CODEDUMP

```
/*reference: https://github.com/akobyl/TM4C129_FreeRTOS_Demo
 *
 * main.c in which we create different tasks for sensors, logger and alert
task.
 * */

#include <stdint.h>
#include <stdbool.h>
#include <string.h>
#include "main.h"
#include "drivers/pinout.h"
#include "driverlib/gpio.h"
#include "utils/uartstdio.h"
#include "inc/hw_memmap.h"
#include "driverlib/rom_map.h"
#include "driverlib/sysctl.h"
#include "utils/uartstdio.h"
#include "driverlib/pin_map.h"
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include "inc/hw_types.h"
#include "driverlib/rom.h"
#include "driverlib/uart.h"
#include "utils/uartstdio.h"
#include "driverlib/adc.h"
#include "driverlib/rom.h"

#define myQueueLength 500

#define ULONG_MAX 0xFFFFFFFF

// TivaWare includes
#include "driverlib/sysctl.h"
#include "driverlib/debug.h"
#include "driverlib/rom.h"
#include "driverlib/rom_map.h"
#include "driverlib/i2c.h"

// FreeRTOS includes
#include "FreeRTOSConfig.h"
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#include "timers.h"
#include "semphr.h"

#include "si7021.h"
#include "uart.h"
#include "gas_flame.h"
#include "startup.h"
#include "unit_test.h"
```

```
//#include "logger.h"

#include "inc/hw_memmap.h"
#include "driverlib/gpio.h"
#include "driverlib/pin_map.h"
#include "driverlib/ssi.h"
#include "driverlib/sysctl.h"
#include "driverlib/uart.h"
#include "utils/uartstdio.h"

#define BAUD_RATE 115200
#define SysClock 120000000

// Demo Task declarations
void Gas_Task(void *pvParameters);
void Flame_Task(void *pvParameters);
void Humidity_Task(void *pvParameters);
void Temp_Task(void *pvParameters);
void Log_Task(void *pvParameters);
void Alert_Task(void *pvParameters);
void UART_read(void *pvParameters);

// Timer task declarations
void Log_Task_timer(TimerHandle_t xTimer5);
void Gas_Task_timer(TimerHandle_t xTimer1);
void Flame_Task_timer(TimerHandle_t xTimer2);
void Temp_Task_timer(void *pvParameters);
void Humidity_Task_timer(void *pvParameters);
void Humidity_Task(TimerHandle_t xTimer3);
void Temp_Task(TimerHandle_t xTimer4);

void Queue_init();


volatile uint32_t ADC0Value[1];
volatile uint32_t ADC1Value[1];
uint16_t samplePeriod;
uint32_t sequence;
volatile uint32_t rh, tp;
double humidity_val, temp_val;

#define SLAVE_ADDRESS 0x40
#define RH_ADDR 0xE5
#define TEMP_ADDR 0xE3


TaskHandle_t Task1Handle;
TaskHandle_t Task2Handle;
TaskHandle_t Task3Handle;
TaskHandle_t Task4Handle;
TaskHandle_t LogTaskHandle;
TaskHandle_t AlertTaskHandle;
TaskHandle_t UARTTaskHandle;
```

```
xQueueHandle queue_handle;

SemaphoreHandle_t my_sem;
SemaphoreHandle_t sem_uart;

QueueHandle_t myQueue;
volatile char buff1[400] = {0};
volatile char buff2[10] = {0};
volatile char *ptr;
volatile char *ptr2;
volatile char abuff[400] = {0};
volatile char *aptr;

typedef struct{
    float data;
    int TaskID;
    int alert;
    //char * string_msg
}message;

typedef enum
{
    Gas_task = 1,
    Flame_task = 2,
    Temperature_task = 3,
    Humidity_task = 4,
}task_id;

typedef enum
{
    DATA,
    ERROR,
}log_level;

typedef enum
{
    Co_alert = 1,
    Flame_alert = 2,
    Humidity_alert = 4,
    Temp_alert = 8,
    Sensor_disconnected = 16,
    }alert;

volatile message msg_struct;

void UART_send(char* ptr, int len)
{

    if(xSemaphoreTake(sem_uart, portMAX_DELAY))
        {
               while(len != 0)
               {
                   UARTCharPut(UART2_BASE, *ptr);
                   ptr++;
```

```
                len--;
            }
        }

        xSemaphoreGive(sem_uart);
    }


void ssi_init()
{
    // reference example code spi_master.c

    int NUM_SSI_DATA =3;

        uint32_t pui32DataTx[3];
        uint32_t pui32DataRx[3];
        uint32_t ui32Index;



        // The SSI0 peripheral must be enabled for use.
        SysCtlPeripheralEnable(SYSCTL_PERIPH_SSI3);


        SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);


        GPIOPinConfigure(GPIO_PA2_SSI0CLK);
        GPIOPinConfigure(GPIO_PA3_SSI0FSS);
        GPIOPinConfigure(GPIO_PA4_SSI0XDAT0);
        GPIOPinConfigure(GPIO_PA5_SSI0XDAT1);

        //
        // Configure the GPIO settings for the SSI pins.  This function also
gives
        // control of these pins to the SSI hardware.  Consult the data sheet
to
        // see which functions are allocated per pin.
        // The pins are assigned as follows:
        //      PA5 - SSI0Tx
        //      PA4 - SSI0Rx
        //      PA3 - SSI0Fss
        //      PA2 - SSI0CLK
        // TODO: change this to select the port/pin you are using.
        //
        GPIOPinTypeSSI(GPIO_PORTA_BASE, GPIO_PIN_5 | GPIO_PIN_4 | GPIO_PIN_3 |
                       GPIO_PIN_2);


        // Configure and enable the SSI port for SPI master mode.
        SSIConfigSetExpClk(SSI3_BASE, g_ui32SysClock, SSI_FRF_MOTO_MODE_0,
                           SSI_MODE_MASTER, 1000000, 8);

        SSIAdvModeSet(SSI3_BASE, SSI_ADV_MODE_READ_WRITE);
```

```
        SSIAdvFrameHoldEnable(SSI3_BASE);

        // Enable the SSI0 module.
        SSIEnable(SSI3_BASE);

        while(SSIDataGetNonBlocking(SSI3_BASE, &pui32DataRx[0]))
        {
        }

        //
        // Initialize the data to send.
        //
        pui32DataTx[0] = 's';
        pui32DataTx[1] = 'p';
        pui32DataTx[2] = 'i';


        for(ui32Index = 0; ui32Index < NUM_SSI_DATA; ui32Index++)
        {
            SSIDataPut(SSI3_BASE, pui32DataTx[ui32Index]);
        }


        // Wait until SSI0 is done transferring all the data in the transmit
FIFO.
        while(SSIBusy(SSI3_BASE))
        {
        }


        // Receive 3 bytes of data.
        for(ui32Index = 0; ui32Index < NUM_SSI_DATA; ui32Index++)
        {
            SSIDataGet(SSI3_BASE, &pui32DataRx[ui32Index]);
        }
}

// Main function
int main(void)
{
        g_ui32SysClock = MAP_SysCtlClockFreqSet((SYSCTL_XTAL_25MHZ |
                    SYSCTL_OSC_MAIN | SYSCTL_USE_PLL |
                    SYSCTL_CFG_VCO_480), SysClock);

        ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOE);    //Enable GPIO
       // ssi_init();

        if (startup() == -1)
            exit(-1);

        PinoutSet(false, false);

        ROM_GPIOPinTypeGPIOOutput(GPIO_PORTN_BASE, GPIO_PIN_1);
```

```
        int testGas, testFlame, testTemp, alertGas, alertFlame, alertTemp,
alertGasDis, alertFlameDis, loop;

        testGas = test_gas();
        testFlame = test_flame();
        testTemp = test_temp();
        alertGas = alert_gas();
        alertTemp = alert_temp();
        alertFlame = alert_flame();
        alertGasDis = alert_gas_disconnected();
        alertFlameDis = alert_flame_disconnected();
        //loop = loopback_test();


        // Create tasks

        xTaskCreate(Gas_Task_timer, (const portCHAR *)"Gas task",
                configMINIMAL_STACK_SIZE, NULL, 1, &Task1Handle);

        xTaskCreate(Flame_Task_timer, (const portCHAR *)"Flame task",
                configMINIMAL_STACK_SIZE, NULL, 1, &Task2Handle);

        /*xTaskCreate(Humidity_Task_timer, (const portCHAR *)"Humidity Task",
                configMINIMAL_STACK_SIZE, NULL, 1, &Task3Handle);*/

        xTaskCreate(Temp_Task_timer, (const portCHAR *)"Temperature Task",
                configMINIMAL_STACK_SIZE, NULL, 1, &Task4Handle);

        xTaskCreate(Log_Task_timer, (const portCHAR *)"Logger task",
                        configMINIMAL_STACK_SIZE, NULL, 1, &LogTaskHandle);

        xTaskCreate(Alert_Task, (const portCHAR *)"Alert task",
                                        configMINIMAL_STACK_SIZE, NULL, 1,
&AlertTaskHandle);

        /*xTaskCreate(UART_read, (const portCHAR *)"UART read task",
                                                configMINIMAL_STACK_SIZE,
NULL, 1, &UARTTaskHandle);*/


        vTaskStartScheduler();

        while(1);

        return 0;
}

/**
 * Gas_Task_timer, Flame_Task_timer, Humidity_Task_timer, Temp_Task_timer and
Log_Task_timer triggers the timer after every 1 second
 * and calls the Gas_Task, Flame_Task, Humidity_Task and Temp_Task
respectively.
 */
```

```
void Gas_Task_timer(void *pvParameters)
{
    TimerHandle_t xTimer1 = NULL;
    xTimer1 = xTimerCreate("MyTimer1", pdMS_TO_TICKS(1000), pdTRUE, (void *)
pvTimerGetTimerID(xTimer1), Gas_Task);
    xTimerStart(xTimer1, portMAX_DELAY);
    while(1);
}


void Flame_Task_timer(void *pvParameters)
{
    TimerHandle_t xTimer2 = NULL;
    xTimer2 = xTimerCreate("MyTimer2", pdMS_TO_TICKS(1000), pdTRUE, (void *)
pvTimerGetTimerID(xTimer2), Flame_Task);
    xTimerStart(xTimer2, portMAX_DELAY);
    while(1);
}

void Humidity_Task_timer(void *pvParameters)
{
    TimerHandle_t xTimer3 = NULL;
    xTimer3 = xTimerCreate("MyTimer3", pdMS_TO_TICKS(1000), pdTRUE, (void *)
pvTimerGetTimerID(xTimer3), Humidity_Task);
    xTimerStart(xTimer3, portMAX_DELAY);
    while(1);
}

void Temp_Task_timer(void *pvParameters)
{
    TimerHandle_t xTimer3 = NULL;
    xTimer3 = xTimerCreate("MyTimer4", pdMS_TO_TICKS(1000), pdTRUE, (void *)
pvTimerGetTimerID(xTimer3), Temp_Task);
    xTimerStart(xTimer3, portMAX_DELAY);
    while(1);
}

void Log_Task_timer(void *pvParameters)
{
    TimerHandle_t xTimer5 = NULL;
    xTimer5 = xTimerCreate("MyTimer5", pdMS_TO_TICKS(1000), pdTRUE, (void *)
pvTimerGetTimerID(xTimer5), Log_Task);
    xTimerStart(xTimer5, portMAX_DELAY);
    while(1);
}

/* Function: Gas_Task
 * Description: Reads CO gas value, stores it in a struct and sends it to the
Log_Task via IPC message queue.
 */
void Gas_Task(TimerHandle_t xTimer1)
{
    if(xSemaphoreTake(my_sem, portMAX_DELAY))
        {
```

```
    ADCProcessorTrigger(ADC0_BASE, 3);
    while(!ADCIntStatus(ADC0_BASE, 3, false))
    {
    }
    ADCIntClear(ADC0_BASE, 3);
    ADCSequenceDataGet(ADC0_BASE, 3, ADC0Value);


        float f = co_val(ADC0Value[0]);

        msg_struct.data = f;
        msg_struct.TaskID = Gas_task;
        msg_struct.alert = 0;

        xQueueSendToBack(myQueue, &msg_struct, portMAX_DELAY);

        if (f>9)
                {

                    xTaskNotify( AlertTaskHandle, Co_alert, eSetBits);
                }
        if (f<3.3)
                {

                     xTaskNotify( AlertTaskHandle, Sensor_disconnected,
eSetBits);
                }

    }

    xSemaphoreGive(my_sem);

}

/* Function: Flame_Task
 * Description: Reads Flame sensor value, stores it in a struct and sends it
to the Log_Task via IPC message queue.
 */
void Flame_Task(TimerHandle_t xTimer2)
{
    if(xSemaphoreTake(my_sem, portMAX_DELAY))
    {
        ADCProcessorTrigger(ADC1_BASE, 3);
        while(!ADCIntStatus(ADC1_BASE, 3, false))
        {
        }
        ADCIntClear(ADC1_BASE, 3);
        ADCSequenceDataGet(ADC1_BASE, 3, ADC1Value);


                msg_struct.data = ADC1Value[0];
                msg_struct.TaskID = Flame_task;
                msg_struct.alert = 0;
```

```
        if (200 <= ADC1Value[0] && ADC1Value[0] <= 500)
        {

                xTaskNotify( AlertTaskHandle, Flame_alert, eSetBits);
        }
        if (ADC1Value[0] < 150)
                        {

                                xTaskNotify( AlertTaskHandle,
Sensor_disconnected, eSetBits);
                        }
        xQueueSendToBack(myQueue, &msg_struct, portMAX_DELAY);
    }

            xSemaphoreGive(my_sem);
    }

/* Function: Humidity_Task
 * Description: Reads Humidity value, stores it in a struct and sends it to
the Log_Task via IPC message queue.
 */
void Humidity_Task(TimerHandle_t xTimer3)
{
    if(xSemaphoreTake(my_sem, portMAX_DELAY))
        {
        rh = i2cRead(RH_ADDR);
        humidity_val = humidity(rh);

        msg_struct.data = humidity_val;
        msg_struct.TaskID = Humidity_task;

        msg_struct.alert = 1;
        //strcpy(msg_struct.a, "humidity");

        xQueueSendToBack(myQueue, &msg_struct, portMAX_DELAY);

        if (humidity_val < 20)

                {
                        //xTaskNotify( AlertTaskHandle, Flame_alert,
eSetBits);
                }

        }

        xSemaphoreGive(my_sem);

}


/* Function: Temp_Task
 * Description: Reads temperature value, stores it in a struct and sends it to
the Log_Task via IPC message queue.
```

```
 */
void Temp_Task(TimerHandle_t xTimer4)
{
    if(xSemaphoreTake(my_sem, portMAX_DELAY))
    {
        tp = i2cRead(TEMP_ADDR);
        temp_val = temp(tp);

        msg_struct.data = temp_val;
        msg_struct.TaskID = Temperature_task;
        msg_struct.alert = 0;

        xQueueSendToBack(myQueue, &msg_struct, portMAX_DELAY);
        if (temp_val > 35)
                {

                        xTaskNotify( AlertTaskHandle, Flame_alert, eSetBits);
                }

     }

    xSemaphoreGive(my_sem);

    //UARTprintf("Tp = %d\n",temp_val);
}


/* Function: Alert_Task
 * Description: Receives notification from Gas_Task, Flame_Task, Humidity_Task
and Temp_Task and sends the Alert message to BBG through UART
 */
void Alert_Task(void *pvParameters)
{BaseType_t ret;
    int NotifValue = 0;
            while(1){
                ret = xTaskNotifyWait( 0, 0xFF, &NotifValue, portMAX_DELAY);
            // Notify wait
            if(ret == pdTRUE)
            {
                if (NotifValue & Co_alert)
                {
                    if(xSemaphoreTake(my_sem, 250))
                     {
                        msg_struct.TaskID = Gas_task;
                        msg_struct.alert = 1;
                        xQueueSendToBack(myQueue, &msg_struct, portMAX_DELAY);
                     }
                    xSemaphoreGive(my_sem);

                }
                if (NotifValue & Flame_alert)
                {
                    if(xSemaphoreTake(my_sem, 250))
                    {
```

```
                        msg_struct.TaskID = Flame_task;
                        msg_struct.alert = 1;
                        xQueueSendToBack(myQueue, &msg_struct, portMAX_DELAY);

                     }
                     xSemaphoreGive(my_sem);
                }
                if (NotifValue & Humidity_alert)
                {
                    if(xSemaphoreTake(my_sem, 250))
                    {
                        msg_struct.TaskID = Humidity_task;
                        msg_struct.alert = 1;
                        xQueueSendToBack(myQueue, &msg_struct, portMAX_DELAY);
                    }
                    xSemaphoreGive(my_sem);
                }
                if (NotifValue & Temp_alert)
                {
                    if(xSemaphoreTake(my_sem, 250))
                    {
                        msg_struct.TaskID = Temperature_task;
                        msg_struct.alert = 1;
                        xQueueSendToBack(myQueue, &msg_struct, portMAX_DELAY);
                     }
                     xSemaphoreGive(my_sem);
                }
                if (NotifValue & Sensor_disconnected)
                                {
                                    if(xSemaphoreTake(my_sem, 250))
                                    {

                                        msg_struct.alert = 2;
                                        xQueueSendToBack(myQueue, &msg_struct,
portMAX_DELAY);
                                     }
                                     xSemaphoreGive(my_sem);
                                }
            }
            }
}

// Queue initialization

void Queue_init()
{
    myQueue = xQueueCreate(myQueueLength, sizeof(message));
    if(myQueue == NULL)
    {
        perror("Queue not created");
    }

    my_sem = xSemaphoreCreateMutex();
    sem_uart = xSemaphoreCreateMutex();
```

```
}

/*void UART_read(void *pvParameters)
{
    while(1)
    {
        while(UARTCharsAvail(UART2_BASE))
        {
               perror("Done");
        }
    }
}*/

/* Function: Log_task
 * Description: Receives data struct via IPC message queue from Gas_task,
Flame_Task, Humidity_Task and Temp_Task and sends it to BBG via UART
 */
void Log_Task(void *pvParameters)
{
    sprintf(buff2, "%s", "\n");
    ptr2 = &buff2;

    while(uxQueueSpacesAvailable(myQueue) != myQueueLength)
    {
            if(xSemaphoreTake(my_sem, 250))
            {
                 xQueueReceive(myQueue, &msg_struct, portMAX_DELAY);
            }

        //sprintf(buff1, "Data:%f, Length:%i, TaskId:%i, LogLevel:%i\n\0",
msg_struct.data, msg_struct.data_len, msg_struct.TaskID, msg_struct.LogLevel);
        //ptr = &buff1;
        ptr = (uint8_t *)&msg_struct;

        UART_send(ptr, sizeof(message));
        //UART_send(ptr, sizeof(msg_struct));

        /*if(msg_struct.TaskID == Temperature_task)
            UART_send(ptr2, strlen(buff2));*/
        xSemaphoreGive(my_sem);
    }
}

/*  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)
    {
    }
```

```
}


/*
 * main.h
 *
 *  Created on: Mar 28, 2015
 *
 */

#ifndef MAIN_H_
#define MAIN_H_

// System clock rate, 120 MHz
#define SYSTEM_CLOCK    120000000U



// struct to store all the sensor values


#endif /* MAIN_H_ */



/* Si7021 Task file */

#include "si7021.h"

 i2c_init()
{
                //enable GPIO peripheral that contains I2C 0
                SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB);

                // Configure the pin muxing for I2C0 functions on port B2 and
B3.
                GPIOPinConfigure(GPIO_PB2_I2C0SCL);
                GPIOPinConfigure(GPIO_PB3_I2C0SDA);

                // Select the I2C function for these pins.
                GPIOPinTypeI2CSCL(GPIO_PORTB_BASE, GPIO_PIN_2);
                GPIOPinTypeI2C(GPIO_PORTB_BASE, GPIO_PIN_3);

                SysCtlPeripheralDisable(SYSCTL_PERIPH_I2C0);
                SysCtlPeripheralReset(SYSCTL_PERIPH_I2C0);
                SysCtlPeripheralEnable(SYSCTL_PERIPH_I2C0);
                while(!SysCtlPeripheralReady(SYSCTL_PERIPH_I2C0));

                I2CMasterInitExpClk(I2C0_BASE, g_ui32SysClock, false);

    }

uint32_t i2cRead(int RegAddr)
```

```
{
    uint16_t return_val[2];
    int count = 0;
           I2CMasterSlaveAddrSet(I2C0_BASE, SLAVE_ADDRESS, false);

           //specify register to be read
           I2CMasterDataPut(I2C0_BASE, RegAddr);

           //send control byte and register address byte to slave device
           I2CMasterControl(I2C0_BASE, I2C_MASTER_CMD_SINGLE_SEND);

           //wait for MCU to finish transaction
           while(I2CMasterBusy(I2C0_BASE))
           {
           }
           //while(!(I2CSlaveStatus(I2C0_BASE) & I2C_SLAVE_ACT_TREQ));
           //specify that we are going to read from slave device
           I2CMasterSlaveAddrSet(I2C0_BASE, SLAVE_ADDRESS, true);

           //send control byte and read from the register we
           //specified
           I2CMasterControl(I2C0_BASE, I2C_MASTER_CMD_SINGLE_RECEIVE);

           //wait for MCU to finish transaction
           while(I2CMasterBusy(I2C0_BASE));

           //return data pulled from the specified register
           return_val[0] =  I2CMasterDataGet(I2C0_BASE);
           I2CMasterControl(I2C0_BASE, I2C_MASTER_CMD_SINGLE_RECEIVE);

           //wait for MCU to finish transaction
           while(I2CMasterBusy(I2C0_BASE));
           return_val[1] =  I2CMasterDataGet(I2C0_BASE);

           uint32_t r = (return_val[0] *256) + return_val[1];
           return r;
    }

double humidity(uint32_t rh)
{
    double final;
    final = rh*125;
    final = final/65536;
    final = final - 6;
    return final;
    }

double temp(uint32_t rh)
{
    double final;
    final = rh*175.72;
    final = final/65536;
    final = final - 46.85;
    return final;
```

```
    }


#include <stdint.h>
#include <stdbool.h>
#include <string.h>
#include "main.h"
#include "drivers/pinout.h"
#include "driverlib/gpio.h"
#include "utils/uartstdio.h"
#include "inc/hw_memmap.h"
#include "driverlib/rom_map.h"
#include "driverlib/sysctl.h"
#include "utils/uartstdio.h"
#include "driverlib/pin_map.h"
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include "inc/hw_types.h"
#include "driverlib/rom.h"
#include "driverlib/uart.h"
#include "utils/uartstdio.h"
#include "driverlib/adc.h"
#include "driverlib/rom.h"

// TivaWare includes
#include "driverlib/sysctl.h"
#include "driverlib/debug.h"
#include "driverlib/rom.h"
#include "driverlib/rom_map.h"
#include "driverlib/i2c.h"

// FreeRTOS includes
#include "FreeRTOSConfig.h"
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#include "timers.h"


volatile uint32_t rh, tp;
double humidity_val, temp_val;
uint32_t g_ui32SysClock;
static uint16_t r;

TaskHandle_t AlertTaskHandle;



#define SLAVE_ADDRESS 0x40
#define RH_ADDR 0xE5
#define TEMP_ADDR 0xE3

/* @brief  I2C initialization function
```

```
 * This function will initialize i2c
 * @param  void
 * @return 0: on success
 *        -1: on error
 * */
int i2c_init();

/* @brief  I2C read
 * This function will read values from Si7021 sensor
 * @param  RegAddr: address of the register from whcih values will be read
 * @return uint32_t: value read
 * */
uint32_t i2cRead(int RegAddr);

/* @brief  Conversion function(%RH)
 * This function will convert the value read from the registers to humidity in
percentage
 * @param  rh: value read from registers
 * @return double: converted value
 * */
double humidity(uint32_t rh);

/* @brief  Conversion function(Temperature)
 * This function will convert the value read from the registers to Temperature
in Celsius
 * @param  rh: value read from registers
 * @return double: converted value
 * */
double temp(uint32_t rh);


#include "main.h"
#include "gas_flame.h"
#include "si7021.h"
#include "uart.h"
#include "math.h"

int adc_ch0_init()
{
    SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC0);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC0);

    while(!SysCtlPeripheralReady(SYSCTL_PERIPH_ADC0));

    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);
    MAP_ADCReferenceSet(ADC0_BASE, ADC_REF_EXT_3V);

    //ROM_ADCHardwareOversampleConfigure(ADC0_BASE,0);
    ADCSequenceEnable(ADC0_BASE, 3);
```

```
    return 0;
    }

int adc_ch1_init()
{
    SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC1);
    //SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC0);

    while(!SysCtlPeripheralReady(SYSCTL_PERIPH_ADC1));

    GPIOPinTypeADC(GPIO_PORTE_BASE,GPIO_PIN_2);
    ADCSequenceConfigure(ADC1_BASE, 3, ADC_TRIGGER_PROCESSOR, 0);

    ADCSequenceStepConfigure(ADC1_BASE,3,0,ADC_CTL_CH1
|ADC_CTL_IE|ADC_CTL_END);
    MAP_ADCReferenceSet(ADC1_BASE, ADC_REF_EXT_3V);

    //ROM_ADCHardwareOversampleConfigure(ADC0_BASE,0);
    ADCSequenceEnable(ADC1_BASE, 3);

    return 0;
    }

float co_val(uint32_t val)
{
    double e = 2.718;
       double f;
       f = (val*3.3)/4095;
       f = (1.0698*f);
       f = pow(e, f);
       f = 3.027 * f;
       f = (float)f;
    return f;
    }


#include <stdint.h>
#include <stdbool.h>
#include <string.h>
#include "main.h"
#include "drivers/pinout.h"
#include "driverlib/gpio.h"
#include "utils/uartstdio.h"
#include "inc/hw_memmap.h"
#include "driverlib/rom_map.h"
#include "driverlib/sysctl.h"
#include "utils/uartstdio.h"
#include "driverlib/pin_map.h"
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include "inc/hw_types.h"
#include "driverlib/rom.h"
#include "driverlib/uart.h"
```

```c
#include "utils/uartstdio.h"
#include "driverlib/adc.h"
#include "driverlib/rom.h"

// TivaWare includes
#include "driverlib/sysctl.h"
#include "driverlib/debug.h"
#include "driverlib/rom.h"
#include "driverlib/rom_map.h"
#include "driverlib/i2c.h"

// FreeRTOS includes
#include "FreeRTOSConfig.h"
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#include "timers.h"

/* @brief  ADC initialization
 * This function will initialize ADC ch0
 * @param  void
 * @return 0: on success
 *        -1: on error
 * */
int adc_ch0_init();

/* @brief  ADC initialization
 * This function will initialize ADC ch1
 * @param  void
 * @return 0: on success
 *        -1: on error
 * */
int adc_ch1_init();

/* @brief  Conversion (CO in ppm)
 * This function will convert the value from ADC to ppm
 * @param  val: value obtained from ADC
 * @return float: value of CO gas in ppm
 * */
float co_val(uint32_t val);


#include "si7021.h"
#include "uart.h"
#include "gas_flame.h"

int startup()
{
    //call init functions

            if(ConfigureUART2() == -1)
            {
                return -1;
            }
```

```
            Queue_init();

            if(adc_ch0_init() == -1) // ADC for gas sensor
            {
                return -1;
            }

            if(adc_ch1_init() == -1) // ADC for flame sensor
            {
                return -1;
            }

            if(i2c_init() == -1)
            {
                return -1;
            }

            return 0;
    }


#include "main.h"
#include "uart.h"
#include "si7021.h"
#include "gas_flame.h"
#include "driverlib/interrupt.h"
#include "inc/hw_ints.h"

int ConfigureUART0(void)
{
        ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);    //Enable GPIO

        ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);    //Enable UART0

        ROM_GPIOPinConfigure(GPIO_PA0_U0RX);                //Configure UART
pins
        ROM_GPIOPinConfigure(GPIO_PA1_U0TX);
        ROM_GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1);

        //UARTStdioConfig(2, BAUD_RATE, g_ui32SysClock);         //Initialize
UART
        ROM_UARTConfigSetExpClk(UART0_BASE, g_ui32SysClock, 115200,
                                (UART_CONFIG_WLEN_8 | UART_CONFIG_STOP_ONE
|
                                                    UART_CONFIG_PAR_NONE));
        return 0;

}

int 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, g_ui32SysClock, 115200,
                                (UART_CONFIG_WLEN_8 | UART_CONFIG_STOP_ONE |
                                            UART_CONFIG_PAR_NONE));

    //UARTFIFOEnable(UART2_BASE)0

    IntMasterEnable();

//    ROM_IntEnable(UART2_BASE);

    IntEnable(INT_UART2);
    ROM_UARTIntEnable(UART2_BASE, UART_INT_RX | UART_INT_RT);
    return 0;
}

void UARTIntHandler()
{
    char c;
    uint32_t status = ROM_UARTIntStatus(UART2_BASE, true);
    ROM_UARTIntClear(UART2_BASE, status);
    while(UARTCharsAvail(UART2_BASE))
    {
        c = ROM_UARTCharGet(UART2_BASE);
        //UARTprintf("%c", c);
    }

}


#include <stdint.h>
#include <stdbool.h>
#include <string.h>
#include "main.h"
#include "drivers/pinout.h"
#include "driverlib/gpio.h"
#include "utils/uartstdio.h"
#include "inc/hw_memmap.h"
#include "driverlib/rom_map.h"
#include "driverlib/sysctl.h"
#include "utils/uartstdio.h"
#include "driverlib/pin_map.h"
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include "inc/hw_types.h"
#include "driverlib/rom.h"
#include "driverlib/uart.h"
#include "utils/uartstdio.h"
```

```c
#include "driverlib/adc.h"
#include "driverlib/rom.h"

// TivaWare includes
#include "driverlib/sysctl.h"
#include "driverlib/debug.h"
#include "driverlib/rom.h"
#include "driverlib/rom_map.h"
#include "driverlib/i2c.h"

// FreeRTOS includes
#include "FreeRTOSConfig.h"
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#include "timers.h"



/* @brief  UART send
 * This function will send data to the BBG via UART
 * @param  ptr: pointer which points to the struct to be send
 *         struct_len: length of the struct
 * @return void
 * */
//void UART_send(char* ptr, int struct_len);


/* @brief  COnfigure UART0
 * This function will configure UART0
 * @param  void
 * @return 0: on success
 *        -1: on error
 * */
int ConfigureUART0(void);


/* @brief  COnfigure UART2
 * This function will configure UART2
 * @param  void
 * @return 0: on success
 *        -1: on error
 * */
int ConfigureUART2(void);

void UARTIntHandler();


/* File: unit_test.c
 * Description: We test various APIs and functions in this file
 * */
#include "unit_test.h"
```

```
SemaphoreHandle_t my_sem;
SemaphoreHandle_t sem_uart;


int test_gas()
{
    uint32_t ADC0Value[1];
    float f;

        ADCProcessorTrigger(ADC0_BASE, 3);
        while(!ADCIntStatus(ADC0_BASE, 3, false))
        {
        }
        ADCIntClear(ADC0_BASE, 3);
        ADCSequenceDataGet(ADC0_BASE, 3, ADC0Value);

        f = co_val(ADC0Value[0]);

        if (f >= 3.5)
            return SUCCESS;
        else return FAIL;

    }

int test_flame()
{
    uint32_t ADC1Value[1];
            ADCProcessorTrigger(ADC1_BASE, 3);
            while(!ADCIntStatus(ADC1_BASE, 3, false))
            {
            }
            ADCIntClear(ADC1_BASE, 3);
            ADCSequenceDataGet(ADC1_BASE, 3, ADC1Value);

        if (200 <= ADC1Value[0] && ADC1Value[0] <= 5000)
        {
            return SUCCESS;
        }
        return FAIL;
    }

int test_temp()
{
    tp = i2cRead(TEMP_ADDR);
    temp_val = temp(tp);

     if (temp_val > 20)
         return SUCCESS;
     else return FAIL;
    }

int alert_gas()
{
    uint32_t ADC0Value[1];
```

```
        float f;

            ADCProcessorTrigger(ADC0_BASE, 3);
            while(!ADCIntStatus(ADC0_BASE, 3, false))
            {
            }
            ADCIntClear(ADC0_BASE, 3);
            ADCSequenceDataGet(ADC0_BASE, 3, ADC0Value);

            f = co_val(ADC0Value[0]);

            if (f>9)
                return SUCCESS;
            else return FAIL;
    }

int alert_gas_disconnected()
{
    uint32_t ADC0Value[1];
        float f;

            ADCProcessorTrigger(ADC0_BASE, 3);
            while(!ADCIntStatus(ADC0_BASE, 3, false))
            {
            }
            ADCIntClear(ADC0_BASE, 3);
            ADCSequenceDataGet(ADC0_BASE, 3, ADC0Value);

            f = co_val(ADC0Value[0]);

            if (f<3.5)
                return SUCCESS;
            else return FAIL;
    }

int alert_flame()
{
    uint32_t ADC1Value[1];
               ADCProcessorTrigger(ADC1_BASE, 3);
               while(!ADCIntStatus(ADC1_BASE, 3, false))
               {
               }
               ADCIntClear(ADC1_BASE, 3);
               ADCSequenceDataGet(ADC1_BASE, 3, ADC1Value);

           if (200 <= ADC1Value[0] && ADC1Value[0] <= 500)
           {
               return SUCCESS;
           }
           else return FAIL;
    }

int alert_flame_disconnected()
{
```

```
    uint32_t ADC1Value[1];
    ADCProcessorTrigger(ADC1_BASE, 3);
    while(!ADCIntStatus(ADC1_BASE, 3, false))
    {
    }
    ADCIntClear(ADC1_BASE, 3);
    ADCSequenceDataGet(ADC1_BASE, 3, ADC1Value);

    if (ADC1Value[0] < 50)
    {
        return SUCCESS;
     }
    else return FAIL;
    }

int alert_temp()
{
    tp = i2cRead(TEMP_ADDR);
        temp_val = temp(tp);

         if (temp_val > 35)
             return SUCCESS;
         else return FAIL;

    }



#include <stdint.h>
#include <stdbool.h>
#include <string.h>
#include "main.h"
#include "si7021.h"
#include "gas_flame.h"
#include "uart.h"
#include "semphr.h"

typedef enum{
    FAIL,
    SUCCESS,

}result;


/* @brief  Test gas sensor
 * This function will test Gas sensor values
 * @param  void
 * @return SUCCESS
 *         FAIL
 * */
int test_gas();

/* @brief  Test flame sensor
 * This function will test flame sensor values
```

```
 * @param  void
 * @return SUCCESS
 *         FAIL
 * */
int test_flame();

/* @brief  Test Si7021 sensor
 * This function will test Si7021 sensor values
 * @param  void
 * @return SUCCESS
 *         FAIL
 * */
int test_temp();

/* @brief  ALert gas sensor
 * This function will test the alert notification for gas sensor
 * @param  void
 * @return SUCCESS
 *         FAIL
 * */
int alert_gas();

/* @brief  ALert Si7021 sensor
 * This function will test the alert notification for Si7021 sensor
 * @param  void
 * @return SUCCESS
 *         FAIL
 * */
int alert_temp();

/* @brief  ALert Flame sensor
 * This function will test the alert notification for flame sensor
 * @param  void
 * @return SUCCESS
 *         FAIL
 * */
int alert_flame();

/* @brief  Gas sensor disconnected
 * This function will test if the notification is send when the gas sensor is
disconnected
 * @param  void
 * @return SUCCESS
 *         FAIL
 * */
int alert_gas_disconnected();

/* @brief  Flame sensor disconnected
 * This function will test if the notification is send when the flame sensor
is disconnected
 * @param  void
 * @return SUCCESS
 *         FAIL
 * */
```

```
int alert_flame_disconnected();


//client.c

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <netdb.h>

typedef enum
{
    Gas = 1,
    Flame = 2,
     Humidity = 3,
     Temp = 4
}command_enum;

void main(int argc, char *argv[])
{
      //while(1)
      {
      int sock;
      struct sockaddr_in server;
      char buff[1024];
      struct hostent *hp;
      int command;

      printf("List of commands available:\n");
      printf("1. Get CO gas value\n");
      printf("2. Get Flame sensor value\n");
      printf("3. Get Temperature\n");
      printf("Enter approprate command number\n");

      scanf("%d", &command);
      sock = socket(AF_INET, SOCK_STREAM, 0);
      if(sock < 0)
      {
            perror("socket failed");
            exit(1);
      }
      server.sin_family = AF_INET;
      hp = gethostbyname(argv[1]);
      if(hp == 0)
      {
            perror("gethost failed");
            exit(1);
      }
```

```
      memcpy(&server.sin_addr, hp->h_addr, hp->h_length);
      server.sin_port = htons(6011);

      if(connect(sock, (struct sockaddr *)&server, sizeof(server))
< 0)
      {
            perror("connection failes");
            exit(1);
      }

      //int command = Light;


      if(send(sock, (void*)&command, sizeof(command), 0) < 0)
      {
            perror("Send failed");
            exit(1);
      }
      printf("Sent : %d \n", command);
      //sleep(1);

      //int mysock;
      //mysock = accept(sock, (struct sockaddr *)0, 0);

      float incoming;
      int data_in = read(sock, &incoming, sizeof(incoming));
        if (data_in < 0)
      {
            perror("Read failed");
            exit(1);
      }
      printf("Data got back: %f \n", incoming);

      if(command == 1)
      {
            if(incoming < 9 && incoming >3.5)
            {
                  printf("Gas API functionality successful in
normal condition\n");
            }
      }

      if(command == 2)
      {
            if(incoming < 4000 && incoming > 200)
            {
                  printf("Fire API functionalty successful in
normal condition\n");
            }
      }
      if(command == 3)
      {
```

```
            if(incoming < 35 && incoming > 15)
            {
                  printf("Temperature API functionalty successful
in normal condition\n");
            }
      }
      //close(sock);
      //exit(1);
      sleep(1);
      }
}

//socket_task.c

/*
@file - socket_task.c
@brief - Includes all the functions for socket server
@author - Nikhil Divekar & Vipraja Patil
*/

#include "socket_task.h"
#include <mqueue.h>
#include <fcntl.h>
#include <sys/stat.h>
#include "uart.h"

#define QUEUE_PERMISSIONS 0666
#define MAX_MESSAGES 10
#define MAX_MSG_SIZE 256

int fd;

typedef struct
{
      float data;
      int TaskID;
      int alert;
}message;

void socket_server()
{
      int sock;
      struct sockaddr_in server, client;
      int mysock;
      char buff[1024];
      int rval;

      //create socket
      sock = socket(AF_INET, SOCK_STREAM, 0);
      if(sock < 0)
      {
            perror("Failed to create a socket");
```

```
            exit(1);
      }

      server.sin_family = AF_INET;
      server.sin_addr.s_addr = INADDR_ANY;
      server.sin_port = htons(6011);

      //bind
      if(bind(sock, (struct sockaddr *)&server, sizeof(server))
< 0)
      {
            perror("Didn't bind");
            exit(1);
      }

      //Listen
      if(listen(sock, 5) < 0)
      {
            perror("Listening error");
            exit(1);
      }

      //Accept
      while(1)
      {
      mysock = accept(sock, (struct sockaddr *)0, 0);
      if(mysock == -1)
      {
            perror("Accept failed");
            exit(1);
      }
      int incoming;
      int data_in = read(mysock,&incoming,sizeof(incoming));

      if (data_in < 0)
      {
            perror("Error reading");
            exit(1);
      }
        printf("Message: %d \n", incoming);

      int i;

      if(incoming == 1)
      for(i = 0; i< 3; i++)
      {
            message my_data;
            read(fd, &my_data, sizeof(message));
            if(my_data.TaskID != 1)
                  continue;
            printf("External request serviced");
            //fprintf(fptr, "External request services");
```

```
            //if(my_data.TaskID == 1)
            {
            printf("Task Id: %d \n", my_data.TaskID);
            //fprintf(fptr, "Task Id: %d \n", my_data.TaskID);

            printf("Data: %f \n", my_data.data);
            //fprintf(fptr, "Data: %f \n", my_data.data);

            printf("Alert: %d \n", my_data.alert);
            //fprintf(fptr, "Alert: %d \n", my_data.alert);

            printf("\n");
            }

            float value = my_data.data;
            int data_out = send(mysock, (void*)&value,
sizeof(value), 0);
            if (data_out < 0)
            {
                  perror("Error writing");
                  exit(1);
            }
            printf("Data sent back\n");

            break;
      }

      if(incoming == 2)
      for(i = 0; i< 2; i++)
      {
            message my_data;
            read(fd, &my_data, sizeof(message));
            if(my_data.TaskID != 2)
                  continue;
            printf("External request serviced");
            //fprintf(fptr, "External request services");

            printf("Task Id: %d \n", my_data.TaskID);
            //fprintf(fptr, "Task Id: %d \n", my_data.TaskID);

            printf("Data: %f \n", my_data.data);
            //fprintf(fptr, "Data: %f \n", my_data.data);

            printf("Alert: %d \n", my_data.alert);
            //fprintf(fptr, "Alert: %d \n", my_data.alert);

            printf("\n");

            float value = my_data.data;
            int data_out = send(mysock, (void*)&value,
sizeof(value), 0);
            if (data_out < 0)
```

```
            {
                  perror("Error writing");
                  exit(1);
            }
            printf("Data sent back\n");

            break;
      }

      if(incoming == 3)
      for(i = 0; i< 2; i++)
      {
            message my_data;
            read(fd, &my_data, sizeof(message));
            if(my_data.TaskID != 3)
                  continue;
            printf("External request serviced");
            //fprintf(fptr, "External request services");

            printf("Task Id: %d \n", my_data.TaskID);
            //fprintf(fptr, "Task Id: %d \n", my_data.TaskID);

            printf("Data: %f \n", my_data.data);
            //fprintf(fptr, "Data: %f \n", my_data.data);

            printf("Alert: %d \n", my_data.alert);
            //fprintf(fptr, "Alert: %d \n", my_data.alert);

            printf("\n");

            float value = my_data.data;
            int data_out = send(mysock, (void*)&value,
sizeof(value), 0);
            if (data_out < 0)
            {
                  perror("Error writing");
                  exit(1);
            }
            printf("Data sent back");

            break;
      }

      }
      exit(1);
}

//socket_task.h
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
```

```
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <netdb.h>

void socket_server();


//uart.h

#ifndef UART_H
#define UART_H

#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <termios.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <errno.h>
#include <time.h>


void termios_setup(struct termios * my_term, int descriptor);

void read_byte(int fd,char *received);

void uart_setup(void);

#endif

//uart.c

//Reference:
https://en.wikibooks.org/wiki/Serial_Programming/termios

#include "uart.h"
#include "string.h"
#include <stdio.h>
#include <stdint.h>
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <string.h>
#include <unistd.h>
#include <netdb.h>
#include <fcntl.h>
```

```
#include <sys/stat.h>
#include <mqueue.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <netdb.h>
#include <stdbool.h>
#include "userled.h"
#include "socket_task.h"

struct termios *my_term;

typedef enum
{
    Alive = 0,
    Dead = 1,
}command_enum;

pthread_t read_thread;
int read_thread_check;

pthread_t write_thread;
int write_thread_check;

pthread_t api_thread;
int api_thread_check;

pthread_t heartbeat_thread;
int heartbeat_check;

FILE * fptr;
pthread_mutex_t pmutex;

char * uart_driver = "/dev/ttyO4";
char log_name[50];
extern int fd;

typedef struct
{
     float data;
     int TaskID;
     int alert;
}message;

void uart_setup();

void termios_setup(struct termios * my_term, int descriptor);
```

```
void read_byte(int fd, char * received);

void * read_thread_func()
{
      char buffer[1024];
      fptr = fopen(log_name, "w");
      char recv = 'a';
      message my_data;
      int retval;
      int command;
      int sock;
      while(1)
      {
            fptr = fopen(log_name, "a");
            pthread_mutex_lock(&pmutex);

            retval = read(fd, &my_data, sizeof(message));

            int sock;
            struct sockaddr_in server;
            char buff[1024];
            struct hostent *hp;

            sock = socket(AF_INET, SOCK_STREAM, 0);
            if(sock < 0)
            {
                  perror("socket failed");
                  exit(1);
            }
            server.sin_family = AF_INET;
            hp = gethostbyname("localhost");
      if(hp == 0)
      {
            perror("gethost failed");
            exit(1);
      }
      memcpy(&server.sin_addr, hp->h_addr, hp->h_length);
      server.sin_port = htons(6006);

      if(connect(sock, (struct sockaddr *)&server, sizeof(server))
< 0)
      {
            perror("connection failes");
            exit(1);
      }

      int command = Alive;


      if(send(sock, (void*)&command, sizeof(command), 0) < 0)
      {
            perror("Send failed");
```

```
            exit(1);
      }
      close(sock);

            if(retval > 0 && my_data.TaskID != 0)
            {
                  if(my_data.alert == 2)
                  {
                        printf("Connection Lost to sensor.
Terminating Task");
                        fprintf(fptr, "Connection Lost to sensor.
Terminating Task");
                        exit(-1);
                  }
                  printf("Task Id: %d \n", my_data.TaskID);
                  fprintf(fptr, "Task Id: %d \n", my_data.TaskID);

                  printf("Data: %f \n", my_data.data);
                  fprintf(fptr, "Data: %f \n", my_data.data);

                  printf("Alert: %d \n", my_data.alert);
                  fprintf(fptr, "Alert: %d \n", my_data.alert);

                  char msg_data_string[100];
                  if(my_data.alert == 1 && my_data.TaskID == 1)
                  {
                  strcpy(msg_data_string, "ALERT RECEIEVED FROM GAS
SENSOR \n");
                  printf("%s", msg_data_string);
                  fprintf(fptr, "%s", msg_data_string);
                  userLED(3,1);
                  }

                  if(my_data.alert == 1 && my_data.TaskID == 2)
                  {
                  strcpy(msg_data_string, "ALERT RECEIEVED FROM
FLAME SENSOR \n");
                  printf("%s", msg_data_string);
                  fprintf(fptr, "%s", msg_data_string);
                  userLED(2,1);
                  }

                  if(my_data.alert == 1 && my_data.TaskID == 3)
                  {
                  strcpy(msg_data_string, "ALERT RECEIEVED FROM
TEMPERATURE SENSOR \n");
                  printf("%s", msg_data_string);
                  fprintf(fptr, "%s", msg_data_string);
                  userLED(3,1);
                  }

                  fprintf(fptr, "%s", "\n");
```

```
                printf("\n");
            }

            pthread_mutex_unlock(&pmutex);
            fclose(fptr);
        }
        command = Dead;
        if(send(sock, (void*)&command, sizeof(command), 0) < 0)
        {
            userLED(3,1);
            perror("Send failed");
            printf("Some thread dead, Terminating process\n");
        }
}

void * write_thread_func()
{
        int i;
        char data[3] = "Nik";
           i = write(fd, &data, sizeof(data));
        printf("%d", i);
}

void * api_thread_func()
{
        int command;
        int sock;
            struct sockaddr_in server;
            char buff[1024];
            struct hostent *hp;

            sock = socket(AF_INET, SOCK_STREAM, 0);
            if(sock < 0)
            {
                perror("socket failed");
                exit(1);
            }
            server.sin_family = AF_INET;
            hp = gethostbyname("localhost");
        if(hp == 0)
        {
            perror("gethost failed");
            exit(1);
        }
        memcpy(&server.sin_addr, hp->h_addr, hp->h_length);
        server.sin_port = htons(6006);

        if(connect(sock, (struct sockaddr *)&server, sizeof(server))
< 0)
        {
            perror("connection failes");
            exit(1);
```

```
	}

	command = Alive;


	if(send(sock, (void*)&command, sizeof(command), 0) < 0)
	{
		perror("Send failed");
		exit(1);
	}
	close(sock);
	socket_server();

	command = Dead;
	if(send(sock, (void*)&command, sizeof(command), 0) < 0)
	{
		userLED(3,1);
		perror("Send failed");
		printf("Some thread dead, Terminating process\n");
	}
}

void * hb_thread_func()
{
	int sock;
	struct sockaddr_in check_server;
	int mysock;
	char buff[1024];
	int rval;
	int flag = 0;
	//create socket
	sock = socket(AF_INET, SOCK_STREAM, 0);
	if(sock < 0)
	{
		perror("Failed to create a socket");
		exit(1);
	}

	check_server.sin_family = AF_INET;
	check_server.sin_addr.s_addr = INADDR_ANY;
	check_server.sin_port = htons(6006);

	//bind
	if(bind(sock, (struct sockaddr *)&check_server,
sizeof(check_server)) < 0)
	{
		perror("Didn't bind");
		exit(1);
	}

	//Listen
	if(listen(sock, 5) < 0)
```

```
      {
            perror("Listening error");
            exit(1);
      }

      //Accept
      while(1)
      {
      mysock = accept(sock, (struct sockaddr *)0, 0);
      if(mysock == -1)
      {
            perror("Accept failed");
            exit(1);
      }
      int incoming;

      int data_in = read(mysock,&incoming,sizeof(incoming));

      if (data_in < 0)
      {
            perror("Error reading");
            exit(1);
      }

      if(incoming  == 2)
      {
            printf("Temp task Dead\n");
            break;
      }

      }
      exit(1);
}

void uart_setup()
{
    fd = open(uart_driver, O_RDWR, O_SYNC, O_NOCTTY);
    if(fd < 0)
    {
     perror("Error opening uart driver");
    }

    my_term  = (struct termios *)malloc(sizeof(struct termios));

    termios_setup(my_term, fd);
}

void termios_setup(struct termios * my_term, int descriptor)
{
    tcgetattr(descriptor, my_term);
    my_term->c_iflag &= ~(IGNBRK | ICRNL | INLCR | PARMRK |
ISTRIP | IXON);
```

```
    my_term->c_oflag &= ~OPOST;
    my_term->c_lflag &= ~(ECHO | ECHONL | ICANON | IEXTEN |
ISIG);
    my_term->c_cflag |= CS8 | CLOCAL;

    cfsetispeed(my_term, B115200);
    cfsetospeed(my_term, B115200);

    if(tcsetattr(descriptor, TCSAFLUSH, my_term) < 0)
    {
        perror("ERROR in set attr\n");
    }
}

void read_byte(int fd,char *received)
{
    if(read(fd, received, 1)<0)
    {
           perror("Read failed");
     userLED(3,1);
    }
}


int main(int argc, char *argv[])
{
    uart_setup();
    sleep(1);
    char recv;
    int rec_data;
    char received_string[100];
    printf("Starting things\n");

    memset(log_name, '\0', sizeof(log_name));
    strncpy(log_name, argv[1], strlen(argv[1]));
    //fptr = fopen(log_name, "w");

    int i;
    int j,k=0;
    int flag1, flag2 = 0;
    message msg_struct;

    read_thread_check = pthread_create(&read_thread, NULL,
read_thread_func, NULL);
     if(read_thread_check)
     {
          perror("Error creating read thread");
          userLED(3,1);
          exit(-1);
     }

    /*write_thread_check = pthread_create(&write_thread, NULL,
```

```
write_thread_func, NULL);
      if(write_thread_check)
      {
            perror("Error creating read thread");
            exit(-1);
      }*/

    api_thread_check = pthread_create(&api_thread, NULL,
api_thread_func, NULL);
      if(api_thread_check)
      {
            perror("Error creating read thread");
            userLED(3,1);
            exit(-1);
      }

    heartbeat_check = pthread_create(&heartbeat_thread, NULL,
hb_thread_func, NULL);
      if(heartbeat_check)
      {
            perror("Error creating read thread");
            userLED(3,1);
            exit(-1);
      }

    printf("Threads created\n");
    //fptr = fopen("log_name", "a");

    //while(1)
    {
      //do
      //{
            //fptr = fopen(log_name, "a");
            //read_byte(fd, &recv);
            //fprintf(fptr, "%c", recv);
            //read(fd, &msg_struct, sizeof(msg_struct));
            //printf("%d", msg_struct.data);
            //printf("%c", recv);

            /*if(recv == 'C')
            {
                  flag1=1;
                  k=j;
                  j++;
                  //printf("C Detected");
            }
            if(recv == 'O')
            {
                  flag2 = 1;
                  k++;
                  if(j==k)
                  {
```

```
                        printf("O Detected");
                    }
               }
               if(flag2 == 1)
                    printf("ALERT DETECTED");
               flag1 = flag2 = 0;
               rec_data = (int)recv;
               fclose(fptr);*/
        //}while(rec_data != 0);
      }
      //while(1)
      {
      //char data[3] = "Nik";
      //i = write(fd, &data, sizeof(data));
//      printf("%d", i);

      //close(fd);
      //fclose(fptr);
        pthread_join(read_thread, NULL);
        //pthread_join(write_thread, NULL);
        //pthread_join(api_thread, NULL);
        //pthread_join(heartbeat_thread, NULL);
      }
}
```