting[PIDKi] += v;

}

void RTOSPIDGet(int\* p, int\* i, int\* d) {

\*p = RTOSPIDSetting[PIDKp];

\*i = RTOSPIDSetting[PIDKi];

\*d = RTOSPIDSetting[PIDKd];

}

// this state for the robot run a long a wall with a distance

void RTOSPIDRunAlongWall() {

// BEGIN PID CONTROL

//allow error within 2mm or 20 [10xmm]

// PIDError = Right - Offset: > 0 when it is far away from wall

if (PIDError > 2) {

// left wheel > right wheel

// choose left wheel max, and right wheel decreasing

RTOSPIDSetting[PIDPWMLeft] = RTOSPIDSetting[PIDPWMMax];

RTOSPIDSetting[PIDPWMRight] -= abs(PIDPWMAdjust);

//RTOSPIDSetting[PIDPWMRight] = RTOSPIDSetting[PIDPWMMin];

}

else if (PIDError < -2) { // PIDError < 0 when it is closed to the wall

// rightwheel > left wheel

// choose right wheel max, adjust left wheel

RTOSPIDSetting[PIDPWMRight] = RTOSPIDSetting[PIDPWMMax];

RTOSPIDSetting[PIDPWMLeft] -= abs(PIDPWMAdjust);

//RTOSPIDSetting[PIDPWMLeft] = RTOSPIDSetting[PIDPWMMin];

}

// END PID CONTROL

// range is between PWM Min and PWM Max

// RTOSPIDSetting[PIDPWMMin], RTOSPIDSetting[PIDPWMMax]

if (RTOSPIDSetting[PIDPWMRight] > RTOSPIDSetting[PIDPWMMax]) RTOSPIDSetting[PIDPWMRight] = RTOSPIDSetting[PIDPWMMax];

if (RTOSPIDSetting[PIDPWMRight] < RTOSPIDSetting[PIDPWMMin]) RTOSPIDSetting[PIDPWMRight] = RTOSPIDSetting[PIDPWMMin];

if (RTOSPIDSetting[PIDPWMLeft] > RTOSPIDSetting[PIDPWMMax]) RTOSPIDSetting[PIDPWMLeft] = RTOSPIDSetting[PIDPWMMax];

if (RTOSPIDSetting[PIDPWMLeft] < RTOSPIDSetting[PIDPWMMin]) RTOSPIDSetting[PIDPWMLeft] = RTOSPIDSetting[PIDPWMMin];

} // end RTOSPIDRunAlongWall

// the robot will make a right turn

void RTOSPIDRightTurn() {

RTOSPIDSetting[PIDPWMRight] = 800;

RTOSPIDSetting[PIDPWMLeft] = 8500;

} // end RTOSPIDRightTurn

// the robot keep turning left until find a path (u-turn or turn left)

// return true when the new path is found

bool RTOSPIDTurnAround() {

// perform turn left

// slowly uturn until front greater than 2000 (20cm)

// use d ml and d mr to find out the value for the motors

RTOSPIDSetting[PIDPWMRight] = 2000;

RTOSPIDSetting[PIDPWMLeft] = -6000;

if (RTOSDistanceFront > 1500 && RTOSDistanceRight > 500) {

// new path is found

return true;

}

return false;

} // end RTOSPIDTurnAround

void RTOSPIDCompute() {

// get distance sensor

RTOSDistanceGet(&RTOSDistanceFront, &RTOSDistanceRight);

PIDError = RTOSDistanceRight - RTOSPIDSetting[PIDRightOffset]; // 10x[mm]

// PIDError < 0: temp\_dr < PIDRightOffset: when it is far away from the wall

// PIDError > 0: temp\_dr > PIDRightOffset: when it is too closed from the wall

//PIDtotalerror += PIDError;

temp\_p = RTOSPIDSetting[PIDKp] \* PIDError; // negative when far away

temp\_i = RTOSPIDSetting[PIDKi] \* (PIDError + PIDPreError);

temp\_d = RTOSPIDSetting[PIDKd] \* (PIDError - PIDPreError);

PIDPWMAdjust = (temp\_p + temp\_i + temp\_d);

// send error to modbus

if(RTOSLightGetLightCount() == 1) { // only send data when the first line is crossed

// LED values - 2=RED, 4=BLUE, 8=GREEN

if (iloop >= 20) {

// recalculate every 5ms

// get error data each 100ms => 100 / 5 = 20

iloop = 0; // reset the loop

if (RTOSModbusPushData(abs(PIDError/10))) { // /10 to get mm

RTOSFlashingSetColor(8);

RTOSModbusPrint();

}

}

}

// update Error

PIDPreError = PIDError;

// SET STATE OF ROBOT

//enum PIDStateConst { PIDSTATESTOPALL, PIDSTATEALONGWALL, PIDSTATETURNRIGHT, PIDSTATETURNAROUND };

switch (PIDState) {

case PIDSTATEALONGWALL:

// robot use the right sensor to keep a constant distance to the wall

RTOSPIDRunAlongWall();

break;

case PIDSTATETURNRIGHT:

// when sensor detects a possible way to turn right, robot will make a turn right

RTOSPIDRightTurn();

break;

case PIDSTATETURNAROUND:

// robot keep turning left to detect clear path

if(RTOSPIDTurnAround())

PIDState = PIDSTATEALONGWALL;

else

return;

// new path still not found, return to prevent the state change

break;

default:

break;

}

// change state

if (RTOSDistanceFront < 1000 && RTOSDistanceRight < 1200) // 10\*60m need to stop

{

RTOSMotorStop();

PIDState = PIDSTATETURNAROUND;

}

else if (RTOSDistanceRight > 1200) {

// change to right turn mode when right distance > 15cm

PIDState = PIDSTATETURNRIGHT;

}

else {

PIDState = PIDSTATEALONGWALL;

}

iloop++; // increase to count number of loop

} // end RTOSPIDCompute

// Use to enable the robot

void RTOSPIDPIDEnable(int8\_t flag) {

if (flag == 1) {

RTOSPIDSetting[PIDPWMRight] = 3000;

RTOSPIDSetting[PIDPWMLeft] = 3000;

PIDState = PIDSTATEALONGWALL;

} else {

// stop the robot

PIDState = PIDSTATESTOPALL;

RTOSMotorStop();

}

} // end RTOSPIDPIDEnable

void RTOSPIDRun() {

// clear timer interrupt

TimerIntClear(TIMER3\_BASE, TIMER\_TIMA\_TIMEOUT);

if (PIDState != PIDSTATESTOPALL) {

RTOSPIDCompute();

RTOSMotorDriveRight(RTOSPIDSetting[PIDPWMRight]);

RTOSMotorDriveLeft(RTOSPIDSetting[PIDPWMLeft]);

}

}

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

RTOSPID.h :

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

/\*

\* RTOSPID.h

\*

\* Created on: Oct 11, 2021

\* Author: dattr

\* This file will read distance sensors and use PID to control the robot

\* Hwi will be used on this one

\*/

#ifndef RTOS\_RTOSPID\_H\_

#define RTOS\_RTOSPID\_H\_

#include <xdc/std.h>

void RTOSPIDInit(void);

// Get raw data from Distance sensor on the front and on the right

// We can update the calibration distance by using SetSetting and enum

//void RTOSPIDGetDistance(uint32\_t\* front, uint32\_t\* right);

void RTOSPIDGetSettings(char\* s);

void RTOSPIDSetSetting(int32\_t type, int32\_t value);

void RTOSPIDCompute();

void RTOSPID\_ClockHandler(UArg arg);

void RTOSPIDRun();

// Use this one to add +1 when plus = true and -1 when plus = false

void RTOSPIDTuneP(int8\_t v);

void RTOSPIDTuneD(int8\_t v);

void RTOSPIDTuneI(int8\_t v);

void RTOSPIDGet(int\* p, int\* i, int\* d);

void RTOSPIDPIDEnable(int8\_t flag); // enable pid by set flag = 1

//enum PIDLogEnum {iKpLog, iKiLog, iKdLog, iKadjust, iPWMLeftLog, iPWMRightLog};

//void RTOSPIDLog(int32\_t kp, int32\_t ki, int32\_t kd,int32\_t kadjust,int32\_t pwmleft,int32\_t pwmright); // log infomation of pid changing

void RTOSPIDRunAlongWall(); // this state for the robot run a long a wall with a distance

void RTOSPIDRightTurn(); // the robot will make a right turn

bool RTOSPIDTurnAround(); // the robot keep turning left until find a path (u-turn or turn left)

// return true when the new path is found

#endif /\* RTOS\_RTOSPID\_H\_ \*/