```
/*****************************************************************************************
***
* File name   : bist.c
  *
* Authors     : Puneet Bansal and Nachiket Kelkar
  *
* Description : The function definition used for built in self test.
  *
* Tools used  : GNU make, gcc, arm-linux-gcc
  *
*****************************************************************************************
**/
#include "bist.h"
#include <stdio.h>
#include "temp_i2c.h"

int lightSensorBIST(int fileDesc)
{
    uint8_t* rb= malloc(6);
    rb=lightSensorRead(fileDesc,IDREG,1);
        if(rb == NULL)
        {
                return -1;
        }
    if(*rb!=0x50)
    {
        return -1;
    }
return 0;
}


int tempSensorBIST(int fileDesc)
{
    uint16_t retVal=temp_i2c_read_from_reg(fileDesc, CONFIG_REG_ADDR);
        if(retVal == 10000)
                return -1;
    if(retVal!=DEFAULT_CONFIG)
        return -1;
    return 0;
}


/*****************************************************************************************
***
* File name   : bist.h
  *
* Authors     : Puneet Bansal and Nachiket Kelkar
  *
* Description : The function declarations used for built in self test.
  *
* Tools used  : GNU make, gcc, arm-linux-gcc
  *
*****************************************************************************************
**/
#include "lightsensor.h"

/*
 *@name: lightSensorBIST
 *@param: i2c file descriptor
 *@description: reads from the ID register of the light sensor and compares it with the defaul
t
 *               value to make sure sensor is powered on and the I2c communication is active.
```

```
 */
int lightSensorBIST(int fileDesc);

/*
 * @name: tempSensorBIST
 * @param: i2c file descriptor
 * @description: reads from the configuration register of the temp sensor and compares it with
 *        the default value to make sure sensor is powered on and the I2c communication is act
ive.
 */
int tempSensorBIST(int fileDesc);


/**************************************************************************************************
***
* File name   : gpio.c
  *
* Authors     : Nachiket Kelkar and Puneet Bansal
  *
* Description : The functions used for gpio operations. Setting the direction of pin and
  *
*               the value. This functions are restricted for use of only USER LED pins.
  *
* Tools used  : GNU make, gcc, arm-linux-gcc.
  *
**************************************************************************************************
**/
#define _GNU_SOURCE

/* Including standard libraries */
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <signal.h>
#include <unistd.h>
#include <fcntl.h>
#include <linux/gpio.h>

/* Including user libraries */
#include "gpio.h"


void gpio_init(int gpio_pin,int gpio_direction)
{
        FILE *fp;
        char *file = (char*)malloc(40);

        if(is_pin_valid(gpio_pin))
        {
                fp = fopen("/sys/class/gpio/export","w");
                fprintf(fp,"%d",gpio_pin);
                fclose(fp);
                sprintf(file,"/sys/class/gpio/gpio%d/direction",gpio_pin);
                fp = fopen(file,"w");
                if(gpio_direction == out)
                {
                        fprintf(fp,"out");
                }
                else if(gpio_direction == in)
                {
                        fprintf(fp,"in");
                }
                else
```

```
                {
                        printf("Enter direction only as in or out");
                }
                fclose(fp);
        }
        else
        {
                printf("Enter valid pin number");
        }
                free(file);
}


void gpio_write_value(int gpio_pin, int gpio_value)
{
        FILE *fp;
        char *file = (char*)malloc(40);

        if(is_pin_valid(gpio_pin))
        {
                sprintf(file,"/sys/class/gpio/gpio%d/value",gpio_pin);
                fp = fopen(file,"w");
                if(gpio_value == low)
                {
                        fprintf(fp,"%d",low);
                }
                else if(gpio_value == high)
                {
                        fprintf(fp,"%d",high);
                }
                else
                {
                        printf("Enter value only as low or high");
                }
                fclose(fp);
        }
        else
        {
                printf("Enter valid pin number");
        }
        free(file);
}


int gpio_read_value(int gpio_pin)
{
        FILE *fp;
        char *file = (char*)malloc(40);
        int value;

        if(is_pin_valid(gpio_pin))
        {
                sprintf(file,"/sys/class/gpio/gpio%d/value",gpio_pin);
                fp = fopen(file,"r");
                fscanf(fp,"%d",&value);
                fclose(fp);
        }
        else
        {
                printf("Enter valid pin number");
        }
        free(file);
        return value;
```

```
}


bool is_pin_valid(int gpio_pin)
{
        int gpio_allowed[total_gpio] = access_pin_allowed;
        bool is_valid = false;

        for(int i=0; i<total_gpio; i++)
        {
                if(gpio_pin == gpio_allowed[i])
                        is_valid = is_valid | true;
                else
                        is_valid = is_valid | false;
        }
        return is_valid;
}


void gpio_interrupt_state(int gpio_pin, gpio_interrupt interrupt)
{
        FILE *fp;
        char *file = (char*)malloc(40);

        if(is_pin_valid(gpio_pin))
        {
                gpio_init(gpio_pin,in);
                sprintf(file,"/sys/class/gpio/gpio%d/edge",gpio_pin);
                fp = fopen(file,"w");
                switch(interrupt)
                {
                case rising:
                        fprintf(fp,"rising");
                        break;
                case falling:
                        fprintf(fp,"falling");
                        break;
                case both:
                        fprintf(fp,"both");
                        break;
                case none:
                        fprintf(fp,"none");
                        break;
                }
                fclose(fp);
        }
        else
        {
                printf("Enter valid pin number");
        }
        free(file);
}


int gpio_open_value(int gpio_pin)
{
        char *file = (char*)malloc(40);
        int fd;
        if(is_pin_valid(gpio_pin))
        {
                sprintf(file,"/sys/class/gpio/gpio%d/value",gpio_pin);
                fd = open(file, O_RDONLY);
        }
```

```
        else
        {
                printf("Enter valid pin number");
                fd = -1;
        }
        return fd;
}


int gpio_read_val_with_fd(int fd)
{
        int value;
        read(fd, &value, sizeof(value));
        lseek(fd, 0, SEEK_SET);
        return value & 0x1;
}



/*****************************************************************************************
***
* File name   : gpio.h
  *
* Authors     : Nachiket Kelkar and Puneet Bansal
  *
* Description : The functions used for gpio operations. Setting the direction of pin and
  *
*               the value.
  *
* Tools used  : GNU make, gcc, gcc-linux-gcc.
  *
*****************************************************************************************
**/
#include <stdbool.h>

#define total_gpio 5
#define access_pin_allowed {53,54,55,56,60}

/*************** Enumerations used for gpio direction and gpio value ***************/
enum gpio_direction{
in = 0,
out,
};

enum gpio_value{
low = 0,
high,
};

typedef enum{
falling,
rising,
both,
none,
}gpio_interrupt;

/********************** Functions for the gpio operations *********************/
/*
* Function name:- gpio_init
* Description:- The function takes the gpio pin number and assignes it as input pin or
*               output pin.
* @param:- int (gpio pin number), int (gpio pin direction)
* @return:- void
```

```
* gpio pin direction - 0 for in and 1 for out.
*/
void gpio_init(int,int);


/*
* Function name:- gpio_write_value
* Description:- The function takes the gpio pin number and outputs the pin high or low.
* @param:- int (gpio pin number), int (gpio pin value)
* @return:- void
* gpio pin direction - 0 for in and 1 for out.
*/
void gpio_write_value(int,int);


/*
* Function name:- gpio_read_value
* Description:- The function takes the gpio pin number and returns the value on the pin.
* @param:- int (gpio pin number), int (gpio pin value)
* @return:- int (value high or low)
*/
int gpio_read_value(int);


/*
* Function name:- is_pin_valid
* Description:- The function takes the gpio pin number and returns if valid pin no is entered.
* @param:- int (gpio pin number)
* @return:- bool (true if pin number is valid and false if not)
* gpio pin direction - 0 for in and 1 for out.
* Need to maintain pin values and no of valid pins in above define.
*/
bool is_pin_valid(int);


/*
* Function name:- gpio_interrupt_state
* Description:- The function takes the gpio pin number and sets the gpio interrupt as rising
*               falling, both or none based on second parameter.
* @param:- int (gpio pin number), gpio_interrupt (which interrupt);
* @return:- void
* Comments:- gpio_interrupt: can be falling, rising, both or none to disable the interrupts.
*            Need to maintain pin values and no of valid pins in above define.
*/
void gpio_interrupt_state(int, gpio_interrupt);


/*
* Function name:- gpio_open_value
* Description:- The function takes the gpio pin number and opens the file and returns the
*               file descriptor.
* @param:- int (gpio pin number)
* @return:- int (file descriptor)
* Comments:- Need to maintain pin values and no of valid pins in above define.
*/
int gpio_open_value(int);


/*
* Function name:- gpio_read_val_with_fd
* Description:- The function takes the file descriptior of gpio pin and reurnts the state of t
he
*               pin wether high or low.
```

```
* @param:- int (file descriptor)
* @return:- int (pin state)
* Comments:- pin state: Pin state can be high or low.
*/
int gpio_read_val_with_fd(int);


/*****************************************************************************************
***
* File name   : lightsonsor.c
  *
* Authors     : Puneet Bansal and Nachiket Kelkar
  *
* Description : The function definition used for communication with light sensor.
  *
* Tools used  : GNU make, gcc, arm-linux-gcc
  *
*****************************************************************************************
**/
#include<stdio.h>
#include"myi2c.h"
#include<stdint.h>
#include"lightsensor.h"

uint16_t data0Val;
uint16_t data1Val;
float lux;
float ch1ch0Ratio;


uint8_t* lightSensorRead(int fileDescrip, uint8_t regAdd, uint8_t len)
{
        uint8_t* readBuffer= malloc(10);
        uint8_t writeBuffer[10];
        uint8_t cmdReg=0;
        int ret;
        if(len==1)
        {
            cmdReg=CMDBYTE;
        }
        else if(len==2)
        {
            cmdReg=CMDWORD;
        }
        else if(len<=0 || len>2)
        {
        exit(1);
        }
        *writeBuffer=(cmdReg | regAdd);
        ret=myi2cWrite(fileDescrip,writeBuffer,1);
        readBuffer=myi2cRead(fileDescrip,len);
        if(readBuffer == NULL)
        {
                perror("Read failed");
                return NULL;
        }
        return readBuffer;
}



int lightSensorWrite(int fileDescrip,uint8_t regAdd, uint16_t data, uint8_t len)
{
```

```
        uint8_t writeBuffer[10];
        int ret;

        uint8_t cmdReg=0;
        if(len==1)
        {
            cmdReg=CMDBYTE;
        }
        else if(len==2)
        {
            cmdReg=CMDWORD;
        }
        else if(len<=0 || len>2)
        {
            printf("Invalid len");
            exit(1);
        }
        writeBuffer[0]=(cmdReg | regAdd);
        writeBuffer[1]=data;
        ret=myi2cWrite(fileDescrip,writeBuffer,len);
        if(ret < 0)
        {
                perror("Write failed");
                return -1;
        }
        return 0;
}


float luxCalc(int fd)
{
    data0Val=0;
    data1Val=0;
    ch1ch0Ratio=0;
    lux=0;

    uint8_t* readBuffer=malloc(2);
    readBuffer=lightSensorRead(fd,DATA0LOW,2);
        if(readBuffer == NULL)
        {
                perror("Read failed");
                return -1;
        }

    data0Val=readBuffer[1]<<8;
    data0Val |= readBuffer[0];

    readBuffer=lightSensorRead(fd,DATA1LOW,2);
        if(readBuffer == NULL)
        {
                perror("Read failed");
                return -1;
        }

    data1Val=readBuffer[1]<<8;
    data1Val |= readBuffer[0];

    float temp0=data0Val;
    float temp1=data1Val;


    ch1ch0Ratio=temp1/temp0;
```

```
        if(ch1ch0Ratio>0 && ch1ch0Ratio<=0.50)
        {
                lux= ( ( (0.0304)*data0Val ) - (0.062 * data1Val * pow(ch1ch0Ratio,1.4)) );
        }
        else if(ch1ch0Ratio>0.50 && ch1ch0Ratio<=0.61)
        {
                lux= ( ( (0.0224)*data0Val ) - (0.031 * data1Val) );
        }
        else if(ch1ch0Ratio>0.61 && ch1ch0Ratio<=0.80)
        {
                lux= ( ( (0.0128)*data0Val ) - (0.0153 * data1Val) );
        }
        else if(ch1ch0Ratio>0.80 && ch1ch0Ratio<=1.30)
        {
                lux= ( ( (0.00146)*data0Val ) - (0.00112 * data1Val) );
        }
        else if(ch1ch0Ratio>1.30)
        {
                lux=0;
        }

        return lux;
}


/*****************************************************************************************
***
* File name   : lightsonsor.h
  *
* Authors     : Puneet Bansal and Nachiket Kelkar
  *
* Description : The function declaration used for communication with light sensor.
  *
* Tools used  : GNU make, gcc, arm-linux-gcc
  *
*****************************************************************************************
**/
#include <stdint.h>
#include <stdlib.h>
#include "myi2c.h"
#include <math.h>

uint8_t readBuffer[10];

#define slaveAddFloat 0x39
#define slaveAddGnd 0x29
#define slaveAddVdd 0x49

/*Light Sensor Register addressess*/
#define CMDBYTE 0x80
#define CMDWORD 0xA0
#define CNTRLREG 0x00
#define INTCTL 0x06
#define DATA0LOW 0x0c
#define DATA0HIGH 0x0d
#define DATA1LOW 0x0e
#define TIMINGREG 0x01
#define THRESHLOWLOW 0x02
#define THRESHLOWHIGH 0x03
#define THRESHHIGHLOW 0x04
#define THRESHHIGHHIGH 0x05
#define INTCNTL 0X06
```

```
#define IDREG 0x0a


/**
 * @name : lightsensorRead
 *
 * @description: Function to read from the specified i2c registers using i2c.
 * The register address from where data is to be read, is first written via myi2cwrite functio
n, which is basically a wrapper to write to the file
 * specified by the filedescriptor. After this a read operation of the required number of byte
s is performed via myi2cread function. The value received
 * is written on the buffer and returned.
 *
 *
 * @param1: filedescrip- i2c file descriptor
 * @param2: regadd- light sensor register to read fromm
 * @param3: len- number of bytes that should be read (1/2 bytes)
 *
 *
 * @return type: (uint8_t*)- character buffer consisting of the value read from the i2c read f
unction.
 *
 * */
uint8_t* lightSensorRead(int, uint8_t, uint8_t);


/**
 * @name: lightSensorWrite
 * @description: Writes the specified number of bytes(len) of data (specified in parameters) t
o the register (specified in parameter)
 *
 * @param1: fileDescrip- i2c file descriptor
 * @param2: regAdd- light sensor register to write to
 * @param3: data: 16bit/8bit data to write to the register.
 * @param4: len: number of bytes tot write (1/2)
 * */
int lightSensorWrite(int fileDescrip,uint8_t, uint16_t, uint8_t);


/**
 * @name: luxCalc
 * @desc: reads from adc channel 0 and adc channel 1 using lightSensorRead function. Does the
necessary computations to calculate LUX and
 * returns the lux value in float to the user.
 *
 * @param1- i2c file descriptor
 *
 * @return type:  float - lux value in float.*
 *
 * */
float luxCalc(int);


/*****************************************************************************************
***
* File name   : logger.c
  *
* Authors     : Puneet Bansal and Nachiket Kelkar
  *
* Description : The function definition used for communication by logger task.
  *
* Tools used  : GNU make, gcc, arm-linux-gcc
  *
```

```
*****************************************************************************************
**/
#include "logger.h"
extern int count;


void logToFile(char *fileName, logStruct dataToReceive)
{

    //printf("value of count is %d",count);
    FILE *logging;
    char level[20];
    //char state[20];


    if(dataToReceive.logLevel==alert)
    {
        strcpy(level,"[ALERT]");
    }
    else if(dataToReceive.logLevel==info)
    {
        strcpy(level,"[INFO]");
    }
    else
    {
        strcpy(level,"[DEBUG]");
    }


    //mainInfoToOthers fileInfo;
    //printf("file name in logtofile function is %s\n",fileName);
        //printf("debug level is %s\n",level);
    //printf("value of timestamp in logtofile function is %s\n",printTimeStamp());
    if(strcmp(dataToReceive.source,"temperature")==0)
    {
       //printf("Recevied unit is %s\n",dataToReceive.unit);
        if(count ==1)
        {
            logging = fopen(fileName,"w");
            count=0;
        }
        else
        {
            logging = fopen(fileName,"a");
        }


       if(dataToReceive.status==success)
       {
            fprintf(logging,"%s %s [%s] : %f %s\n",printTimeStamp(),level,dataToReceive.source
,dataToReceive.value,dataToReceive.unit);

        }
       else
       {
           fprintf(logging, "%s %s [%s] (Failure message) - %s\n",printTimeStamp(),level,dataT
oReceive.source,dataToReceive.message);
       }
    fclose(logging);
    }

    if(strcmp(dataToReceive.source,"light")==0)
    {
```

```
       if(count ==1)
        {
            logging = fopen(fileName,"w");
            count=0;
        }
        else
        {
            logging = fopen(fileName,"a");
        }

    if(dataToReceive.status==success)
       {
            fprintf(logging,"%s %s [%s] : %f\n",printTimeStamp(),level,dataToReceive.source,da
taToReceive.value);
        }
       else
       {
           fprintf(logging, "%s %s [%s] %s\n",printTimeStamp(),level,dataToReceive.source,data
ToReceive.message);
       }

    fclose(logging);

    }

    if(strcmp(dataToReceive.source,"socket")==0)
    {
       if(count ==1)
        {
            logging = fopen(fileName,"w");
            count=0;
        }
        else
        {
            logging = fopen(fileName,"a");
        }

        fprintf(logging, "%s %s [%s] %s\n",printTimeStamp(),level,dataToReceive.source,dataToR
eceive.message);


    fclose(logging);

    }

    if(strcmp(dataToReceive.source,"main")==0)
    {
       if(count ==1)
        {
            logging = fopen(fileName,"w");
            count=0;
        }
        else
        {
            logging = fopen(fileName,"a");
        }

        fprintf(logging, "%s %s [%s] %s\n",printTimeStamp(),level,dataToReceive.source,dataToR
eceive.message);


    fclose(logging);
```

```
    }
}

char* printTimeStamp()
{
    char* time_stamp=malloc(40);
    struct timespec thTimeSpec;
    clock_gettime(CLOCK_REALTIME, &thTimeSpec);
    sprintf(time_stamp,"[s: %ld, ns: %ld]",thTimeSpec.tv_sec,thTimeSpec.tv_nsec);
    //printf("Value of time_stamp is %s",time_stamp);
    return time_stamp;
}


/*****************************************************************************************
***
* File name   : logger.c
  *
* Authors     : Puneet Bansal and Nachiket Kelkar
  *
* Description : The function declaration used for communication by logger task.
  *
* Tools used  : GNU make, gcc, arm-linux-gcc
  *
*****************************************************************************************
**/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include "mq.h"

/**
 * @name: logToFile
 *
 * @desc: Takes the structure with the data as a parameter. Prints this data to the file along
 with the timestamp, source of the message, loglevel,
 * value and unit.
 * The source can be lightsensor,tempsensor, maintask and sockettask.
 * Three log levels are defined.
 * 1)DEBUG: to print general debug information
 * 2)INFO: printing the temp and light sensor values
 * 3)ALERT: important error messages.
 *
 * @param1: char*: log file name
 * @param2: logStruct: the populated logger structure that needs to be printed to the file
 *
 * */
void logToFile(char *, logStruct);

/**
 * name: printTimeStamp
 *
 * @desc: takes the present timestamp using clock_gettime function, converts it into a string
and returns that string.
 *
 * @return : (char*)- timestamp string.
 * */
char *printTimeStamp();

/*****************************************************************************************
***
* File name   : maintask.c
```

```
  *
* Authors     : Nachiket Kelkar and Puneet Bansal
  *
* Description : The main logic of the code
  *
* Tools used  : GNU make, gcc, arm-linux-gcc
  *
*****************************************************************************************
**/

#include <unistd.h>
#include <sys/syscall.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h>
#include <time.h>
#include <stdbool.h>
#include <stdint.h>
#include <netdb.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <poll.h>

/*User defined headers*/
#include "maintask.h"
#include "temp_i2c.h"
#include "lightsensor.h"
#include "gpio.h"
#include "bist.h"
#include "myi2c.h"

typedef struct
{
  char *logFileName;
  int noOfParam;
}mainInfoToOthers;

typedef enum
{
    init,
    dark,
    light
}state;

int fd,fd1;

pthread_t tempSensorTask,lightSensorTask,loggerTask,socketTask;
static void signal_handler(int , siginfo_t *, void*);
bool measureTemperature = true;
bool measureLight=false;
bool loggerHeartBeat=false;
bool socketHeartBeat=false;
bool exitThread = false;


/*****************************************************************************
* Function name:- tempSensorRoutine                                          *
* Description:- This function is used by a Temperature sensor thread for execution of *
*               temperature sensor task.                                     *
* @param:- void* (data from main)                                            *
* @return:- static void* (pthread exit value)                                *
```

```
*******************************************************************************/
static void* tempSensorRoutine(void *dataObj)
{
        mainStruct dataToSendToMain;
        logStruct dataToSendToLogger;
        socketStruct dataToSendToSocket;
        tempStruct dataReceived;
        int i2c_file_des;
        float temperature;
        uint16_t config;
        int gpio_val_fd;
        int gpio_state;
        struct pollfd fds;

        int init_status = init_success;
        struct sigevent tempEvent;
        struct sigaction tempAction;
        struct itimerspec timerSpec;
        timer_t tempTimer;
        bool sendTempToSocket = false;

        strcpy(dataToSendToMain.source,"temperature");
        dataToSendToMain.messageType  = update;

        /* Initialize gpio LED pin as output and pin 60 for interrupts */
        gpio_init(53, out);
        gpio_interrupt_state(60, both);
        gpio_val_fd = gpio_open_value(60);

        fds.fd = gpio_val_fd;
        fds.events = POLLPRI | POLLERR;

        /* Setting up all queues */
    mqd_t tempToMain = mqueue_init(MAINQUEUENAME, MAIN_QUEUE_SIZE, sizeof(dataToSendToMain));
        mqd_t tempToLogger = mqueue_init(LOGQUEUENAME, LOG_QUEUE_SIZE, sizeof(dataToSendToLogg
er));
        mqd_t tempQueue = mqueue_init(TEMPQUEUENAME, TEMP_QUEUE_SIZE, sizeof(dataReceived));
        mqd_t socketQueue = mqueue_init(SOCKETQUEUENAME, SOCKET_QUEUE_SIZE, sizeof(dataToSendT
oSocket));

        if(tempToMain < 0)
    {
        perror("TemptoMain queue creation failed");
    }
        if(tempToLogger <0)
        {
                perror("TempToLog queue creation failed");
        }
        if(tempQueue < 0)
    {
        perror("Temp queue creation failed");
    }
        if(socketQueue < 0)
    {
        perror("socketQueue creation failed");
    }

        dataToSendToLogger.logLevel = alert;
        dataToSendToLogger.status = fail;
        dataToSendToLogger.source= "temperature";

        /* Register signal handler for a signal*/
        tempAction.sa_flags     = SA_SIGINFO;
```

```
    tempAction.sa_sigaction = signal_handler;
        if((sigaction(SIGRTMIN + 4, &tempAction, NULL))<0)
        {
                perror("Failed setting signal handler for temp measurement");
                dataToSendToLogger.message = "Signal handler set failed";
                mq_send(tempToLogger,(char*)&dataToSendToLogger,sizeof(logStruct),0);
                init_status = init_failure;
        }

        /* Assigning signal to timer */
        tempEvent.sigev_notify             = SIGEV_SIGNAL;
        tempEvent.sigev_signo              = SIGRTMIN + 4;
        tempEvent.sigev_value.sival_ptr    = &tempTimer;
        if((timer_create(CLOCK_REALTIME, &tempEvent, &tempTimer)) < 0)
        {
                perror("Timer creation failed for temperature task");
                dataToSendToLogger.message = "Assigning signal failed";
                mq_send(tempToLogger,(char*)&dataToSendToLogger,sizeof(logStruct),0);
                init_status = init_failure;
        }

        /* Setting the time and starting the timer */
        timerSpec.it_interval.tv_nsec = 200000000;
        timerSpec.it_interval.tv_sec  = 0;
        timerSpec.it_value.tv_nsec    = 200000000;
        timerSpec.it_value.tv_sec     = 0;

        if((timer_settime(tempTimer, 0, &timerSpec, NULL)) < 0)
        {
                perror("Starting timer in temperature task failed");
                dataToSendToLogger.message = "Failed to start timer";
                mq_send(tempToLogger,(char*)&dataToSendToLogger,sizeof(logStruct),0);
                init_status = init_failure;
        }

        /* Open i2c file to start communication with temperature sensor. */

        i2c_file_des = fd1;

        /* Initialize temperature sensor lower limit alert register */
        int success = temp_i2c_write_to_reg(i2c_file_des, TLOW_REG_ADDR, 26);
        if(success < 0)
        {
                printf("Write failure\nExiting Temperature measurement task\n");
                dataToSendToLogger.message = "Write failure";
                mq_send(tempToLogger,(char*)&dataToSendToLogger,sizeof(logStruct),0);
                init_status = init_failure;
                gpio_write_value(53,high);
        }

        /* Initialize temperature sensor higer limit alert register. */
        success = temp_i2c_write_to_reg(i2c_file_des, THIGH_REG_ADDR, 30);
        if(success < 0)
        {
                printf("Write failure\nExiting Temperature measurement task\n");
                dataToSendToLogger.message = "Write failure";
                mq_send(tempToLogger,(char*)&dataToSendToLogger,sizeof(logStruct),0);
                init_status = init_failure;
                gpio_write_value(53,high);
        }

        /* Send inititalization success/fail message to main. */
        dataToSendToMain.status = init_status;
```

```
        int x = mq_send(tempToMain,(char*)&dataToSendToMain,sizeof(dataToSendToMain),0);
        if(x < 0)
        {
                perror("Sending data failed");
                dataToSendToLogger.message = "Fail sending temp init success";
                mq_send(tempToLogger,(char*)&dataToSendToLogger,sizeof(logStruct),0);
        }
        if(init_status == init_failure)
        {
                dataToSendToLogger.message = "Temperature sensor init failure";
                mq_send(tempToLogger,(char*)&dataToSendToLogger,sizeof(logStruct),0);
                pthread_exit(NULL);
        }

        strcpy(dataToSendToMain.source,"temperature");
        dataToSendToLogger.source= "temperature";
        strcpy(dataToSendToLogger.unit,"Celsius");
        while(1)
        {
                /* Do only each 100msec */
                if(measureTemperature)
                {
                        int ret = mq_receive(tempQueue,(char*)&dataReceived,sizeof(dataReceive
d),0);
                        if(ret < 0)
                        {
                                //perror("No requests received");
                        }
                        else
                        {
                                sendTempToSocket = true;
                        }
                        measureTemperature = false;
                        dataToSendToMain.messageType = update;
                        dataToSendToMain.status = 1;

                        dataToSendToLogger.status = 1;

                        /* Get the temperature values in Celsius */
                        temperature = read_temperature(i2c_file_des, TEMP_REG_ADDR);
                        if(temperature == 10000)
                        {
                                printf("Temperature sensor pulled out\nExiting Temperature Sen
sor Task\n");
                                dataToSendToLogger.logLevel = alert;
                                dataToSendToLogger.status = fail;
                                dataToSendToLogger.source= "temperature";
                                dataToSendToLogger.message = "Read from sensor failed";
                                mq_send(tempToLogger,(char*)&dataToSendToLogger,sizeof(logStru
ct),0);
                                gpio_write_value(53,1);
                                close(i2c_file_des);
                                mq_close(tempToMain);
                                mq_close(tempToLogger);
                                mq_close(tempQueue);
                                mq_close(socketQueue);
                                mq_unlink(MAINQUEUENAME);
                                mq_unlink(TEMPQUEUENAME);
                                mq_unlink(SOCKETQUEUENAME);
                                mq_unlink(LOGQUEUENAME);
                                timer_delete(tempTimer);
                                pthread_exit(NULL);
                        }
```

```
                        dataToSendToLogger.logLevel = info;
                        dataToSendToLogger.value = temperature;

                        /* Send heartbeat message to main */
                        mq_send(tempToMain,(char*)&dataToSendToMain,sizeof(mainStruct),0);
                        /* Send temperature values to logger */
                        mq_send(tempToLogger,(char*)&dataToSendToLogger,sizeof(logStruct),0);

                        /* Send temperature values to socket only if it is requested. */
                        if(sendTempToSocket)
                        {
                                sendTempToSocket = false;
                                dataToSendToSocket.source = "temperature";
                                if(strcmp(dataReceived.unit,"Celsius") == 0)
                                {
                                        temperature = convert_to_unit(temperature,Celsius);
                                }
                                else if(strcmp(dataReceived.unit,"Fahrenheit") == 0)
                                {
                                        temperature = convert_to_unit(temperature,Fahrenheit);
                                }
                                else if(strcmp(dataReceived.unit,"Kelvin") == 0)
                                {
                                        temperature = convert_to_unit(temperature,Kelvin);
                                }
                                dataToSendToSocket.value  = temperature;
                                int ret = mq_send(socketQueue, (char*)&dataToSendToSocket, siz
eof(dataToSendToSocket),0);
                                if (ret < 0)
                                {
                                        perror("Temperature sending to socket failed");
                                        dataToSendToLogger.logLevel = alert;
                                        dataToSendToLogger.status = fail;
                                        dataToSendToLogger.source= "temperature";
                                        strcpy(dataToSendToLogger.message,"Sending temp to soc
ket failed");
                                        mq_send(tempToLogger,(char*)&dataToSendToLogger,sizeof
(logStruct),0);
                                }
                        }
                        if(exitThread)
                        {
                                break;
                        }
                }
                int ret = poll(&fds, 1, 1);
                if(ret > 0)
                {
                        gpio_state = gpio_read_val_with_fd(gpio_val_fd);
                        if(gpio_state == 0x01)
                                printf("Temperature is below 26C\n");
                        else
                                printf("Temperature rise above 30C\n");
                }
        }
        printf("_____________________Temperature task exiting_____________________\n");
        /* Cleanup all initializations */
        close(i2c_file_des);
        mq_close(tempToMain);
        mq_close(tempToLogger);
        mq_close(tempQueue);
        mq_close(socketQueue);
        mq_unlink(MAINQUEUENAME);
```

```
        mq_unlink(TEMPQUEUENAME);
        mq_unlink(SOCKETQUEUENAME);
        mq_unlink(LOGQUEUENAME);
        timer_delete(tempTimer);
    pthread_exit(NULL);
}


/*****************************************************************************
* Function name:- lightSensorRoutine                                         *
* Description:- This function is used by a light sensor thread for execution of *
*               light sensor task.                                           *
* @param:- void* (data from main)                                            *
* @return:- static void* (pthread exit value)                               *
*****************************************************************************/
static void* lightSensorRoutine(void *dataObj)
{
    mainStruct dataToSendToMain;
        logStruct dataTOSendTOLogger;
        lightStruct dataReceivedFromMain;
        socketStruct dataToSendToSocket;
        static state presentState, prevState;
    presentState=init;
    prevState=init;

        int fd;
        bool sendLightToSocket=false;

        strcpy(dataToSendToMain.source,"light");
        dataToSendToMain.messageType = update;
        dataToSendToMain.status=init_success;

        dataTOSendTOLogger.source="main";
        dataTOSendTOLogger.logLevel=alert;
        dataTOSendTOLogger.status=fail;

        /* Initialize gpio LED pin as output */
        gpio_init(54, out);

        /* Setting up queues */
    mqd_t lightToMain = mqueue_init(MAINQUEUENAME,MAIN_QUEUE_SIZE,sizeof(dataToSendToMain));
        mqd_t lightToLogger = mqueue_init(LOGQUEUENAME,LOG_QUEUE_SIZE,sizeof(logStruct));
        mqd_t lightQueue = mqueue_init(LIGHTQUEUENAME,LIGHT_QUEUE_SIZE,sizeof(lightStruct));
        mqd_t lightToSocket = mqueue_init(SOCKETQUEUENAME,SOCKET_QUEUE_SIZE,sizeof(socketStruc
t));

        if(lightToMain < 0)
    {
        perror("Light queue creation failed");
                dataToSendToMain.status=init_failure;
                dataTOSendTOLogger.message="Light To Main queue creation failed";
                mq_send(lightToLogger,(char*)&dataTOSendTOLogger,sizeof(logStruct),0);
    }
    if(lightToLogger<0)
        {
                perror("LightToLogger queue creation failed");
                dataToSendToMain.status=init_failure;
                dataTOSendTOLogger.message="LightToLogger queue creation failed";
                mq_send(lightToLogger,(char*)&dataTOSendTOLogger,sizeof(logStruct),0);
        }

        if(lightQueue<0)
        {
```

```
                perror("LightQueue queue creation failed");
                dataToSendToMain.status=init_failure;
                dataTOSendTOLogger.message="LightQueue queue creation failed";
                mq_send(lightToLogger,(char*)&dataTOSendTOLogger,sizeof(logStruct),0);
        }

        if(lightToSocket<0)
        {
                perror("LightToSocket Queue queue creation failed");
                dataToSendToMain.status=init_failure;
                dataTOSendTOLogger.message="LightToSocket Queue queue creation failed";
                mq_send(lightToLogger,(char*)&dataTOSendTOLogger,sizeof(logStruct),0);
        }

        /* Initialising I2C */
        fd=myi2cInit(slaveAddFloat);
        int ret = lightSensorWrite(fd,CNTRLREG,0x03,2); //writing to control register.
        if(ret < 0)
        {
                dataTOSendTOLogger.message="Writing to control register failed";
                mq_send(lightToLogger,(char*)&dataTOSendTOLogger,sizeof(logStruct),0);
                gpio_write_value(54,high);
        }

        /* Initialising Timer */
    struct sigevent lightEvent;
        struct sigaction lightAction;
        struct itimerspec lightTimerSpec;
        timer_t lightTimer;

        /* Assigning signal handler to a signal */
        lightAction.sa_flags = SA_SIGINFO;
        lightAction.sa_sigaction = signal_handler;


        if(sigaction(SIGRTMIN + 5 , &lightAction, NULL)<0)
        {
                perror("Light sensor, timer handler initialisation failed");
                dataToSendToMain.status=init_failure;
                dataTOSendTOLogger.message="Light sensor, timer handler initialisation failed"
;
                mq_send(lightToLogger,(char*)&dataTOSendTOLogger,sizeof(logStruct),0);

        }

        /* Setting up timer */
        lightEvent.sigev_notify = SIGEV_SIGNAL;
    lightEvent.sigev_signo = SIGRTMIN + 5;
    lightEvent.sigev_value.sival_ptr = &lightTimer;

        if((timer_create(CLOCK_REALTIME, &lightEvent, &lightTimer)) < 0)
        {
                perror("Timer creation failed for temperature task");
                dataToSendToMain.status=init_failure;
                dataTOSendTOLogger.message="Timer creation failed for temperature task";
                mq_send(lightToLogger,(char*)&dataTOSendTOLogger,sizeof(logStruct),0);
                //exit(1);
        }

        /* Start the timer */
        lightTimerSpec.it_interval.tv_nsec = 100000000; //To get the value after every 100 ms
        lightTimerSpec.it_interval.tv_sec  = 0;
        lightTimerSpec.it_value.tv_nsec    = 100000000;
```

```
        lightTimerSpec.it_value.tv_sec     = 0;

        if(timer_settime(lightTimer,0,&lightTimerSpec,NULL)<0)
        {
                perror("Light sensor timer settime failed");
                dataToSendToMain.status=init_failure;
                dataTOSendTOLogger.message="Light sensor timer settime failed";
                mq_send(lightToLogger,(char*)&dataTOSendTOLogger,sizeof(logStruct),0);
                //exit(1);
        }

        /*Sending initialisation success/fail message*/
        mq_send(lightToMain,(char*)&dataToSendToMain,sizeof(mainStruct),0);

        int y=0;
        float lux=0;
        while(1)
        {
                if(measureLight)
                {
                        if(mq_receive(lightQueue,(char*)&dataReceivedFromMain,sizeof(lightStru
ct),0)>-1)
                        {
                                sendLightToSocket=true;
                        }
                        measureLight=false;

                        //take the light sensor value and send it to logger task
                        dataToSendToMain.messageType = update;
                        dataToSendToMain.status = success;

                        dataTOSendTOLogger.source= "light";
                        dataTOSendTOLogger.logLevel=alert;
                        dataTOSendTOLogger.status=fail;

                        lux=luxCalc(fd);
                        if(lux < 0)
                        {
                                printf("Light sensor pulled out\n Exiting Light task\n");
                                dataTOSendTOLogger.message = "Failed reading the light value";
                                mq_send(lightToLogger,(char*)&dataTOSendTOLogger,sizeof(dataTO
SendTOLogger),0);
                                gpio_write_value(54,high);
                                close(fd);
                                mq_close(lightToMain);
                                mq_close(lightToLogger);
                                mq_close(lightQueue);
                                mq_close(lightToSocket);

                                mq_unlink(MAINQUEUENAME);
                                mq_unlink(LIGHTQUEUENAME);
                                mq_unlink(SOCKETQUEUENAME);
                                mq_unlink(LOGQUEUENAME);
                                timer_delete(lightTimer);
                                pthread_exit(NULL);
                                }

                        if(lux>50)
                {
                        presentState=light;
                    if(prevState!=light)
                    {
                        printf("State changed to Light\n");
```

```
                                dataTOSendTOLogger.message="State changed to Light";
                        prevState=presentState;
                                mq_send(lightToLogger,(char*)&dataTOSendTOLogger,sizeo
f(dataTOSendTOLogger),0);
                        }
                }
                else
                {
                        presentState=dark;
                        if(prevState!=dark)
                    {
                        printf("State changed to Dark\n");
                                dataTOSendTOLogger.message="State changed to Dark";
                            prevState=presentState;
                                mq_send(lightToLogger,(char*)&dataTOSendTOLogger,sizeo
f(dataTOSendTOLogger),0);
                        }
                }

                        dataTOSendTOLogger.status = success;
                        dataTOSendTOLogger.source= "light";
                        dataTOSendTOLogger.value = lux;
                        dataTOSendTOLogger.logLevel=info;
                        int ret=mq_send(lightToMain,(char*)&dataToSendToMain,sizeof(mainStruct
),0);
                        int ret1=mq_send(lightToLogger,(char*)&dataTOSendTOLogger,sizeof(dataT
OSendTOLogger),0);
                        if(ret1==-1) perror("sending data to log queue failed");
                        if(sendLightToSocket)
                        {
                                sendLightToSocket=false;
                                dataToSendToSocket.source="light";
                                dataToSendToSocket.value=lux;
                                if(mq_send(lightToSocket,(char*)&dataToSendToSocket,sizeof(soc
ketStruct),0)<0)
                                {
                                        perror("sending data to socket failed\n");
                                }
                        }
                        y++;
                }
            if(exitThread)
                {
                        break;
                }
        }


        printf("_____________________Light task exiting_____________________\n");
        close(fd);
        mq_close(lightToMain);
        mq_close(lightToLogger);
        mq_close(lightQueue);
        mq_close(lightToSocket);

        mq_unlink(MAINQUEUENAME);
        mq_unlink(LIGHTQUEUENAME);
        mq_unlink(SOCKETQUEUENAME);
        mq_unlink(LOGQUEUENAME);
        timer_delete(lightTimer);
        pthread_exit(NULL);
}
```

```
/*****************************************************************************
* Function name:- socketRoutine                                              *
* Description:- This function is used by a socket sensor thread for execution of *
*               socket sensor task.                                          *
* @param:- void* (data from main)                                            *
* @return:- static void* (pthread exit value)                                *
*****************************************************************************/
static void* socketRoutine(void *dataObj)
{
    printf("Entered socketRoutine\n");
        int serverSocket,clientConnected,len;
        struct sockaddr_in clientAddr,serverAddr;
        struct hostent *ptr;
        int n=0;
        bool receive = false;

        mainStruct dataToSendToMain;
        logStruct dataTOSendTOLogger;
        socketStruct dataReceivedFromSensors;

        dataToSendToMain.status=init_success;
        strcpy(dataToSendToMain.source,"socket");
        dataToSendToMain.messageType = update;

        dataTOSendTOLogger.source="socket";
        dataTOSendTOLogger.logLevel=alert;
        dataTOSendTOLogger.status=fail;

        mqd_t socketToMain = mqueue_init(MAINQUEUENAME,MAIN_QUEUE_SIZE,sizeof(mainStruct));
        mqd_t socketToLogger = mqueue_init(LOGQUEUENAME,LOG_QUEUE_SIZE,sizeof(logStruct));
        struct mq_attr queue_attr1;
        queue_attr1.mq_maxmsg  = SOCKET_QUEUE_SIZE;
    queue_attr1.mq_msgsize = sizeof(socketStruct);
        mqd_t sensorToSocket = mq_open(SOCKETQUEUENAME, O_CREAT | O_RDWR, 0666, &queue_attr1);

        if(socketToMain < 0)
    {
        perror("socketTomain queue creation failed");
                dataToSendToMain.status=init_failure;
                dataTOSendTOLogger.message="socketTomain queue creation failed";
                mq_send(socketToLogger,(char*)&dataTOSendTOLogger,sizeof(logStruct),0);
    }
    if(socketToLogger<0)
        {
                perror("socketToLogger queue creation failed");
                dataToSendToMain.status=init_failure;
                dataTOSendTOLogger.message="socketToLogger queue creation failed";
                mq_send(socketToLogger,(char*)&dataTOSendTOLogger,sizeof(logStruct),0);
        }

         if(sensorToSocket<0)
        {
                perror("sensorToSocket queue creation failed");
                dataToSendToMain.status=init_failure;
                dataTOSendTOLogger.message="sensorToSocket queue creation failed";
                mq_send(socketToLogger,(char*)&dataTOSendTOLogger,sizeof(logStruct),0);

        }

        /*Initialising Timer*/
    struct sigevent socketEvent;
        struct sigaction socketAction;
```

```
        struct itimerspec socketTimerSpec;
        timer_t socketTimer;

        socketAction.sa_flags = SA_SIGINFO;
        socketAction.sa_sigaction = signal_handler;

        if(sigaction(SIGRTMIN + 7 , &socketAction, NULL)<0)
        {
                perror("Socket, timer handler initialisation failed");
                dataToSendToMain.status=init_failure;
                dataTOSendTOLogger.message="Socket, timer handler initialisation failed";
                mq_send(socketToLogger,(char*)&dataTOSendTOLogger,sizeof(logStruct),0);
        }

        socketEvent.sigev_notify = SIGEV_SIGNAL;
    socketEvent.sigev_signo = SIGRTMIN + 7;
    socketEvent.sigev_value.sival_ptr = &socketTimer;

        /* Creating timer */
        if((timer_create(CLOCK_REALTIME, &socketEvent, &socketTimer)) < 0)
        {
                perror("Timer creation failed for sensor task");
                dataToSendToMain.status=init_failure;
                dataTOSendTOLogger.message="Timer creation failed for sensor task";
                mq_send(socketToLogger,(char*)&dataTOSendTOLogger,sizeof(logStruct),0);
        }

        socketTimerSpec.it_interval.tv_nsec = 500000000; //To get the heart beat value after e
very 1 s
        socketTimerSpec.it_interval.tv_sec  = 0;//1;
        socketTimerSpec.it_value.tv_nsec    = 500000000;
        socketTimerSpec.it_value.tv_sec     = 0;//1;

        /* Starting timer */
        if(timer_settime(socketTimer,0,&socketTimerSpec,NULL)<0)
        {
                perror("Socket task timer settime failed");
                dataToSendToMain.status=init_failure;
                dataTOSendTOLogger.message="Socket task timer settime failed";
                mq_send(socketToLogger,(char*)&dataTOSendTOLogger,sizeof(logStruct),0);
                //exit(1);
        }

        /*Setting up server socket*/
        serverSocket=socket(AF_INET, SOCK_STREAM, 0);

        memset((char*)&serverAddr,0, sizeof(serverAddr));

        serverAddr.sin_family = AF_INET;
        serverAddr.sin_port = htons(10000);

        serverAddr.sin_addr.s_addr = htonl(INADDR_ANY);

        if(bind(serverSocket,(struct sockaddr*)&serverAddr,sizeof(serverAddr)) == -1)
        {
                printf("Bind Failure\n");
                dataToSendToMain.status=init_failure;
                dataTOSendTOLogger.message="Bind Failure";
                mq_send(socketToLogger,(char*)&dataTOSendTOLogger,sizeof(logStruct),0);
        }
        else
        {
                printf("Bind Success:<%u>\n", serverSocket);
```

```
                dataTOSendTOLogger.status=success;
                dataTOSendTOLogger.message="Bind Success";
                mq_send(socketToLogger,(char*)&dataTOSendTOLogger,sizeof(logStruct),0);
        }

        /*Sending init success/fail message to main*/
        mq_send(socketToMain,(char*)&dataToSendToMain,sizeof(mainStruct),0);

        listen(serverSocket,5);
        len=sizeof(struct sockaddr_in);

        printf("----------> Calling accept system call.\n");
        clientConnected=accept(serverSocket,(struct sockaddr*)&clientAddr,&len);
        if (clientConnected !=-1)
        {
                printf("Connection accepted:<%u>\n", clientConnected);
                dataTOSendTOLogger.status=success;
                dataTOSendTOLogger.message="Connection accepted";
                mq_send(socketToLogger,(char*)&dataTOSendTOLogger,sizeof(logStruct),0);
        }

        int input;
        const int errval=-500;

        while(1)
        {
                if(receive)
                {
                        receive = false;
                        int ret = mq_receive(sensorToSocket,(char*)&dataReceivedFromSensors,si
zeof(socketStruct),0);
                        if(ret > -1)
                        {
                                printf("Data received by server queue from %s\n",dataReceivedF
romSensors.source);

                                if(dataReceivedFromSensors.source=="temperature")
                                {
                                        printf("Temperature readings received from Temp task t
o socket server\n");
                                        if(send(clientConnected, (void*)&dataReceivedFromSenso
rs.value,sizeof(dataReceivedFromSensors.value),0)!=sizeof(dataReceivedFromSensors.value))
                                                {
                                                perror("Sending temp value to client failed\n"
);
                                                dataTOSendTOLogger.status=fail;
                                                dataTOSendTOLogger.message="Sending temp value
 to client failed";
                                                mq_send(socketToLogger,(char*)&dataTOSendTOLog
ger,sizeof(logStruct),0);
                                        }
                                }
                                else if(dataReceivedFromSensors.source=="light")
                                {
                                        printf("Light request received in server. Sending data
 to client\n");
                                        if(send(clientConnected, (void*)&dataReceivedFromSenso
rs.value,sizeof(dataReceivedFromSensors.value),0)!=sizeof(dataReceivedFromSensors.value))
                                        {
                                                perror("Sending light value to client failed\n
");
                                                dataTOSendTOLogger.status=fail;
                                                dataTOSendTOLogger.message="Sending light valu
```

```
e to client failed";
                                                mq_send(socketToLogger,(char*)&dataTOSendTOLog
ger,sizeof(logStruct),0);
                                        }
                                }
                                else
                                {
                                        strcpy(dataTOSendTOLogger.message,dataReceivedFromSens
ors.message);
                                        strcpy(dataTOSendTOLogger.source,"socket");
                                        dataTOSendTOLogger.status=fail;
                                        dataTOSendTOLogger.logLevel=alert;
                                        mq_send(socketToLogger,(char*)&dataTOSendTOLogger,size
of(logStruct),0);
                                        if(send(clientConnected, (void*)&errval,sizeof(errval)
,0)!=sizeof(errval))
                                        {
                                                perror("Sending error value to client failed\n
");
                                                dataTOSendTOLogger.status=fail;
                                                dataTOSendTOLogger.message="Sending error valu
e to client failed";
                                                mq_send(socketToLogger,(char*)&dataTOSendTOLog
ger,sizeof(logStruct),0);
                                        }
                                }
                        }
                }
                int rb=read(clientConnected,&input, sizeof(input));
                if(rb==sizeof(input))
                {
                        receive = true;
                        printf("Message received from client is %d\n",input);
                        strcpy(dataToSendToMain.source,"socket");
                        dataTOSendTOLogger.status=success;
                        dataTOSendTOLogger.logLevel=alert;
                        dataTOSendTOLogger.message="Request message received from client";
                        mq_send(socketToLogger,(char*)&dataTOSendTOLogger,sizeof(logStruct),0)
;

                        dataToSendToMain.messageType= request;
                        switch(input)
                        {
                                case 1:
                                strcpy(dataToSendToMain.unit,"Celsius");
                                break;

                                case 2:
                                strcpy(dataToSendToMain.unit,"Kelvin");
                                break;

                                case 3:
                                strcpy(dataToSendToMain.unit,"Fahrenheit");
                                break;

                                case 4:
                                strcpy(dataToSendToMain.unit,"Light");
                                break;

                        }
                        mq_send(socketToMain,(char *) &dataToSendToMain,sizeof(mainStruct),0);
                }
                if(socketHeartBeat)
```

```
                {
                        socketHeartBeat=false;
                        strcpy(dataToSendToMain.source,"socket");
                        dataToSendToMain.messageType = update;
                        dataToSendToMain.status = success;
                        mq_send(socketToMain,(char *) &dataToSendToMain,sizeof(mainStruct),0);
                }
                if(exitThread)
                {
                        break;
                }
        }
        printf("_____________________Socket task exiting_____________________\n");

        mq_close(socketToMain);
        mq_close(socketToLogger);
        mq_close(sensorToSocket);
        mq_unlink(MAINQUEUENAME);
        mq_unlink(LOGQUEUENAME);
        mq_unlink(SOCKETQUEUENAME);
        close(serverSocket);

        dataTOSendTOLogger.logLevel=alert;
        dataTOSendTOLogger.message="Server Socket Closed !!";
        mq_send(socketToLogger,(char*)&dataTOSendTOLogger,sizeof(logStruct),0);

        printf("\nServer Socket Closed !!\n");
        pthread_exit(NULL);
}


/*****************************************************************************
* Function name:- loggerRoutine                                              *
* Description:- This function is used by a logger thread.                    *
* @param:- void* (data from main)                                            *
* @return:- static void* (pthread exit value)                                *
*****************************************************************************/
static void* loggerRoutine(void *dataObj)
{
    mainStruct dataToSend;
        logStruct dataToReceive;

        mainInfoToOthers *obj1;
        obj1=malloc(sizeof(mainInfoToOthers));
        obj1=(mainInfoToOthers *)dataObj;

        /*Initialising Timer*/
    struct sigevent loggerEvent;
        struct sigaction loggerAction;
        struct itimerspec loggerTimerSpec;
        timer_t loggerTimer;

        loggerAction.sa_flags = SA_SIGINFO;
        loggerAction.sa_sigaction = signal_handler;

        if(sigaction(SIGRTMIN + 6 , &loggerAction, NULL)<0)
        {
                perror("Logger, timer handler initialisation failed");
        }

        loggerEvent.sigev_notify = SIGEV_SIGNAL;
    loggerEvent.sigev_signo = SIGRTMIN + 6;
    loggerEvent.sigev_value.sival_ptr = &loggerTimer;
```

```
        if((timer_create(CLOCK_REALTIME, &loggerEvent, &loggerTimer)) < 0)
        {
                perror("Timer creation failed for logger task");
                exit(1);
        }

        loggerTimerSpec.it_interval.tv_nsec = 0;
        loggerTimerSpec.it_interval.tv_sec  = 2;
        loggerTimerSpec.it_value.tv_nsec    = 0;
        loggerTimerSpec.it_value.tv_sec     = 2;

        if(timer_settime(loggerTimer,0,&loggerTimerSpec,NULL)<0)
        {
                perror("logger timer settime failed");
                exit(1);
        }


    strcpy(dataToSend.source,"logger");
        dataToSend.messageType = update;
        dataToSend.status = init_success;
    mqd_t loggerToMain = mqueue_init(MAINQUEUENAME,MAIN_QUEUE_SIZE,sizeof(mainStruct));
    if(loggerToMain < 0)
    {
        perror("logger queue creation failed");
    }

        mqd_t loggerQueue = mqueue_init(LOGQUEUENAME,LOG_QUEUE_SIZE,sizeof(logStruct));
        if(loggerQueue < 0)
    {
        perror("logger queue creation failed");
    }

        mq_send(loggerToMain,(char*)&dataToSend,sizeof(mainStruct),0);
        int y=0;
        while(1)
        {
                if(loggerHeartBeat)
                {
                        loggerHeartBeat=false;
                        dataToSend.messageType = update;
                        dataToSend.status = success;
                        mq_send(loggerToMain,(char*)&dataToSend,sizeof(mainStruct),0);
                        y++;
                }
                else
                {
                        int ret= mq_receive(loggerQueue, (char*)&dataToReceive, sizeof(logStru
ct),0);
                        if(ret!=-1)
                        {
                                logToFile(obj1->logFileName, dataToReceive);
                        }
                }
                if(exitThread)
                {
                        break;
                }
        }
        printf("_____________________Logger task exiting_____________________\n");
        mq_close(loggerToMain);
        mq_close(loggerQueue);
```

```
        mq_unlink(MAINQUEUENAME);
        mq_unlink(LOGQUEUENAME);
    timer_delete(loggerTimer);
        pthread_exit(NULL);
}

uint8_t isAlive = 0x0F;
int count;
int main(int argc, char *argv[])
{
        printf("Inside main task\n");
        count=1;
    mainStruct dataToReceive;
        tempStruct requestForTemp;
        socketStruct responseToSocket;
        lightStruct requestForLight;
        logStruct dataToLog;
        mainInfoToOthers dataObj;

        if(argc != 2)
        {
                printf("Try execution in below syntax\n");
                printf("Execute in format -> ./maintask <log file name>\n");
                return 0;
        }

        /* I2C initialisations */
        fd=myi2cInit(slaveAddFloat);
        fd1=temp_i2c_init(DEFAULT_SLAVE_ADDRESS);
        printf("fd in maintask is %d\n",fd);

        /* Initialize logger queue */
        mqd_t logQueue = mqueue_init(LOGQUEUENAME, LOG_QUEUE_SIZE, sizeof(logStruct));
    if(logQueue < 0)
    {
        perror("Main queue creation failed");
    }

        /*Built in self test calls*/
        int lightBIST=lightSensorBIST(fd);
        if(lightBIST==-1)
        {
                dataToLog.source = "main";
                dataToLog.message = "Light sensor BIST failed";
                mq_send(logQueue, (char*)&dataToLog, sizeof(dataToLog),0);
                mq_close(logQueue);
                mq_unlink(LOGQUEUENAME);
                printf("Light sensor BIST failed\n");
                return 0;
        }
        else
        {
                dataToLog.source = "main";
                dataToLog.message = "Light sensor BIST passed";
                mq_send(logQueue, (char*)&dataToLog, sizeof(dataToLog),0);
                printf("Light sensor BIST passed\n");
        }

        int tempBIST=tempSensorBIST(fd1);
        if(tempBIST==-1)
        {
                dataToLog.source = "main";
                dataToLog.message = "Temp sensor BIST failed";
```

```
                mq_send(logQueue, (char*)&dataToLog, sizeof(dataToLog),0);
                mq_close(logQueue);
                mq_unlink(LOGQUEUENAME);
                printf("Temp sensor BIST failed\n");
                return 0;
        }
        else
        {
                dataToLog.source = "main";
                dataToLog.message = "Temp sensor BIST passed";
                mq_send(logQueue, (char*)&dataToLog, sizeof(dataToLog),0);
                printf("Temp sensor BIST passed\n");
        }

        dataObj.logFileName=malloc(20);
        strcpy(dataObj.logFileName,argv[1]);
        printf("Received file name is %s\n",dataObj.logFileName);

        dataToLog.source = "main";

        //Creating queues for Inter Process Communication
    mqd_t mainQueue = mqueue_init(MAINQUEUENAME, MAIN_QUEUE_SIZE, sizeof(mainStruct));
    if(mainQueue < 0)
    {
                dataToLog.message = "Main queue creation failed";
                mq_send(logQueue, (char*)&dataToLog, sizeof(dataToLog),0);
        perror("Main queue creation failed");
    }

    mqd_t tempQueue = mqueue_init(TEMPQUEUENAME, TEMP_QUEUE_SIZE, sizeof(tempStruct));
    if(tempQueue < 0)
    {
                dataToLog.message = "Temp queue creation failed";
                mq_send(logQueue, (char*)&dataToLog, sizeof(dataToLog),0);
        perror("Temp queue creation failed");
    }

    mqd_t lightQueue = mqueue_init(LIGHTQUEUENAME, LIGHT_QUEUE_SIZE, sizeof(lightStruct));
    if(lightQueue < 0)
    {
                dataToLog.message = "Light queue init failed";
                mq_send(logQueue, (char*)&dataToLog, sizeof(dataToLog),0);
        perror("Light queue init failed");
    }

        mqd_t socketQueue = mqueue_init(SOCKETQUEUENAME, SOCKET_QUEUE_SIZE, sizeof(socketStruc
t));
        if(socketQueue < 0)
        {
                dataToLog.message = "Socket queue creation failed";
                mq_send(logQueue, (char*)&dataToLog, sizeof(dataToLog),0);
                perror("Socket queue creation failed");
    }

    printf("Main thread created with PID: %d and TID: %d\n",getpid(),(pid_t)syscall(SYS_gettid
));

    /*Creating Temperature Sensor Thread */
    if(pthread_create(&tempSensorTask,NULL,&tempSensorRoutine,(void *)&dataObj)!=0)
    {
                dataToLog.message = "tempSensorTask create failed";
                mq_send(logQueue, (char*)&dataToLog, sizeof(dataToLog),0);
        perror("tempSensorTask create failed");
```

```
    }

    /*Creating Light Sensor Thread */
    if(pthread_create(&lightSensorTask,NULL,&lightSensorRoutine,(void *)&dataObj)!=0)
    {
                dataToLog.message = "lightSensorTask create failed";
                mq_send(logQueue, (char*)&dataToLog, sizeof(dataToLog),0);
        perror("lightSensorTask create failed");
    }

    /*Creating Logger Thread */
    if(pthread_create(&loggerTask,NULL,&loggerRoutine,(void *)&dataObj)!=0)
    {
                dataToLog.message = "loggerTask create failed";
                mq_send(logQueue, (char*)&dataToLog, sizeof(dataToLog),0);
        perror("loggerTask create failed");
    }

    /*Creating Socket Thread */
    if(pthread_create(&socketTask,NULL,&socketRoutine,(void *)&dataObj)!=0)
    {
                dataToLog.message = "socketTask create failed";
                mq_send(logQueue, (char*)&dataToLog, sizeof(dataToLog),0);
        perror("socketTask create failed");
    }

        struct sigevent sigevTemp, sigevLight, sigevLog, sigevSocket;
        struct itimerspec timerConfigTemp, timerConfigLight, timerConfigLog, timerConfigSocket
;
        timer_t timerTemp, timerLight, timerSocket, timerLog;
        struct sigaction sigact;

        //Assigning signal handlers for timeout signals
        sigact.sa_flags = SA_SIGINFO;
    sigact.sa_sigaction = signal_handler;
        if((sigaction(SIGRTMIN, &sigact, NULL))<0)
        {
                perror("Failed setting signal handler");
        }

        if((sigaction(SIGRTMIN + 1, &sigact, NULL))<0)
        {
                perror("Failed setting signal handler");
        }

        if((sigaction(SIGRTMIN + 2, &sigact, NULL))<0)
        {
                perror("Failed setting signal handler");
        }

        if((sigaction(SIGRTMIN + 3, &sigact, NULL))<0)
        {
                perror("Failed setting signal handler");
        }

        if((sigaction(SIGINT, &sigact, NULL))<0)
        {
                perror("Failed setting signal handler");
        }

        //Creating 4 timer for monoring 4 created tasks.
        sigevTemp.sigev_notify            = SIGEV_SIGNAL;
        sigevTemp.sigev_signo             = SIGRTMIN;
```

```
        sigevTemp.sigev_value.sival_ptr   = &timerTemp;

        sigevLight.sigev_notify           = SIGEV_SIGNAL;
        sigevLight.sigev_signo            = SIGRTMIN + 1;
        sigevLight.sigev_value.sival_ptr  = &timerLight;

        sigevLog.sigev_notify             = SIGEV_SIGNAL;
        sigevLog.sigev_signo              = SIGRTMIN + 2;
        sigevLog.sigev_value.sival_ptr    = &timerLog;

        sigevSocket.sigev_notify          = SIGEV_SIGNAL;
        sigevSocket.sigev_signo           = SIGRTMIN + 3;
        sigevSocket.sigev_value.sival_ptr = &timerSocket;

        if((timer_create(CLOCK_REALTIME, &sigevTemp, &timerTemp)) < 0)
        {
                dataToLog.message = "Temp Timer setup failed";
                mq_send(logQueue, (char*)&dataToLog, sizeof(dataToLog),0);
                perror("Temp Timer setup failed");
                exit(1);
        }
        if((timer_create(CLOCK_REALTIME, &sigevLight, &timerLight)) < 0)
        {
                dataToLog.message = "Light Timer setup failed";
                mq_send(logQueue, (char*)&dataToLog, sizeof(dataToLog),0);
                perror("Light Timer setup failed");
                exit(1);
        }
        if((timer_create(CLOCK_REALTIME, &sigevLog, &timerLog)) < 0)
        {
                dataToLog.message = "Log Timer setup failed";
                mq_send(logQueue, (char*)&dataToLog, sizeof(dataToLog),0);
                perror("Log Timer setup failed");
                exit(1);
        }
        if((timer_create(CLOCK_REALTIME, &sigevSocket, &timerSocket)) < 0)
        {
                dataToLog.message = "Socket Timer setup failed";
                mq_send(logQueue, (char*)&dataToLog, sizeof(dataToLog),0);
                perror("Socket Timer setup failed");
                exit(1);
        }

        //Setiing supervisory timeout for each tasks.
        timerConfigTemp.it_interval.tv_nsec = 0;
        timerConfigTemp.it_interval.tv_sec  = 0;
        timerConfigTemp.it_value.tv_nsec    = 0;
        timerConfigTemp.it_value.tv_sec     = 1;

        timerConfigLight.it_interval.tv_nsec = 0;
        timerConfigLight.it_interval.tv_sec  = 0;
        timerConfigLight.it_value.tv_nsec    = 0;
        timerConfigLight.it_value.tv_sec     = 1;

        timerConfigLog.it_interval.tv_nsec = 0;
        timerConfigLog.it_interval.tv_sec  = 0;
        timerConfigLog.it_value.tv_nsec    = 0;
        timerConfigLog.it_value.tv_sec     = 10;

        timerConfigSocket.it_interval.tv_nsec = 0;
        timerConfigSocket.it_interval.tv_sec  = 0;
        timerConfigSocket.it_value.tv_nsec    = 0;
        timerConfigSocket.it_value.tv_sec     = 15;
```

```
        if((timer_settime(timerTemp, 0, &timerConfigTemp, NULL)) < 0)
        {
                dataToLog.message = "Temp Timer set failed";
                mq_send(logQueue, (char*)&dataToLog, sizeof(dataToLog),0);
                perror("Temp Timer set failed");
                exit(1);
        }
        if((timer_settime(timerLight, 0, &timerConfigLight, NULL)) < 0)
        {
                dataToLog.message = "Light Timer set failed";
                mq_send(logQueue, (char*)&dataToLog, sizeof(dataToLog),0);
                perror("Light Timer set failed");
                exit(1);
        }
        if((timer_settime(timerLog, 0, &timerConfigLog, NULL)) < 0)
        {
                dataToLog.message = "Log Timer set failed";
                mq_send(logQueue, (char*)&dataToLog, sizeof(dataToLog),0);
                perror("Log Timer set failed");
                exit(1);
        }
        if((timer_settime(timerSocket, 0, &timerConfigSocket, NULL)) < 0)
        {
                dataToLog.message = "Socket Timer set failed";
                mq_send(logQueue, (char*)&dataToLog, sizeof(dataToLog),0);
                perror("Socket Timer set failed");
                exit(1);
        }

        // Check if the initialization of all the tasks is done.
        int task_up = 0, retries = 0;
        while(task_up != 4)
        {
                int ret=mq_receive(mainQueue, (char*)&dataToReceive, sizeof(mainStruct),0);
                if (ret > -1 && dataToReceive.status == init_success)
                {
                        printf("---%s Task is up and running---\n",dataToReceive.source);
                        task_up++;
                }
                else if(ret > -1 && dataToReceive.status == init_failure)
                {
                        retries++;
                }
                if(retries > 10)
                {
                        dataToLog.message = "All the tasks are not up.";
                        mq_send(logQueue, (char*)&dataToLog, sizeof(dataToLog),0);
                        printf("All tasks are not up");
                        return 0;
                }
        }


    while(1)
    {
        int ret=mq_receive(mainQueue, (char*)&dataToReceive, sizeof(mainStruct),0);
        if(ret>-1)
        {
                        //Heart beat message received from temperature task
                        if((strcmp(dataToReceive.source,"temperature"))==0 && dataToReceive.me
ssageType == update)
                        {
```

```
                switch(dataToReceive.status)
                {
                case init_success:
                        isAlive = isAlive | TEMPERATURE_TASK;
                        break;
                case success:
                        if((timer_settime(timerTemp, 0, &timerConfigTemp, NULL
)) < 0)
                        {
                                perror("Temp Timer set failed");
                        }
                        break;
                case fail:
                case init_failure:
                        isAlive = isAlive & ~TEMPERATURE_TASK;
                        break;
                }
        }

        //Heart beat message received from light task
        else if((strcmp(dataToReceive.source,"light"))==0 && dataToReceive.mes
sageType == update)
        {
                switch(dataToReceive.status)
                {
                case init_success:
                        isAlive = isAlive | LIGHT_TASK;
                        break;
                case success:
                        if((timer_settime(timerLight, 0, &timerConfigLight, NU
LL)) < 0)
                        {
                                perror("Light Timer set failed");
                        }
                        break;
                case fail:
                case init_failure:
                        isAlive = isAlive & ~LIGHT_TASK;
                        break;
                }
        }

        //Heart beat message received from logger task
        else if((strcmp(dataToReceive.source,"logger"))==0 && dataToReceive.me
ssageType == update)
        {
                switch(dataToReceive.status)
                {
                case init_success:
                        isAlive = isAlive | LOGGER_TASK;
                        break;
                case success:
                        if((timer_settime(timerLog, 0, &timerConfigLog, NULL))
 < 0)
                        {
                                perror("Logger Timer set failed");
                        }
                        break;
                case fail:
                case init_failure:
                        isAlive = isAlive & ~LOGGER_TASK;
                        break;
                }
```

```
                        }

                        //Heart beat message received from socket task
                        else if((strcmp(dataToReceive.source,"socket"))==0 && dataToReceive.me
ssageType == update)
                        {
                                switch(dataToReceive.status)
                                {
                                case init_success:
                                        isAlive = isAlive | SOCKET_TASK;
                                        break;
                                case success:
                                        if((timer_settime(timerSocket, 0, &timerConfigSocket,
NULL)) < 0)
                                        {
                                                perror("Socket Timer set failed");
                                        }
                                        break;
                                case fail:
                                case init_failure:
                                        isAlive = isAlive & ~SOCKET_TASK;
                                        break;
                                }
                        }

                        //Request message is recevied from socket server for getting temperatu
re or light values
                        else if((strcmp(dataToReceive.source,"socket"))==0 && dataToReceive.me
ssageType == request)
                        {
                                if(strcmp(dataToReceive.unit,"Celsius")==0 || strcmp(dataToRec
eive.unit,"Fahrenheit")==0 || strcmp(dataToReceive.unit,"Kelvin")==0)
                                {
                                        if(isAlive & TEMPERATURE_TASK)
                                        {
                                                printf("\nTemp task is alive\n");
                                                printf("Sending request to temp sensor task to
 send data to socket task\n");
                                                strcpy(requestForTemp.source, "main");
                                                strcpy(requestForTemp.unit, dataToReceive.unit
);
                                                int noOfBytesSent = mq_send(tempQueue, (char*)
&requestForTemp, sizeof(requestForTemp), 0);
                                                if(noOfBytesSent < 0)
                                                {
                                                        perror("Sending request failed");
                                                }
                                        }
                                        else
                                        {
                                                printf("Temp sensor task is not active\n");
                                                strcpy(responseToSocket.source, "main");
                                                strcpy(responseToSocket.message, "Temp Sensor
Task not active");
                                                int noOfBytesSent = mq_send(socketQueue, (char
*)&responseToSocket, sizeof(responseToSocket), 0);
                                                if(noOfBytesSent < 0)
                                                {
                                                        perror("Sending response failed");
                                                }
                                        }
                                }
                                else
```

```
                                {
                                        if(isAlive & LIGHT_TASK)
                                        {
                                                printf("\nLight task is alive\n");
                                                printf("Sending request to light sensor task t
o send data to socket task\n");
                                                strcpy(requestForLight.source, "main");
                                                int noOfBytesSent = mq_send(lightQueue, (char*
)&requestForLight, sizeof(requestForLight), 0);
                                                if(noOfBytesSent < 0)
                                                {
                                                        perror("Sending request failed");
                                                }
                                        }
                                        else
                                        {
                                                printf("Light sensor task is not active\n");
                                                strcpy(responseToSocket.source, "main");
                                                strcpy(responseToSocket.message,"Light sensor
task not active");
                                                int noOfBytesSent = mq_send(socketQueue, (char
*)&responseToSocket, sizeof(responseToSocket), 0);
                                                if(noOfBytesSent < 0)
                                                {
                                                        perror("Sending response failed");
                                                }
                                        }
                                }
                        }
                }
                if(!isAlive)
                {
                        break;
                }

    }
        printf("_____________________Main task exiting_____________________\n");
    mq_close(mainQueue);
        mq_close(tempQueue);
        mq_close(lightQueue);
        mq_close(socketQueue);

        mq_unlink(MAINQUEUENAME);
        mq_unlink(TEMPQUEUENAME);
        mq_unlink(LIGHTQUEUENAME);
        mq_unlink(SOCKETQUEUENAME);

    pthread_join(tempSensorTask,NULL);
    pthread_join(lightSensorTask,NULL);
    pthread_join(loggerTask,NULL);
        pthread_cancel(socketTask);
    pthread_join(socketTask,NULL);
    return 0;
}


static void signal_handler(int sig, siginfo_t *si, void *uc)
{
        switch(sig)
        {
        case 2:
                printf("SIGINT signal is received\n ----------> Exiting thread <---------\n");
                exitThread = true;
```

```
                break;
        case 34:
                // Siganl indicating temperature sensor timeout i.e. Temperature task failed
                printf("Temperature task timeout\n");
                isAlive &= ~TEMPERATURE_TASK;
                break;
        case 35:
                // Signal indicating light sensor timeout i.e. Light task failed
                printf("Light Task Timeout\n");
                isAlive &= ~LIGHT_TASK;
                break;
        case 36:
                // Signal indicating logger timeout i.e. Logger task failed
                printf("Logger Task Timeout\n");
                isAlive &= ~LOGGER_TASK;
                break;
        case 37:
                // Signal indicating socket sensor timeout i.e. Socket task failed
                isAlive &= ~SOCKET_TASK;
                break;
        case 38:
                // Signal instructing to take temperature sensor reading and send heartbeat
                measureTemperature = true;
                break;
        case 39:
                // Signal instructing to take light sensor reading and send heartbeat
                measureLight = true;
                break;
        case 40:
                // Signal instructing logger task to send heartbeat to main indicating it is a
live
                loggerHeartBeat = true;
                break;
        case
                socketHeartBeat = true;
                break;
        }
}


/*****************************************************************************************
***
* File name   : maintask.h
  *
* Authors     : Puneet Bansal and Nachiket Kelkar
  *
* Description : The defines used by maintask.c
  *
* Tools used  : GNU make, gcc, arm-linux-gcc
  *
*****************************************************************************************
**/
//#include "mq.h"
#include "logger.h"

#define TEMPERATURE_TASK 0x01
#define LIGHT_TASK       0x02
#define LOGGER_TASK      0x04
#define SOCKET_TASK      0x08


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

```
* File name   : mq.c
  *
* Authors     : Nachiket Kelkar and Puneet Bansal
  *
* Description : The function definition used for mqueue.
  *
* Tools used  : GNU make, gcc, arm-linux-gcc
  *
*******************************************************************************
**/
#include "mq.h"
#include <stdio.h>


mqd_t mqueue_init(const char* queue_name, int queue_size, int message_size)
{
    mqd_t msg_q_des;
    struct mq_attr queue_attr;

    queue_attr.mq_maxmsg  = queue_size;
    queue_attr.mq_msgsize = message_size;
    queue_attr.mq_flags   = O_NONBLOCK;
    msg_q_des = mq_open(queue_name, O_CREAT | O_RDWR | O_NONBLOCK, 0666, &queue_attr);

    return msg_q_des;
}


/*******************************************************************************
***
* File name   : mq.h
  *
* Authors     : Nachiket Kelkar and Puneet Bansal
  *
* Description : The function declaration used for mqueue, enums and structures.
  *
* Tools used  : GNU make, gcc, arm-linux-gcc
  *
*******************************************************************************
**/
#include <mqueue.h>
#include <stdbool.h>

/* Message queues for all the tasks */
#define MAINQUEUENAME   "/mainqueue"
#define TEMPQUEUENAME   "/tempqueue"
#define LIGHTQUEUENAME  "/lightqueue"
#define SOCKETQUEUENAME "/socketqueue"
#define LOGQUEUENAME    "/logqueue"

/* Message queue size for all the tasks */
#define MAIN_QUEUE_SIZE   10
#define TEMP_QUEUE_SIZE   5
#define LIGHT_QUEUE_SIZE  5
#define SOCKET_QUEUE_SIZE 5
#define LOG_QUEUE_SIZE    10

typedef enum {
        request,
        update
}messageTypeEnum;

typedef enum{
```

```
        fail,
        success,
        init_success,
        init_failure,
}statusEnum;

typedef enum{
        debug,
        alert,
        info
}logLevelEnum;

/* SStructure to communicate to the maintask*/
typedef struct
{
    char source[20];
        messageTypeEnum messageType;
        statusEnum status;
        char unit[20];
}mainStruct;

/* Structure to communicate to the temperature task*/
typedef struct{
        char source[20];
        char unit[20];
}tempStruct;


/* Structure to communicate to the light task*/
typedef struct{
        char source[20];
}lightStruct;

/* Structure to communicate to the logger task*/
typedef struct{
        logLevelEnum logLevel;
        char* source;
        statusEnum status;
        char* message;
        float value;
        char unit[20];
}logStruct;

/* Structure to communicate to the socket task*/
typedef struct{
        char* source;
        float value;
        char* unit;
        char message[30];
}socketStruct;

/*user defined functions*/

/**
 * @name: mqueue_init
 *
 * @param1: message queue name
 * @param2: max message queue size
 * @param3: size of the data to send
 *
 * @description: wrapper around mq_open function. Sets the attributes of the queue and opens t
he queue with the specified parameters.
 *
```

```
 * return: message queue file descriptor.
 * */
mqd_t mqueue_init(const char*, int, int);


/*****************************************************************************************
***
* File name   : myi2c.c
  *
* Authors     : Puneet Bansal and Nachiket Kelkar
  *
* Description : The function definition for reading and writing to i2c device
  *
* Tools used  : GNU make, gcc, arm-linux-gcc
  *
*****************************************************************************************
**/
// ref:https://elinux.org/Interfacing_with_I2C_Devices

#include<linux/i2c.h>
#include<linux/i2c-dev.h>
#include <errno.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include "myi2c.h"

int myi2cInit(uint8_t slaveAdd)
{
char fileName[50];
int fileDescrip;
sprintf(fileName,"/dev/i2c-2");
fileDescrip=open(fileName,O_RDWR);

if(fileDescrip < 0)
{
  perror("Failed to open i2c file");
  exit(1);
}

if(ioctl(fileDescrip, I2C_SLAVE, slaveAdd) < 0)
{
  perror("Failed to communicate with slave"); //or use errno
  exit(1);
}
return fileDescrip;
}

uint8_t* myi2cRead(int fileDescrip,uint8_t len)
{
static uint8_t readBuffer[2];

pthread_mutex_lock(&lock);
if (read(fileDescrip,readBuffer,len) != len)
{
  perror("Read failed");
  pthread_mutex_unlock(&lock);
  return NULL;
```

```
  //check whether we need to return or not.
}
pthread_mutex_unlock(&lock);
//printf("Value of buffer is %x-%x\n",readBuffer[0],readBuffer[1]);
return readBuffer;
}

int myi2cWrite(int fileDescrip, uint8_t writeBuffer[], uint8_t len)
{
pthread_mutex_lock(&lock);
if (write(fileDescrip,writeBuffer,len) != len)
{
pthread_mutex_unlock(&lock);
perror("Write failed");
return -1;
}
pthread_mutex_unlock(&lock);
return 0;

}

/*****************************************************************************************
***
* File name   : myi2c.h
  *
* Authors     : Puneet Bansal and Nachiket Kelkar
  *
* Description : The function declaration for reading and writing to i2c device
  *
* Tools used  : GNU make, gcc, arm-linux-gcc
  *
*****************************************************************************************
**/
#include <stdint.h>
#include <pthread.h>

#define slaveAddFloat 0x39
#define slaveAddGnd 0x29
#define slaveAddVdd 0x49

/*Light Sensor Register addressess*/
#define CMDBYTE 0x80
#define CMDWORD 0xA0
#define CNTRLREG 0x00
#define INTCTL 0x06
#define DATA0LOW 0x0c
#define DATA0HIGH 0x0d
#define DATA1LOW 0x0e
#define DATA1HIGH 0x0f

/**
 * @name: myi2cInit
 * @param1: uint8_t slaveAdd
 *
 * @description: Opens the /dev/i2c-2 file and designates the slave.
 *
 * @return: int- i2c file descriptor.
 * */
int myi2cInit(uint8_t slaveAdd);


/**
 * @name: myi2cRead
```

```
 * @param1: fileDescriptor
 * @param2: length of bytes to be read
 *
 * @description:
 * wrapper around the read system call to read the designated number of bytes from the file. W
hen the length read from file is equal
 * to the length specified by the user, it indicates success. NULL is returned on failure.
 *
 * @return: uint8_t* - buffer containing the data read from file
 * */
uint8_t* myi2cRead(int, uint8_t);


/**
 * @name:myi2cWrite
 * @aparam1 : filedescriptor
 * @param2 :  data to write
 * @param3 :  the length of bytes to be written.
 *
 * @description:
 * wrapper around the write system call to write the designated number of bytes to the file. W
hen the length written to file is equal
 * to the length specified by the user, it indicates success and returns 0 . 0 is returned on
failure.
 *
 * return: 0 for success and -1 for error.
 * */
int myi2cWrite(int,uint8_t[],uint8_t);

pthread_mutex_t lock;

/*****************************************************************************************
***
* File name   : temp_i2c.c
  *
* Authors     : Nachiket Kelkar and Puneet Bansal
  *
* Description : The functions used for reading and configuring temperature sensor TMP102 for
  *
*               getting the temperature values through i2c interface.
  *
* Tools used  : GNU make, gcc, arm-linux-gcc
  *
*****************************************************************************************
**/

/************ The standard C libraries included for functionality ************/
#include <stdio.h>
#include <linux/i2c-dev.h>
#include <stdbool.h>
#include <stdlib.h>
#include <pthread.h>
#include <errno.h>
#include <signal.h>
#include <sys/stat.h>
#include <mqueue.h>

/************* The user library containing required information *************/
#include "temp_i2c.h"
#include "myi2c.h"
//#include "logger.h"
```

```
int temp_i2c_init(uint8_t slave_addr)
{
        return myi2cInit(slave_addr);
}


int temp_i2c_write_to_reg(int file_des, uint8_t temp_sens_reg_to_write, int16_t data_to_write)
{
        uint8_t buffer[3];
        buffer[0] = temp_sens_reg_to_write;
        uint16_t config_reg, config_value;

        switch(temp_sens_reg_to_write)
        {
        // If there is write to temperature read only register
        case TEMP_REG_ADDR:
                printf("%s::Not allowed to write to read only register\n",__func__);
        break;

        // If there is write to config register
        case CONFIG_REG_ADDR:
                config_value = temp_i2c_read_from_reg(file_des, CONFIG_REG_ADDR);
                config_reg = data_to_write;
                buffer[1] = config_reg >> 8;
                buffer[2] = config_reg;
        break;

        // If there is write operation to Tlow or Thigh register
        case TLOW_REG_ADDR:
        case THIGH_REG_ADDR:
                data_to_write = data_to_write/0.0625;
                buffer[1] = data_to_write >> 4;
                buffer[2] = data_to_write << 4;
        break;
        }
        // Write the buffer values to temp sensor register using i2c
        if(myi2cWrite(file_des, buffer, sizeof(buffer)) < 0)
        {
                printf("%s\n",__func__);
                perror("Write failed: ");
        return -1;
        }
        return 0;
}


uint16_t temp_i2c_read_from_reg(int file_des, uint8_t temp_sens_to_read_from)
{
        uint8_t* buffer;
        uint16_t reg_val;
        if(myi2cWrite(file_des, &temp_sens_to_read_from, sizeof(temp_sens_to_read_from)) < 0)
        {
                printf("%s\n",__func__);
                perror("Write failed: ");
                return 10000;
        }
        buffer = myi2cRead(file_des, 2);
    if(buffer == NULL)
    {
        printf("Temperature read failed\n");
        return 10000;
    }
        //printf("Value of buffer in %s is %x-%x\n",__func__,buffer[0],buffer[1]);
```

```
        return ((uint16_t)buffer[0] << 8 | buffer[1]);
}


float read_temperature(int file_des, uint8_t temp_sens_to_read_from)
{
        int16_t temperature;
        float final_temp;

        if(temp_sens_to_read_from == CONFIG_REG_ADDR)
        {
                printf("%s::Config register values are not temperature\n",__func__);
                exit(1);
        }
        temperature = temp_i2c_read_from_reg(file_des, temp_sens_to_read_from);
    if(temperature == 10000)
    {
            return 10000;
    }
        //printf("temperature = %x\n",temperature);
        temperature = temperature >> 4;
        //printf("temperature = %x\n",temperature);
        final_temp = temperature * 0.0625;
        //printf("final_temp = %d\n",final_temp);
        return final_temp;
}

float convert_to_unit(float value, int temperature_unit)
{
        float final_temp;
        switch(temperature_unit)
        {
        case Celsius:
                final_temp = value;
                break;
        case Fahrenheit:
                final_temp = value * 1.8;
                final_temp = final_temp + 32;
                break;
        case Kelvin:
                final_temp = value + 273.5;
                break;
        }
        return final_temp;
}


/*************************************************************************************
***
* File name   : temp_i2c.h
  *
* Authors     : Nachiket Kelkar and Puneet Bansal
  *
* Description : The functions used for reading and configuring temperature sensor TMP102 for
  *
*               getting the temperature values through i2c interface.
  *
* Tools used  : GNU make, gcc, arm-linux-gcc.
  *
*************************************************************************************
**/
#include <stdint.h>
```

```
/* Format of meassge that is sent to temperature thread */
// typedef struct{
//     int temperature_unit;
//     char* source;
// }temp_msg;

/* Enumeration for temperature units available */
typedef enum {
Celsius,
Fahrenheit,
Kelvin,
}temperature_unit;

#define DEFAULT_UNIT Celsius;

/* Default slave address */
#define DEFAULT_SLAVE_ADDRESS 0X48

/* The defines for addresses of the registers */
#define TEMP_REG_ADDR   0x00
#define CONFIG_REG_ADDR 0x01
#define TLOW_REG_ADDR   0x02
#define THIGH_REG_ADDR  0x03


/* The macros for configuration register */
#define DEFAULT_CONFIG        0x60A0
#define MODE_12BIT            (0 << 4)
#define MODE_13BIT            (1 << 4)
#define REFRESH_025HZ         (0 << 6)
#define REFRESH_1HZ           (1 << 6)
#define REFRESH_4HZ           (2 << 6)
#define REFRESH_8HZ           (3 << 6)
#define SHUTDOWN_MODE_ENABLE  (1 << 8)
#define SHUTDOWN_MODE_DISABLE (0 << 8)
#define INTERRUPT_MODE        (1 << 9)
#define COMPARATOR_MODE       (0 << 9)
#define ALERT_ACTIVE_HIGH     (1 << 10)
#define ALERT_ACTIVE_LOW      (0 << 10)
#define ALERT_ON_1_FAULT      (0 << 11)
#define ALERT_ON_2_FAULT      (1 << 11)
#define ALERT_ON_4_FAULT      (2 << 11)
#define ALERT_ON_6_FAULT      (3 << 11)
#define START_CONVERSION      (1 << 15)


/* Default values for configuring the registers */
#define TLOW_REG_DEFAULT 25
#define THIGH_REG_DEFAULT 35

/* Macros for queue setup */
//#define TEMP_SENS_QUEUE  "/temp_sens"
//#define TEMP_QUEUE_SIZE  10

/* Macros for timer setup */
#define CLOCK_TO_USE               CLOCK_REALTIME
#define SIGNAL_NOTIFICATION_METHOD SIGEV_SIGNAL
#define SIGNAL_NO                  SIGRTMIN
#define TIME_IN_NANOSEC            100000000  //100msec

/* The functions that are used to communicate to the i2c temperature sensor TMP102 */

/*
```

```
* Function name:- temp_i2c_init
* Description:- This function opens the i2c file for i2c transactions. It then sets
*               the slave address for the transactions according to the parameter.
* @param:- uint8_t (slave address)
* @return:- int (file descriptor)
*/
int temp_i2c_init(uint8_t);


/*
* Function name:- temp_i2c_write_to_reg
* Description:- This function takes the file descriptor as parameter which is used to
*               write to a file. It writes the data to the temperature sensor register
*               which is described in parameter. For writing the data to THIGH or TLOW
*               regiser the data to write should be in Celsius.
* @param:- int (file descriptor), uint8_t (temperature sensor register address),
*          int16_t (data to write)
* @return:- int (return error)
*/
int temp_i2c_write_to_reg(int, uint8_t, int16_t);


/*
* Function name:- temp_i2c_read_from_reg
* Description:- This function takes the file descriptor as parameter which is used to
*               write to a file. It reads the received i2c data from register passed
*               and return the buffer value in uint16_t format.
* @param:- int (file descriptor), uint8_t (temperature sensor register address),
* @return:- uint16_t (contents of the register)
*/
uint16_t temp_i2c_read_from_reg(int, uint8_t);


/*
* Function name:- read_temperature
* Description:- This function takes the file descriptor as parameter which is used to
*               write to a file. It formats the data of the register passed in the format
*               of Celcius unit. As configuration register does not contain
*               temperature passing config register address will cause an error.
* @param:- int (file descriptor), uint8_t (temperature sensor register address),
* @return:- float (temperature in Celsius)
*/
float read_temperature(int, uint8_t);


/*
* Function name:- convert_to_unit
* Description:- This function takes temperature and unit to conver to and then converts
*               the value in required temperature unit and returns the value.
* @param:- float (value to be converted), int (temperature unit to convert to),
* @return:- float (temperature in requested unit)
*/
float convert_to_unit(float, int);


/*****************************************************************************************
***
* File name   : client.c
  *
* Authors     : Nachiket Kelkar and Puneet Bansal
  *
* Description : The functions definition for client
  *
```

```
* Tools used  : GNU make, gcc, arm-linux-gcc.
  *
*******************************************************************************
**/
https://stackoverflow.com/questions/18021189/how-to-connect-two-computers-over-internet-using-
socket-programming-in-c


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

#define SERVER_IP_ADDRESS "10.0.0.59"
//#define SERVER_IP_ADDRESS "128.138.189.162"

int main(void)
{
    int clientSocket; /* Socket Decriptor for Client */
    struct sockaddr_in serverAddr;
    struct hostent *ptr;
    int input;

    clientSocket=socket(AF_INET, SOCK_STREAM, 0);
    if(clientSocket<0)
    {
        perror("Client socket creation failed");
    }

    memset((char*)&serverAddr, 0, sizeof(serverAddr));

    serverAddr.sin_family = AF_INET;
    serverAddr.sin_port = htons(10000);

    ptr=gethostbyname(SERVER_IP_ADDRESS);
    memcpy(&serverAddr.sin_addr,ptr->h_addr,ptr->h_length);

    if((connect(clientSocket, (struct sockaddr*)&serverAddr, sizeof(serverAddr)))==-1)
    { printf("\nServer Not Ready !!\n"); exit(1); }

float val;
while(1)
{
    printf("\nWelcome to the Client Socket. Enter the option you want to perform\n");
    printf("1.Get Temperature in Celsius\n2.Get Temperature in Kelvin\n3.Get Temperature in Fa
hrenheit\n4.Get Lux value\n->");
    scanf("%d",&input);
    send(clientSocket, (void*)&input, sizeof(input)+1,0);

    if(read(clientSocket, &val, sizeof(val))==sizeof(val))
    {
        if(val!=-500)
        {
                switch(input)
                {
                        case 1:
                        printf("Temperature value is %f Celsius\n",val);
                        break;
```

```
                        case 2:
                        printf("Temperature value is %f Kelvin\n",val);
                        break;

                        case 3:
                        printf("Temperature value is %f Fahrenheit\n",val);
                        break;

                        case 4:
                        printf("Light value is %f \n",val);
                        if(val>50)
                        printf("The vicinity of sensor has light \n");
                        else
                        printf("The vicinity of sensor is dark \n");
                        break;

                }
        }
        else
        {
        printf("Error reading from sensor\n");
        }
     }
}
return 0;
}
```