```
/*
 * log.c
 *
 *  Created on: Apr 8, 2019
 *      Author: Steve Antony
 */

/*********************************************
 *        Includes
 *********************************************/
#include "log.h"


/*********************************************
 *        Global definitions
 *********************************************/
//receive log data from other tasks on remote node
char log_data_recv[100];

//structure to be transmitted from remote node to control node
typedef struct
{
    char task[5];
    uint32_t time_stamp;
    float distance;
    float lux;
    uint32_t water;
    int8_t mode_RN;
    int8_t Deg_mode;

}send_sensor_data;


send_sensor_data tx_data;


//to receive sensor values from various tasks to logger tasks
float lux_recv, distance_recv;
uint32_t Water_level_recv;
int8_t beat_recv;

//initiating the queues
void queue_init()
{
    myQueue_light = xQueueCreate(QueueLength, sizeof(float));
    if(myQueue_light == NULL)
    {
        UARTprintf("error on queue creation myQueue_light\n");
    }

    myQueue_ultra = xQueueCreate(QueueLength, sizeof(float));
    if(myQueue_ultra == NULL)
    {
        UARTprintf("error on queue creation myQueue_ultra\n");
```

```
    }

    myQueue_water = xQueueCreate(QueueLength, sizeof(uint32_t));
    if(myQueue_water == NULL)
    {
        UARTprintf("error on queue creation myQueue_water\n");
    }

    myQueue_log = xQueueCreate(QueueLength, 100);
    if(myQueue_log == NULL)
    {
        UARTprintf("error on queue creation myQueue_ultra\n");
    }

    myQueue_heartbeat = xQueueCreate(QueueLength, sizeof(int8_t));
    if(myQueue_heartbeat == NULL)
    {
        UARTprintf("error on queue creation myQueue_heartbeat\n");
    }

}

/*******************************************
 *        Logger thread
 *******************************************/
void LogTask(void *pvParameters)
{

    char buffer[50];
    unsigned char *ptr;
    ptr = (uint8_t *) (&tx_data);

    unsigned char *ptr1;
        ptr1 = (uint8_t *) (log_data_recv);




    for(;;)
    {
        //receive lux sensor value lux task
        if(xQueueReceive(myQueue_light, &lux_recv, 0 ) == pdTRUE )
        {
            strcpy(tx_data.task,"LUX");
            tx_data.lux = lux_recv;
            tx_data.time_stamp = xTaskGetTickCount();
            tx_data.mode_RN = mode;
            tx_data.Deg_mode = DEGRADED_MODE_MANUAL;
            UART_send(ptr, sizeof(tx_data));

        }

        //receive ultrasonic sensor value ultrasonic task
        if(xQueueReceive(myQueue_ultra, &distance_recv, 0 ) == pdTRUE )
```

```
        {
            strcpy(tx_data.task,"DIST");
            tx_data.distance = distance_recv;
            tx_data.time_stamp = xTaskGetTickCount();
            tx_data.mode_RN = mode;
            tx_data.Deg_mode = DEGRADED_MODE_MANUAL;
            UART_send(ptr, sizeof(tx_data));

        }

        //receive water level sensor value water level task
        if(xQueueReceive(myQueue_water, &Water_level_recv, 0 ) == pdTRUE
)
        {
            strcpy(tx_data.task,"WAT");
            tx_data.water = Water_level_recv;
            tx_data.time_stamp = xTaskGetTickCount();
            tx_data.mode_RN = mode;
            tx_data.Deg_mode = DEGRADED_MODE_MANUAL;

            UART_send(ptr, sizeof(tx_data));

        }

        //receive heartbeat from heartbeat task
        if(xQueueReceive(myQueue_heartbeat, &beat_recv, 0 ) == pdTRUE )
        {
            strcpy(tx_data.task,"BEA");
            tx_data.mode_RN = mode;
            tx_data.Deg_mode = DEGRADED_MODE_MANUAL;
            tx_data.time_stamp = xTaskGetTickCount();
            UART_send(ptr, sizeof(tx_data));

        }

        //receive log data from various tasks
        memset(log_data_recv,'\0',sizeof(log_data_recv));
        if(xQueueReceive(myQueue_log, log_data_recv, 0 ) == pdTRUE )
        {

//            UARTprintf("--> Log  %s\n",log_data_recv);
            if(CN_ACTIVE == pdTRUE)
            {
                UART_send_log(ptr1, strlen(log_data_recv));
            }


        }
    }
}

/*Uart function to sensor data to the control node*/
void UART_send(char* ptr, int len)
{
```

```
    while(len != 0)
    {
        UARTCharPut(UART2_BASE, *ptr);
        ptr++;
        len--;
    }
}

/*Uart function to logger data to the control node*/
void UART_send_log(char* ptr, int len)
{
    while(len != 0)
    {
        UARTCharPut(UART3_BASE, *ptr);
        ptr++;
        len--;
    }
}

/* FreeRTOS 8.2 Tiva Demo
 *
 * main.c
 *
 * Steve Antony
 *
 * This is a simple demonstration project of FreeRTOS 8.2 on the Tiva
Launchpad
 * EK-TM4C1294XL.  TivaWare driverlib sourcecode is included.
 */

#include <stdint.h>
#include <stdbool.h>
#include <string.h>
#include <stdio.h>
#include "main.h"
#include "drivers/pinout.h"
#include "utils/uartstdio.h"


// TivaWare includes
#include "driverlib/sysctl.h"
#include "driverlib/debug.h"
#include "driverlib/rom.h"
#include "driverlib/rom_map.h"

// FreeRTOS includes
#include "FreeRTOSConfig.h"
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#include "timers.h"
#include "driverlib/gpio.h"
#include "driverlib/inc/hw_memmap.h"
#include "log.h"
```

```
#include "object_detection.h"
#include "uart.h"
#include "interrupt.h"
#include "rom.h"
#include "driverlib/fpu.h"
#include "motor_driver.h"
#include "heartbeat.h"
#include "waterlevel.h"
#include "semphr.h"

/*********************************************
 *               Tasks
 *********************************************/

/*********************************************
 *         Actuator task
 * Description : Controls the autonomous movement
 *                of the robot in auto mode
 *********************************************/
void Actuator_motor(void *pvParameters);

/*********************************************
 *         ReadUart task
 * Description : This tasks reads the control
 *                data which the Control node sends
 *********************************************/
void ReadUartTask(void *pvParameters);

/*********************************************
 *        Globals
 *********************************************/
// for queues that sends data from different tasks to the logger tasks
QueueHandle_t myQueue_ultra, myQueue_light, myQueue_log, myQueue_water,
myQueue_heartbeat;

//output clock
uint32_t output_clock_rate_hz;

//For object detection notification and heartbeat notification to get
pulses from control node
TaskHandle_t handle_motor,handle_heartbeat;

// flag to start only once based when lux is very low
static uint8_t start_again = 1;

//Flag set when the threads are not created properly
uint8_t STARTUP_FAILED = 0;

/*Sets application mode
 * mode 0  - Auto mode
 * mode 1  - Manual mode
 */
int8_t mode=0; //auto mode on default
```

```
//temporary buffer for logger
char temp_buffer[100];

/*mutex to avoid race condition when many tasks use
 * the same queue for logging
 */

SemaphoreHandle_t xSemaphore;

/*******************************************
 *        Main Function
 *******************************************/
int main(void)
{

    // Initialize system clock to 120 MHz
    output_clock_rate_hz = ROM_SysCtlClockFreqSet(
                               (SYSCTL_XTAL_25MHZ | SYSCTL_OSC_MAIN |
                                SYSCTL_USE_PLL | SYSCTL_CFG_VCO_480),
                               SYSTEM_CLOCK);
    ASSERT(output_clock_rate_hz == SYSTEM_CLOCK);

    // Initialize the GPIO pins for the Launchpad
    PinoutSet(false, false);
    FPUEnable();

    // Set up the UART which is connected to the virtual COM port
    UARTStdioConfig(0, 115200, SYSTEM_CLOCK);

    //initiating message queues for communication between various tasks
and logger
    queue_init();

    //initiating the semaphore
    xSemaphore = xSemaphoreCreateMutex();

    //configures the uarts UART1, UART2, UART3
    ConfigureUART1();
    ConfigureUART2();
    ConfigureUART3();

    //initiating the motor pins
    init_motor();

    // Create logger task
    if(pdPASS != xTaskCreate(LogTask, (const portCHAR *)"Log",
                    configMINIMAL_STACK_SIZE, NULL, 1, NULL))
   {
       STARTUP_FAILED = pdTRUE;
       LOG_ERROR("Thread creation failed for LogTask\n")
   }


    // Create light task
```

```
   if(pdPASS != xTaskCreate(LightTask, (const portCHAR *)"Light",
              configMINIMAL_STACK_SIZE, NULL, 1, NULL))
   {
       STARTUP_FAILED = pdTRUE;
       LOG_ERROR("Thread creation failed for LightTask\n")
   }


    // Create ultrasonic task
    if(pdPASS != xTaskCreate(UtrasonicTask, (const portCHAR
*)"ultrasonic",
                        configMINIMAL_STACK_SIZE, NULL, 1, NULL))
    {
        STARTUP_FAILED = pdTRUE;
        DEGRADED_MODE_MANUAL = 1;
        LOG_ERROR("Thread creation failed for UtrasonicTask\n")
    }

    // Create uart task for reading control data from control node
    if(pdPASS != xTaskCreate(ReadUartTask, (const portCHAR *)"UART",
                          configMINIMAL_STACK_SIZE, NULL, 1, NULL))
    {
        STARTUP_FAILED = pdTRUE;
        LOG_ERROR("Thread creation failed for ReadUartTask\n")
    }

    // Create motor actuator task
    if(pdPASS != xTaskCreate(Actuator_motor, (const portCHAR *)"motion",
                                  configMINIMAL_STACK_SIZE, NULL, 1,
&handle_motor))
    {
        STARTUP_FAILED = pdTRUE;
        LOG_ERROR("Thread creation failed for Actuator_motor\n")
    }

    // Create heartbeat task
    if(pdPASS != xTaskCreate(Control_Node_heartbeat, (const portCHAR
*)"heartbeat",
                                         configMINIMAL_STACK_SIZE, NULL, 1,
&handle_heartbeat))
    {
        STARTUP_FAILED = pdTRUE;
        LOG_ERROR("Thread creation failed for Control_Node_heartbeat\n")
    }

    // Create water level task
    if(pdPASS != xTaskCreate(Water_level, (const portCHAR *)"waterlevel",
                                              configMINIMAL_STACK_SIZE,
NULL, 1, NULL))
    {
        STARTUP_FAILED = pdTRUE;
        LOG_ERROR("Thread creation failed for Water_level\n")
    }
```

```
    //Checks if the threads were created successfully
    if(STARTUP_FAILED == pdTRUE)
    {
        LOG_ERROR("Startup test failed in creating tasks\n")
    }


    /*start the schedule*/
    vTaskStartScheduler();


    return 0;

}

/*Task to receive control data from control node*/
void ReadUartTask(void *pvParameters)
{
    for(;;)
        {
            while(UARTCharsAvail(UART1_BASE))
            {
                char c = ROM_UARTCharGet(UART1_BASE);
                UARTprintf("-> %c\n",c);
                if(c == 'h') //heartbeat
                {
                    xTaskNotifyGive(handle_heartbeat);
                }

                else if((c == '1') && (mode == 0) &&
(DEGRADED_MODE_MANUAL == 0))//object detected in auto mode
                {
                    xTaskNotifyGive(handle_motor);
                }

                else if(c == '2')//Water level low
                {
                    close_value();
                }

                else if(c == '3')//Water level high
                {
                    open_value();
                }
                else if(c == '4')//auto start - lux
                {
                    if((start_again == 1) && (mode == 0) &&
(DEGRADED_MODE_MANUAL == 0))
                    {
                        UARTprintf("CN: Auto on of robot\n");
                        LOG_INFO("Auto on of robot\n")
                        forward();
                        start_again = 0;
                    }
```

```
}

else if(c == 'm') //manual mode
{
    UARTprintf("CN: Manual mode\n");
    LOG_INFO("Switched to Manual mode\n")
    mode = 1 ;
    stop();

}
else if(c == 'a') //auto mode
{
    if((DEGRADED_MODE_MANUAL == 0))
    {
        UARTprintf("CN: Auto mode\n");
        LOG_INFO("Switched to Auto mode\n")
        mode = 0 ;
    }

}

else if(c == 'u') //forward
{
    if(mode == 1)
    {
        UARTprintf("CN: forward\n");
        forward();
    }

}
else if(c == 's') //stop
{
   stop();
   start_again = 1;
   UARTprintf("CN: stop\n");
}

else if(c == 'l') //left
{
    if(mode == 1)
    {
        left();
        UARTprintf("CN: left\n");
        vTaskDelay(300/portTICK_PERIOD_MS);
        stop();

    }

}
else if(c == 'r') //right
{
    if(mode == 1)
    {
        right();
```

```
                        UARTprintf("CN: right\n");
                        vTaskDelay(300/portTICK_PERIOD_MS);
                        stop();

                    }

                }
                else if(c == 'b') //back
                {
                    if(mode == 1)
                    {
                       backward();
                       UARTprintf("CN: back\n");

                    }

                }
                else if(c == 'o') //force start from phone
                {
                    if((DEGRADED_MODE_MANUAL == 0))
                    {
                        mode = 0 ;
                        forward();
                        UARTprintf("CN: force turn on\n");
                        LOG_INFO("force turn on from phone\n")

                    }

                }
            }
    }

}

/*Actuator task to control motors when an object is detected*/
void Actuator_motor(void *pvParameters)
{

        for(;;)
        {

            uint32_t ulNotifiedValue = 0;

            ulNotifiedValue  = ulTaskNotifyTake( pdTRUE, 0  );
            if(ulNotifiedValue > 0)
            {

                UARTprintf("Object detected notified\n");
                LOG_INFO("Object detected\n")
                backward();


                 vTaskDelay(1000/portTICK_PERIOD_MS);
```

```
                //normal run of motors
                right();

                vTaskDelay(500/portTICK_PERIOD_MS);

                forward();

            }

        }
}


/*  ASSERT() Error function
 *
 *  failed ASSERTS() from driverlib/debug.h are executed in this function
 */
void __error__(char *pcFilename, uint32_t ui32Line)
{
    // Place a breakpoint here to capture errors until logging routine is
finished
    while (1)
    {
    }
}



/*Uart to transmit sensor data to the control node*/
//Transmit data on PA7
void ConfigureUART2(void)
{
        ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);    //Enable GPIO

        ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_UART2);    //Enable
UART0

        ROM_GPIOPinConfigure(GPIO_PA6_U2RX);                //Configure
UART pins
        ROM_GPIOPinConfigure(GPIO_PA7_U2TX);
        ROM_GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_6 | GPIO_PIN_7);

        ROM_UARTConfigSetExpClk(UART2_BASE, output_clock_rate_hz, 115200,
                                    (UART_CONFIG_WLEN_8 |
UART_CONFIG_STOP_ONE |
                                                        UART_CONFIG_PAR_NONE));

        UARTprintf("configured 2\n");

}
```

```
/*Uart to receive control data from the control node*/
//UART1 recv on PB0
void ConfigureUART1()
{

        ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB);    //Enable GPIO

        ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_UART1);    //Enable
UART0

        ROM_GPIOPinConfigure(GPIO_PB0_U1RX);                //Configure
UART pins
        ROM_GPIOPinConfigure(GPIO_PB1_U1TX);
        ROM_GPIOPinTypeUART(GPIO_PORTB_BASE, GPIO_PIN_0 | GPIO_PIN_1);


        ROM_UARTConfigSetExpClk(UART1_BASE, output_clock_rate_hz, 115200,
                                    (UART_CONFIG_WLEN_8 |
UART_CONFIG_STOP_ONE |
                                                UART_CONFIG_PAR_NONE));

        UARTprintf("configured 1\n");




}

//logger send/*Uart to transmit log data to the control node*/
//UART3 tx on PA5
void ConfigureUART3()
{

        ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);    //Enable GPIO

        ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_UART3);    //Enable
UART3

        ROM_GPIOPinConfigure(GPIO_PA5_U3TX);                //Configure
UART pins
        ROM_GPIOPinConfigure(GPIO_PA4_U3RX);
        ROM_GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_5 | GPIO_PIN_4);


        ROM_UARTConfigSetExpClk(UART3_BASE, output_clock_rate_hz, 115200,
                                    (UART_CONFIG_WLEN_8 |
UART_CONFIG_STOP_ONE |
                                                UART_CONFIG_PAR_NONE));

        UARTprintf("configured 3\n");
```

```
}



/*
 * motor_driver.c
 *
 *  Created on: Apr 14, 2019
 *      Author: Steve Antony
 */

#include "motor_driver.h"


void init_motor()
{
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOE);

    GPIOPinTypeGPIOOutput(GPIO_PORTE_BASE, GPIO_PIN_0);

GPIOPadConfigSet(GPIO_PORTE_BASE,GPIO_PIN_0,GPIO_STRENGTH_2MA,GPIO_PIN_TY
PE_STD_WPU);

    GPIOPinTypeGPIOOutput(GPIO_PORTE_BASE, GPIO_PIN_1);

GPIOPadConfigSet(GPIO_PORTE_BASE,GPIO_PIN_1,GPIO_STRENGTH_2MA,GPIO_PIN_TY
PE_STD_WPU);

    GPIOPinTypeGPIOOutput(GPIO_PORTE_BASE, GPIO_PIN_2);

GPIOPadConfigSet(GPIO_PORTE_BASE,GPIO_PIN_2,GPIO_STRENGTH_2MA,GPIO_PIN_TY
PE_STD_WPU);


    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOC);
    GPIOPinTypeGPIOOutput(GPIO_PORTC_BASE, GPIO_PIN_7);

GPIOPadConfigSet(GPIO_PORTC_BASE,GPIO_PIN_7,GPIO_STRENGTH_2MA,GPIO_PIN_TY
PE_STD_WPU);

}

void stop()
{
    GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_0, 0);
    GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_1, 0);

    GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_2, 0);
    GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_7, 0);
}

void forward()
{
    GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_0, GPIO_PIN_0);
```

```
    GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_1, 0);

    GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_2, GPIO_PIN_2);
    GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_7, 0);
}

void backward()
{
    GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_0, 0);
    GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_1, GPIO_PIN_1);

    GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_2, 0);
    GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_7, GPIO_PIN_7);
}

void right()
{
    GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_0, GPIO_PIN_0);
    GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_1, 0);

    GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_2, 0);
    GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_7, 0);
}

void left()
{
    GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_0, 0);
    GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_1, 0);

    GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_2, GPIO_PIN_2);
    GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_7, 0);
}
/*********************************************
 *               Includes
 *********************************************/
#include "heartbeat.h"

/*********************************************
 *               Globals
 *********************************************/
//timer flag to check the heartbeat after regular intervals
int FLAG_HB = 0;

//flag is set if the control node is active
int CN_ACTIVE = 0;

//for sending heartbeat from Remote node to control node
int8_t BEAT = 1;

//storing pulses to find heartbeat
static uint32_t Pulse = 0, Prev_pulse = 0;

//temporary buffer for logger
char temp_buffer[100];
```

```
/*
-------------------------------------------------------------------------
-------------------
               Control_Node_heartbeat task
-------------------------------------------------------------------------
-------------------
*   This Task sends heartbeat continuosly from remote node to control
node and
*   Checks if the control node is active
*
*/
void Control_Node_heartbeat(void *pvParameters)
{
    UARTprintf("Created heartbeat task\n");
    long x_heartbeat_id = 1019;
   xTimerHandle xTimer_HB;
   xTimer_HB = xTimerCreate("Heart_beat",              // Just a text
name, not used by the kernel.
                            pdMS_TO_TICKS( 1000 ),     // 100ms
                            pdTRUE,                   // The timers will
auto-reload themselves when they expire.
                            ( void * ) x_heartbeat_id,      // Assign
each timer a unique id equal to its array index.
                            vTimerCallback_HB_handler// Each timer calls
the same callback when it expires.
                           );

    if( xTimer_HB == NULL )
    {
       // The timer was not created.
       UARTprintf("Error on HB timer creation\n");
    }

    xTimerStart( xTimer_HB, 0 );
    for(;;)
    {
        uint32_t ulNotifiedValue = 0;


        //notified when heartbeat is received from control node
        ulNotifiedValue  = ulTaskNotifyTake( pdTRUE, 0  );
        if(ulNotifiedValue > 0)
        {

            Pulse++;
        }

        if(FLAG_HB)
        {
            FLAG_HB = pdFALSE;

            //checks if there was a pulse received
            if(Pulse <= Prev_pulse)
```

```
            {
               // UARTprintf("Control node dead Pr %d P %d\n",Prev_pulse,
Pulse);
                CN_ACTIVE = pdFALSE;
            }
            else
            {
               // UARTprintf("Control node active Pr %d P
%d\n",Prev_pulse, Pulse);
                CN_ACTIVE = pdTRUE;
            }

            Prev_pulse = Pulse;

            //send pulse from remote node to control node
            xQueueSendToBack( myQueue_heartbeat,( void * ) &BEAT,
QUEUE_TIMEOUT_TICKS ) ;

            //turn off the remote node leds when the control node is
active
            if(CN_ACTIVE)
            {
                GPIOPinWrite(CLP_D1_PORT, CLP_D1_PIN, 0);
                GPIOPinWrite(CLP_D2_PORT, CLP_D2_PIN, 0);
                GPIOPinWrite(CLP_D3_PORT, CLP_D3_PIN, 0);
                GPIOPinWrite(CLP_D4_PORT, CLP_D4_PIN, 0);

            }
            //turn on the remote node leds when the control node is
active
            else
            {
                GPIOPinWrite(CLP_D1_PORT, CLP_D1_PIN, CLP_D1_PIN);
                GPIOPinWrite(CLP_D2_PORT, CLP_D2_PIN, CLP_D2_PIN);
                GPIOPinWrite(CLP_D3_PORT, CLP_D3_PIN, CLP_D3_PIN);
                GPIOPinWrite(CLP_D4_PORT, CLP_D4_PIN, CLP_D4_PIN);

                //move to fail safe mode from degraded mode when the
ultrasonic sensor is dead and control node inactive
                if((DEGRADED_MODE_MANUAL == 1))
                {
                    stop();
                    UARTprintf("System shutdown as no ultrasonic sensor
and no control node - Fail safe\n");
                }
            }


        }

    }

}
```

```
/*Heartbeat Timer handler*/
void vTimerCallback_HB_handler( TimerHandle_t  *pxTimer )
{
    FLAG_HB = pdTRUE;
}

/*
 * water_level.c
 *
 *  Created on: Apr 24, 2019
 *      Author: Steve
 */

/*********************************************
 *        Includes
 *********************************************/
#include "waterlevel.h"

/*********************************************
 *        Globals
 *********************************************/
int FLAG_WL = 0;
static char buffer_log[BUFFER];
char temp_buffer[100];

/*water level task*/
void Water_level(void *pvParameters)
{
    vTaskDelay(1000/portTICK_PERIOD_MS);
    UARTprintf("Water level task\n");
    uint32_t Water_level_data;

    init_valve();

    SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC0);

    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOE);

    GPIOPinTypeADC(GPIO_PORTE_BASE, GPIO_PIN_3);

    ADCSequenceConfigure(ADC0_BASE, 3, ADC_TRIGGER_PROCESSOR, 0);

    ADCSequenceStepConfigure(ADC0_BASE, 3, 0, ADC_CTL_CH0 | ADC_CTL_IE |
                                ADC_CTL_END);

    ADCSequenceEnable(ADC0_BASE, 3);

    ADCIntClear(ADC0_BASE, 3);

    long x_WaterL_id = 1009;
    xTimerHandle xTimer_WL;
    xTimer_WL = xTimerCreate("Waterlevel_timer",               // Just a
text name, not used by the kernel.
                                pdMS_TO_TICKS( 2000 ),     // 1000ms
```

```
                                pdTRUE,                     // The timers
will auto-reload themselves when they expire.
                                ( void * ) x_WaterL_id,      // Assign
each timer a unique id equal to its array index.
                                vTimerCallback_WaterLevel_handler// Each
timer calls the same callback when it expires.
                               );



        if( (xTimer_WL == NULL ) )
        {
            // The timer was not created.
            UARTprintf("Error on timer creation - xTimer_WL\n");
        }

    /*start the timer*/
    xTimerStart( xTimer_WL, 0 );


    /*start up test*/
    ADCProcessorTrigger(ADC0_BASE, 3);


    while(!ADCIntStatus(ADC0_BASE, 3, false))
       {
       }
    ADCIntClear(ADC0_BASE, 3);

    ADCSequenceDataGet(ADC0_BASE, 3, &Water_level_data);

    //kill if the startup fails
    if(Water_level_data > 3000)
    {
        UARTprintf("Startup test failed for water level sensor WL %d\n",
Water_level_data);
         LOG_ERROR("Killed water level sensor task - Startup failed\n")
         vTaskDelete( NULL );
    }



    while(1)
    {
        if(FLAG_WL)
        {
            ADCProcessorTrigger(ADC0_BASE, 3);


            while(!ADCIntStatus(ADC0_BASE, 3, false))
               {
               }
            ADCIntClear(ADC0_BASE, 3);
```

```
            ADCSequenceDataGet(ADC0_BASE, 3, &Water_level_data);


            if(CN_ACTIVE)
             {
                 xQueueSendToBack( myQueue_water,( void * )
&Water_level_data, QUEUE_TIMEOUT_TICKS ) ;
                 memset(buffer_log,'\0',BUFFER);
                 sprintf(buffer_log,"W %d\n",Water_level_data);
                 LOG_INFO(buffer_log)
             }


            FLAG_WL = pdFALSE;

        }
    }
}
void vTimerCallback_WaterLevel_handler( TimerHandle_t  *pxTimer )
{
    FLAG_WL = pdTRUE;
}

void init_valve()
{
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOK);

    GPIOPinTypeGPIOOutput(GPIO_PORTK_BASE, GPIO_PIN_7);

GPIOPadConfigSet(GPIO_PORTK_BASE,GPIO_PIN_7,GPIO_STRENGTH_2MA,GPIO_PIN_TY
PE_STD_WPU);
}
void open_value()
{
    GPIOPinWrite(GPIO_PORTK_BASE, GPIO_PIN_7, GPIO_PIN_7);
    UARTprintf("CN: Valve opened\n");
    LOG_INFO("Valve opened")
}

void close_value()
{
    GPIOPinWrite(GPIO_PORTK_BASE, GPIO_PIN_7, 0);
    UARTprintf("CN: Valve closed\n");
    LOG_INFO("Valve closed")
}
/*
 * object_detection.c
 *
 *  Created on: Apr 15, 2019
 *      Author: Steve Antony
 */

/***********************************************
 *        Includes
```

```
 *********************************************/

#include "object_detection.h"


/*********************************************
 *        GLOBALS
 *********************************************/
//to find the pulse duration
uint32_t start, end;

//conversion complete flag
uint32_t FLAG_UL, conv_complete = 0;

//find the duration of echo on pulse
float time_pulse = 0;

//get the distance
float distance_send;

//for local logger
static char buffer_log[BUFFER];

//flag to indicate if the sensor is dead
uint32_t ULT_DEAD = 0;

//indicate degraded mode
uint32_t DEGRADED_MODE_MANUAL = 0;

//mutex for log
SemaphoreHandle_t xSemaphore;

//local logger buffer
char temp_buffer[100];


void init_ultrasonic_sensor()
{
    SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER2);
    TimerConfigure(TIMER2_BASE, TIMER_CFG_PERIODIC_UP);


    //echo pin
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
    GPIOPinTypeGPIOInput(GPIO_PORTF_BASE, GPIO_PIN_3);

//GPIOPadConfigSet(GPIO_PORTF_BASE,GPIO_PIN_3,GPIO_STRENGTH_2MA,GPIO_PIN_
TYPE_STD_WPU);

    GPIOIntEnable(GPIO_PORTF_BASE, GPIO_INT_PIN_3);

    GPIOIntTypeSet(GPIO_PORTF_BASE,GPIO_PIN_3,GPIO_BOTH_EDGES );

    GPIOIntRegister(GPIO_PORTF_BASE,PortFIntHandler);
```

```
    GPIOIntClear(GPIO_PORTF_BASE, GPIO_INT_PIN_3);



    //trigger pin
    GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_1);

GPIOPadConfigSet(GPIO_PORTF_BASE,GPIO_PIN_1,GPIO_STRENGTH_2MA,GPIO_PIN_TY
PE_STD_WPU);

    UARTprintf("configured ultrasonic\n");

}


void find_object()
{
    GPIOPinWrite(GPIO_PORTF_BASE,GPIO_PIN_1, 0);
    vTaskDelay(pdMS_TO_TICKS( 1 ));
    GPIOPinWrite(GPIO_PORTF_BASE,GPIO_PIN_1, GPIO_PIN_1);
    vTaskDelay(pdMS_TO_TICKS( 10 ));
    GPIOPinWrite(GPIO_PORTF_BASE,GPIO_PIN_1, 0);
}


void PortFIntHandler()
{


    taskENTER_CRITICAL();
    GPIOIntClear(GPIO_PORTF_BASE, GPIO_INT_PIN_3);


        if(GPIOPinRead(GPIO_PORTF_BASE, GPIO_INT_PIN_3) ==
GPIO_INT_PIN_3)
        {
            HWREG(TIMER2_BASE + TIMER_O_TAV) = 0;
            TimerEnable(TIMER2_BASE,TIMER_A);
            start = TimerValueGet(TIMER2_BASE,TIMER_A);
        }

        else
        {

            end = TimerValueGet(TIMER2_BASE,TIMER_A);
            TimerDisable(TIMER2_BASE,TIMER_A);
            time_pulse = end - start;
            conv_complete = 1;


        }

    taskEXIT_CRITICAL();
```

```
}

void UtrasonicTask(void *pvParameters)
{
    vTaskDelay(1000/portTICK_PERIOD_MS);
    UARTprintf("Created ultrasonic thread\n");
    long x_ultra_id = 1003;
    xTimerHandle xTimer_ult;
    xTimer_ult = xTimerCreate("Timer_ultrasonic",               // Just a
text name, not used by the kernel.
                              pdMS_TO_TICKS( PERIOD_ULTRASONIC ),
                              pdTRUE,                    // The timers
will auto-reload themselves when they expire.
                               ( void * ) x_ultra_id,      // Assign
each timer a unique id equal to its array index.
                              vTimerCallback_Ultra_handler// Each timer
calls the same callback when it expires.
                             );



    if( (xTimer_ult == NULL ) )
    {
        // The timer was not created.
        UARTprintf("Error on timer creation - xTimer_Temp\n");
    }

    else
    {

        /*start the timer*/
         xTimerStart( xTimer_ult, 0 );

         init_ultrasonic_sensor();

         //startup test
         find_object();
         vTaskDelay(500/portTICK_PERIOD_MS);


         if(time_pulse == 0)
         {
             UARTprintf("Startup test failed for ultrasonic sensor\n");
             LOG_ERROR("Startup failed for ULTRASONIC\n")
             DEGRADED_MODE_MANUAL = 1;
             mode = 1;
             vTaskDelete( NULL );
         }



         for(;;)
         {
```

```
            if(FLAG_UL == pdTRUE)
            {
                find_object();

                if((conv_complete == 1))
                {
                    distance_send =
(((float)(1.0/(output_clock_rate_hz/1000000))*time_pulse)/58);

                    if(CN_ACTIVE)
                    {
                        xQueueSendToBack( myQueue_ultra,( void * )
&distance_send, QUEUE_TIMEOUT_TICKS ) ;
                        memset(buffer_log,'\0',BUFFER);
                        sprintf(buffer_log,"D %f\n",distance_send);
                        LOG_INFO(buffer_log)
                    }
                    else
                    {
                        if(distance_send < 30)
                        {
                            //when object detected
                            xTaskNotifyGive(handle_motor);


                        }

                    }
                    conv_complete = 0;

                }
                else
                {
                    ULT_DEAD++;
                }

                FLAG_UL = pdFALSE;

            }
            if(ULT_DEAD > 5) //switch to degraded mode
            {
               DEGRADED_MODE_MANUAL = 1;
               mode = 1;
               stop();
               UARTprintf("Killed Utrasonic sensor task\n");
               LOG_ERROR("Killed Utrasonic sensor task\n")
               vTaskDelete( NULL );
            }


        }

    }
}
```

```
/*********************************************
 *         Temp timer handler
 *********************************************/
void vTimerCallback_Ultra_handler( TimerHandle_t  *pxTimer )
{
    FLAG_UL = pdTRUE;
}

/*
 * lux.c
 *
 *  Created on: Apr 9, 2019
 *      Author: Steve Antony
 */

/*********************************************
 *        Includes
 *********************************************/
#include <lux.h>

/*********************************************
 *        Globals
 *********************************************/
int FLAG_Light = 0;
struct log_struct_temp log_temp;
static char buffer_log[BUFFER];
char temp_buffer[100];

/*for writing and reading as byte from the registers*/
uint8_t register_data;

/*for storing MSB and LSB of CH0 of lux*/
uint16_t MSB_0;
uint16_t LSB_0;

/*for storing MSB and LSB of CH1 of lux*/
uint16_t MSB_1;
uint16_t LSB_1;

/*16 bit value of CH0 and CH1*/
uint16_t CH0;
uint16_t CH1;
float lux_send;

static uint8_t start_again = 1;


/*********************************************
 *         Temperature thread
 *********************************************/

void LightTask(void *pvParameters)
{
```

```
    vTaskDelay(3000/portTICK_PERIOD_MS);
    UARTprintf("Created Light Task\n");

        long x_light_id = 10005;
        xTimerHandle xTimer_light;
        xTimer_light = xTimerCreate("Timer_Light",              // Just a
text name, not used by the kernel.
                                    pdMS_TO_TICKS( TEMP_TIME_PERIOD_MS ),
// 100ms
                                    pdTRUE,                    // The timers
will auto-reload themselves when they expire.
                                     ( void * ) x_light_id,      // Assign
each timer a unique id equal to its array index.
                                    vTimerCallback_Light_handler// Each
timer calls the same callback when it expires.
                                   );

         if( xTimer_light == NULL )
         {
            // The timer was not created.
            UARTprintf("Error on timer creation\n");
         }
         else
         {
            /*Start led timer*/
            xTimerStart( xTimer_light, 0 );
            i2c_setup();
            vTaskDelay(1000/portTICK_PERIOD_MS);

            //Start up tests for lux sensor
            int8_t ret = lux_sensor_setup();
            if(ret == -1)
            {
                UARTprintf("Startup failed for lux\n");
                LOG_ERROR("Killed lux sensor task - Startup failed\n")
                vTaskDelete( NULL );
            }

            for (;;)
            {

                if(FLAG_Light == pdTRUE)
                {
                    FLAG_Light = pdFALSE;
                    read_lux_CH0();
                    read_lux_CH1();

                    lux_send = lux_measurement(CH0,CH1);
                    if(CN_ACTIVE)
                    {
                        xQueueSendToBack( myQueue_light,( void * )
&lux_send, QUEUE_TIMEOUT_TICKS ) ;
                    memset(buffer_log,'\0',BUFFER);
                    sprintf(buffer_log,"L %f\n",lux_send);
```

```
                        LOG_INFO(buffer_log)
                    }
                    if((CN_ACTIVE == 0) && (lux_send < 1.0)&&(start_again
== 1))
                    {
                        UARTprintf("CN INACTIVE, lux < 1, start\n");
                        forward();
                        start_again = 0;
                    }


                }
            }
        }
}


void vTimerCallback_Light_handler( TimerHandle_t  *pxTimer )
{
    FLAG_Light = pdTRUE;
}


void i2c_setup()
{
    /*Enabling i2c pheripheral*/
    SysCtlPeripheralEnable(SYSCTL_PERIPH_I2C2);

    /*Enabling GPIO*/
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPION);

    /*Configuring I2C SDA GPIO*/
    GPIOPinConfigure(GPIO_PN4_I2C2SDA);

    /*Configuring I2C SCL GPIO*/
    GPIOPinConfigure(GPIO_PN5_I2C2SCL);

    /*Configuring ic2 SCL*/
    GPIOPinTypeI2CSCL(GPIO_PORTN_BASE, GPIO_PIN_5);

    /*Configuring ic2 SDA*/
    GPIOPinTypeI2C(GPIO_PORTN_BASE, GPIO_PIN_4);

    /*wait till specified peripheral is ready
     * returns true if ready*/
    while(!SysCtlPeripheralReady(SYSCTL_PERIPH_I2C2));

    /*initiating i2c master*/
    I2CMasterInitExpClk(I2C2_BASE, output_clock_rate_hz, false);
}

/*Configuring lux sensor*/
int8_t lux_sensor_setup()
{
```

```
    int flag = 0;
    /*command to write on control register*/
    register_data = 0x03;
    write_byte_i2c2(LIGHT_SENSOR, CONTROL_REGISTER, register_data);

    register_data = 0x00;
    read_byte_i2c2(LIGHT_SENSOR, CONTROL_REGISTER, &register_data);

    //UARTprintf("0x03 --> %x",register_data);
    if((register_data == 0x00))
    {
        return -1;
    }

    /*command to write on TIMING_REGISTER*/
    register_data = 0x12;
    write_byte_i2c2(LIGHT_SENSOR, TIMING_REGISTER, register_data);

    return 0;

}


void read_lux_CH0()
{
    /*command to write on control register*/
    register_data = 0x00;
    read_byte_i2c2(LIGHT_SENSOR, DATA0LOW_REGISTER, &register_data);

    LSB_0 = 0;
    LSB_0 = register_data;

    /*command to write on TIMING_REGISTER*/
    register_data = 0x00;
    read_byte_i2c2(LIGHT_SENSOR, DATA0HIGH_REGISTER, &register_data);

    MSB_0 = 0;
    MSB_0 = register_data;

    /*forming the full 16 bit from MSB and LSB*/
    CH0 = (MSB_0 << 8);
    CH0 |= LSB_0;


}

void read_lux_CH1()
{
    /*command to write on control register*/
    register_data = 0x00;
    read_byte_i2c2(LIGHT_SENSOR, DATA1LOW_REGISTER, &register_data);

    LSB_0 = 0;
    LSB_0 = register_data;
```

```
    /*command to write on TIMING_REGISTER*/
    register_data = 0x00;
    read_byte_i2c2(LIGHT_SENSOR, DATA1HIGH_REGISTER, &register_data);

    MSB_1 = 0;
    MSB_1 = register_data;

    /*forming the full 16 bit from MSB and LSB*/
    CH1 = (MSB_1 << 8);
    CH1 |= LSB_1;


}

void read_byte_i2c2(uint8_t slave, uint8_t register_addr, uint8_t * data)
{
    /*select the register to read on slave*/
    I2CMasterSlaveAddrSet(I2C2_BASE, slave, false);

    //command to write to control register
    I2CMasterDataPut(I2C2_BASE, register_addr | WRITE_COMMAND  );

    //Controls the state of the I2C Master , command
    I2CMasterControl(I2C2_BASE, I2C_MASTER_CMD_SINGLE_SEND);

    //Wait until master says it is busy
    while(!I2CMasterBusy(I2C2_BASE));

    //Indicates whether I2C Master is busy
    while(I2CMasterBusy(I2C2_BASE));

    /* reads the data*/
    /*Sets the address that the I2C Master places on the bus*/
     I2CMasterSlaveAddrSet(I2C2_BASE, slave, true);

     I2CMasterControl(I2C2_BASE, I2C_MASTER_CMD_SINGLE_RECEIVE);

     //Wait until master says it is busy
     while(!I2CMasterBusy(I2C2_BASE));

     //Indicates whether I2C Master is busy
     while(I2CMasterBusy(I2C2_BASE));

     *data = I2CMasterDataGet(I2C2_BASE);


}


void write_byte_i2c2(uint8_t slave, uint8_t register_addr, uint8_t data)
{
    /*select the register to read on slave*/
```

```
    I2CMasterSlaveAddrSet(I2C2_BASE, slave, false);

    //command to write to control register
    I2CMasterDataPut(I2C2_BASE, register_addr | WRITE_COMMAND  );

    //Controls the state of the I2C Master , command
    I2CMasterControl(I2C2_BASE, I2C_MASTER_CMD_SINGLE_SEND);

    //Wait until master says it is busy
    while(!I2CMasterBusy(I2C2_BASE));

    //Indicates whether I2C Master is busy
    while(I2CMasterBusy(I2C2_BASE));

    I2CMasterDataPut(I2C2_BASE, data);


     I2CMasterControl(I2C2_BASE, I2C_MASTER_CMD_SINGLE_SEND);

     //Wait until master says it is busy
     while(!I2CMasterBusy(I2C2_BASE));

     //Indicates whether I2C Master is busy
     while(I2CMasterBusy(I2C2_BASE));


}


/*************************************************************
                    Getting lux value
*************************************************************/
float lux_measurement(float CH0, float CH1)
{

    float ratio = (CH1 / CH0);



    //0 < CH1/CH0 â‰¤ 0.50 Sensor Lux = (0.0304 x CH0) â€“ (0.062 x CH0 x
((CH1/CH0)1.4))

    if((ratio <=0.5)&& (ratio > 0))
        return ((0.0304 * CH0) - (0.062 * CH0 * (powf(ratio, 1.4))));

    //0.50 < CH1/CH0 â‰¤ 0.61 Sensor Lux = (0.0224 x CH0) â€“ (0.031 x
CH1)

    else if((ratio  > 0.5)&& (ratio <= 0.61))
        return ((0.0224 * CH0) - (0.031 * CH1));

    //0.61 < CH1/CH0 â‰¤ 0.80 Sensor Lux = (0.0128 x CH0) â€“ (0.0153 x
CH1)
    else if((ratio  > 0.61)&& (ratio <= 0.8))
```

```
        return (0.0128 * CH0) - (0.0153 * CH1);

    //0.80 < CH1/CH0 â‰¤ 1.30 Sensor Lux = (0.00146 x CH0) â€“ (0.00112 x
CH1)
    else if((ratio  > 0.80)&& (ratio <= 1.30))
        return (0.00146 * CH0) - (0.00112 * CH1);

    //CH1/CH0>1.30 Sensor Lux = 0
    else
        return 0;


}
```