```
/*************************************************************************
*
* Author:       Pavan Dhareshwar & Sridhar Pavithrapu
* Date:         03/10/2018
* File:         external_app.h
* Description:  Header file containing the macros, structs/enums, globals
                and function prototypes for source file external_app.c
*************************************************************************
/

#ifndef _EXTERNAL_APP_H_
#define _EXTERNAL_APP_H_

/*---------------------------------- INCLUDES ----------------------------
----*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#include <sys/socket.h>

#include <netinet/in.h>
#include <arpa/inet.h>

/*---------------------------------- INCLUDES ----------------------------
----*/

/*---------------------------------- MACROS ----------------------------
----*/
#define SERVER_PORT_NUM                         8500
#define SERVER_LISTEN_QUEUE_SIZE          5

#define BUFF_SIZE                                 1024

#define SOCK_REQ_MSG_API_MSG_LEN            64

/*---------------------------------- MACROS ----------------------------
----*/

/*---------------------------------- GLOBALS ----------------------------
----*/

/*---------------------------------- GLOBALS ----------------------------
----*/

/*---------------------------- STRUCTURES/ENUMERATIONS -----------------
----*/
enum _req_recipient_
{
    REQ_RECP_TEMP_TASK,
    REQ_RECP_LIGHT_TASK
};

struct _socket_req_msg_struct_
{
    char req_api_msg[SOCK_REQ_MSG_API_MSG_LEN];
    enum _req_recipient_ req_recipient;
    void *ptr_param_list;
```

```
};

struct _int_thresh_reg_struct_
{
    uint8_t thresh_low_low;
    uint8_t thresh_low_high;
    uint8_t thresh_high_low;
    uint8_t thresh_high_high;
};

/*---------------------------- STRUCTURES/ENUMERATIONS ------------------
----*/

/*---------------------------- FUNCTION PROTOTYPES ----------------------
----*/

/*---------------------------- FUNCTION PROTOTYPES ----------------------
----*/

#endif
/**************************************************************************
***
* Author:       Pavan Dhareshwar & Sridhar Pavithrapu
* Date:         03/10/2018
* File:         external_app.c
* Description:  Source file containing the functionality and
implementation
*               of external application
**************************************************************************
**/

#include "external_app.h"

int main(void)
{
    int client_sock;
    struct sockaddr_in serv_addr;

    char buffer[BUFF_SIZE];
    memset(buffer, '\0', sizeof(buffer));

    if ((client_sock = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    {
        printf("\n Socket creation error \n");
        return -1;
    }

    memset(&serv_addr, '0', sizeof(serv_addr));

    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(SERVER_PORT_NUM);

    // Convert IPv4 and IPv6 addresses from text to binary form
    if(inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr)<=0)
    {
        printf("\nInvalid address/ Address not supported \n");
        return -1;
    }
```

```
    if (connect(client_sock, (struct sockaddr *)&serv_addr,
sizeof(serv_addr)) < 0)
    {
        printf("\nConnection Failed \n");
        return -1;
    }

    struct _socket_req_msg_struct_ ext_app_req_msg = {0};
    strcpy(ext_app_req_msg.req_api_msg, "get_lux_data");
    ext_app_req_msg.req_recipient = REQ_RECP_LIGHT_TASK;
    ext_app_req_msg.ptr_param_list = NULL;

    printf("Sending %s request to socket task\n",
ext_app_req_msg.req_api_msg);
    ssize_t num_sent_bytes = send(client_sock, &ext_app_req_msg,
            sizeof(struct _socket_req_msg_struct_), 0);
    if (num_sent_bytes < 0)
    {
        perror("send failed");
    }
    else
    {
        /* Receiving message from parent process */
        size_t num_read_bytes = read(client_sock, buffer,
sizeof(buffer));
        printf("Message received in external app : %s\n", buffer);
    }

    sleep(2);

    strcpy(ext_app_req_msg.req_api_msg,
"get_light_sensor_int_thresh_reg");
    ext_app_req_msg.req_recipient = REQ_RECP_LIGHT_TASK;
    ext_app_req_msg.ptr_param_list = NULL;

    printf("Sending %s request to socket task\n",
ext_app_req_msg.req_api_msg);
    num_sent_bytes = send(client_sock, &ext_app_req_msg,
            sizeof(struct _socket_req_msg_struct_), 0);
    if (num_sent_bytes < 0)
    {
        perror("send failed");
    }
    else
    {
        /* Receiving message from parent process */
        size_t num_read_bytes = read(client_sock, buffer,
sizeof(buffer));
        printf("[External App] : Int Thresh Reg Val:: lowlow: 0x%x,
lowhigh: 0x%x, highlow: 0x%x, highhigh: 0x%x\n",
                ((struct _int_thresh_reg_struct_ *)&buffer)-
>thresh_low_low, ((struct _int_thresh_reg_struct_ *)&buffer)-
>thresh_low_high,
                ((struct _int_thresh_reg_struct_ *)&buffer)-
>thresh_high_low, ((struct _int_thresh_reg_struct_ *)&buffer)-
>thresh_high_high);
    }

    sleep(2);
```

```
    struct _int_thresh_reg_struct_ int_thresh_reg_struct = {0};
    int_thresh_reg_struct.thresh_low_low = 5;
    int_thresh_reg_struct.thresh_low_high = 0;
    int_thresh_reg_struct.thresh_high_low = 20;
    int_thresh_reg_struct.thresh_high_high = 0;

    strcpy(ext_app_req_msg.req_api_msg,
"set_light_sensor_int_thresh_reg");
    ext_app_req_msg.req_recipient = REQ_RECP_LIGHT_TASK;
    ext_app_req_msg.ptr_param_list = &int_thresh_reg_struct;

    printf("Sending %s request to socket task\n",
ext_app_req_msg.req_api_msg);
    num_sent_bytes = send(client_sock, &ext_app_req_msg,
            sizeof(struct _socket_req_msg_struct_), 0);
    if (num_sent_bytes < 0)
    {
        perror("send failed");
    }
    else
    {
        /* Receiving message from parent process */
        size_t num_read_bytes = read(client_sock, buffer,
sizeof(buffer));
        printf("Message received in external app : %s\n", buffer);
    }

    sleep(2);

    strcpy(ext_app_req_msg.req_api_msg,
"get_light_sensor_int_thresh_reg");
    ext_app_req_msg.req_recipient = REQ_RECP_LIGHT_TASK;
    ext_app_req_msg.ptr_param_list = NULL;

    printf("Sending %s request to socket task\n",
ext_app_req_msg.req_api_msg);
    num_sent_bytes = send(client_sock, &ext_app_req_msg,
            sizeof(struct _socket_req_msg_struct_), 0);
    if (num_sent_bytes < 0)
    {
        perror("send failed");
    }
    else
    {
        /* Receiving message from parent process */
        size_t num_read_bytes = read(client_sock, buffer,
sizeof(buffer));
        printf("[External App] : Int Thresh Reg Val:: lowlow: 0x%x,
lowhigh: 0x%x, highlow: 0x%x, highhigh: 0x%x\n",
                ((struct _int_thresh_reg_struct_ *)&buffer)-
>thresh_low_low, ((struct _int_thresh_reg_struct_ *)&buffer)-
>thresh_low_high,
                ((struct _int_thresh_reg_struct_ *)&buffer)-
>thresh_high_low, ((struct _int_thresh_reg_struct_ *)&buffer)-
>thresh_high_high);
    }

     return 0;
}
```

```
/*****************************************************************************
*
* Author:       Pavan Dhareshwar & Sridhar Pavithrapu
* Date:         03/10/2018
* File:         external_app.h
* Description:  Header file containing the macros, structs/enums, globals
                and function prototypes for source file external_app.c
*****************************************************************************
/

#ifndef _EXTERNAL_APP_H_
#define _EXTERNAL_APP_H_

/*---------------------------------- INCLUDES ---------------------------
----*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#include <sys/socket.h>

#include <netinet/in.h>
#include <arpa/inet.h>

/*---------------------------------- INCLUDES ---------------------------
----*/

/*---------------------------------- MACROS -----------------------------
----*/
#define SERVER_PORT_NUM                         8500
#define SERVER_LISTEN_QUEUE_SIZE          5

#define BUFF_SIZE                                 1024

#define SOCK_REQ_MSG_API_MSG_LEN            64

/*---------------------------------- MACROS -----------------------------
----*/

/*---------------------------------- GLOBALS ----------------------------
----*/

/*---------------------------------- GLOBALS ----------------------------
----*/

/*---------------------------- STRUCTURES/ENUMERATIONS -------------------
----*/
enum _req_recipient_
{
    REQ_RECP_TEMP_TASK,
    REQ_RECP_LIGHT_TASK
};

struct _socket_req_msg_struct_
{
    char req_api_msg[SOCK_REQ_MSG_API_MSG_LEN];
    enum _req_recipient_ req_recipient;
    int params;
};
```

```
#if 0
struct _int_thresh_reg_struct_
{
    uint8_t thresh_low_low;
    uint8_t thresh_low_high;
    uint8_t thresh_high_low;
    uint8_t thresh_high_high;
};
#endif

struct _int_thresh_reg_struct_
{
    uint16_t low_thresh;
    uint16_t high_thresh;
};

/*---------------------------- STRUCTURES/ENUMERATIONS ------------------
----*/

/*---------------------------- FUNCTION PROTOTYPES ----------------------
----*/

/*---------------------------- FUNCTION PROTOTYPES ----------------------
----*/

#endif
/**************************************************************************
***
* Author:       Pavan Dhareshwar & Sridhar Pavithrapu
* Date:         03/10/2018
* File:         test_app.c
* Description:  Source file containing the functionality and
implementation
*               of external application
**************************************************************************
**/

#include "test_app.h"

int main(void)
{
    int client_sock;
    struct sockaddr_in serv_addr;

    char buffer[BUFF_SIZE];

    if ((client_sock = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    {
        printf("\n Socket creation error \n");
        return -1;
    }

    memset(&serv_addr, '0', sizeof(serv_addr));

    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(SERVER_PORT_NUM);

    // Convert IPv4 and IPv6 addresses from text to binary form
    if(inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr)<=0)
```

```
    {
        printf("\nInvalid address/ Address not supported \n");
        return -1;
    }

    if (connect(client_sock, (struct sockaddr *)&serv_addr,
sizeof(serv_addr)) < 0)
    {
        printf("\nConnection Failed \n");
        return -1;
    }

    struct _socket_req_msg_struct_ ext_app_req_msg = {0};
      int user_option = 0;

      /* Loop for getting input from the user for different operation */
      while(1)
      {
            /* Menu for the user */

      printf("\n/************************************************************
************/\n");
            printf("You can enter the option to perform the following
operations using this application:\n");
            printf("Enter (1) to get temperature sensor data.\n");
            printf("Enter (2) to get T-Low value of temperature sensor.
\n");
            printf("Enter (3) to get T-High value of temperature
sensor.\n");
            printf("Enter (4) to get temperature sensor configuration
register data.\n");
            printf("Enter (5) to get temperature sensor em.\n");
            printf("Enter (6) to get temperature sensor conversion
rate.\n");
            printf("Enter (7) to get temperature sensor fault bits.\n");
            printf("Enter (8) to control temperature sensor (ON-0 / OFF-
1).\n");
            printf("Enter (9) to set extended mode operation of
temperature sensor (Normal Mode-0 / Extended mode-1).\n");
            printf("Enter (10) to set conversion rate of temperature
sensor (0.2Hz-0 , 1Hz-1, 4Hz(Default)-2, 8hz-3).\n");
            printf("Enter (11) to set T-Low value of temperature sensor.
\n");
            printf("Enter (12) to set T-High value of temperature
sensor.\n");
            printf("Enter (13) to set fault bits of temperature
sensor.\n");

printf("/************************************************************
****/\n");
            printf("/*********************LIGHT TASK
OPERATIONS************************/\n");

      printf("/************************************************************
**********/\n");
            printf("You can enter the option to perform the following
operations using this application for temperature task:\n");
            printf("Enter (14) to get light sensor control register.\n");
            printf("Enter (15) to get light sensor lux data.\n");
            printf("Enter (16) to get light sensor ID.\n");
```

```
            printf("Enter (17) to get light sensor timing register.\n");
            printf("Enter (18) to get light sensor interrupt threshold
register.\n");
            printf("Enter (19) to set light sensor control register.\n");
            printf("Enter (20) to set light sensor integration time.\n");
            printf("Enter (21) to set light sensor gain.\n");
            printf("Enter (22) to set light sensor interrupt threshold
low.\n");
            printf("Enter (23) to set light sensor interrupt threshold
high.\n");

      printf("/************************************************************
**********/\n");

      printf("/*****************************EXIT***************************
**********/\n");

      printf("/************************************************************
**********/\n");
            printf("Enter (24) to exit the external application.\n");

      printf("/************************************************************
**********/\n");

            printf("\nEnter the option number you want to select:\n");
            scanf("%d",&user_option);

            if((user_option > 0) || (user_option < 25)){
             memset(buffer, '\0', sizeof(buffer));

                  if(user_option == 1){

                        strcpy(ext_app_req_msg.req_api_msg,
"get_temp_data");
                        ext_app_req_msg.req_recipient =
REQ_RECP_TEMP_TASK;
                        ext_app_req_msg.params = -1;

                        printf("Sending %s request to socket task\n",
ext_app_req_msg.req_api_msg);
                        ssize_t num_sent_bytes = send(client_sock,
&ext_app_req_msg,
                                    sizeof(struct _socket_req_msg_struct_),
0);
                        if (num_sent_bytes < 0)
                        {
                              perror("send failed");
                        }
                        else
                        {
                              /* Receiving message from parent process */
                              size_t num_read_bytes = read(client_sock,
buffer, sizeof(buffer));
                              printf("Message received in external app :
%s\n", buffer);
                        }

                  }
                  else if(user_option == 2){
```

```
                        strcpy(ext_app_req_msg.req_api_msg,
"get_temp_low_data");
                        ext_app_req_msg.req_recipient =
REQ_RECP_TEMP_TASK;
                        ext_app_req_msg.params = -1;

                        printf("Sending %s request to socket task\n",
ext_app_req_msg.req_api_msg);
                        ssize_t num_sent_bytes = send(client_sock,
&ext_app_req_msg,
                                    sizeof(struct _socket_req_msg_struct_),
0);
                        if (num_sent_bytes < 0)
                        {
                              perror("send failed");
                        }
                        else
                        {
                              /* Receiving message from parent process */
                              size_t num_read_bytes = read(client_sock,
buffer, sizeof(buffer));
                              printf("Message received in external app :
%s\n", buffer);
                        }

                  }
                  else if(user_option == 3){

                        strcpy(ext_app_req_msg.req_api_msg,
"get_temp_high_data");
                        ext_app_req_msg.req_recipient =
REQ_RECP_TEMP_TASK;
                        ext_app_req_msg.params = -1;

                        printf("Sending %s request to socket task\n",
ext_app_req_msg.req_api_msg);
                        ssize_t num_sent_bytes = send(client_sock,
&ext_app_req_msg,
                                    sizeof(struct _socket_req_msg_struct_),
0);
                        if (num_sent_bytes < 0)
                        {
                              perror("send failed");
                        }
                        else
                        {
                              /* Receiving message from parent process */
                              size_t num_read_bytes = read(client_sock,
buffer, sizeof(buffer));
                              printf("Message received in external app :
%s\n", buffer);
                        }

                  }
                  else if(user_option == 4){

                        strcpy(ext_app_req_msg.req_api_msg,
"get_temp_conf_data");
                        ext_app_req_msg.req_recipient =
REQ_RECP_TEMP_TASK;
```

```
                    ext_app_req_msg.params = -1;

                    printf("Sending %s request to socket task\n",
ext_app_req_msg.req_api_msg);
                    ssize_t num_sent_bytes = send(client_sock,
&ext_app_req_msg,
                                sizeof(struct _socket_req_msg_struct_),
0);
                    if (num_sent_bytes < 0)
                    {
                         perror("send failed");
                    }
                    else
                    {
                         /* Receiving message from parent process */
                         size_t num_read_bytes = read(client_sock,
buffer, sizeof(buffer));
                         printf("Message received in external app :
%s\n", buffer);
                    }

               }
               else if(user_option == 5){

                    strcpy(ext_app_req_msg.req_api_msg,
"get_temp_em");
                    ext_app_req_msg.req_recipient =
REQ_RECP_TEMP_TASK;
                    ext_app_req_msg.params = -1;

                    printf("Sending %s request to socket task\n",
ext_app_req_msg.req_api_msg);
                    ssize_t num_sent_bytes = send(client_sock,
&ext_app_req_msg,
                                sizeof(struct _socket_req_msg_struct_),
0);
                    if (num_sent_bytes < 0)
                    {
                         perror("send failed");
                    }
                    else
                    {
                         /* Receiving message from parent process */
                         size_t num_read_bytes = read(client_sock,
buffer, sizeof(buffer));
                         printf("Message received in external app :
%s\n", buffer);
                    }

               }
               else if(user_option == 6){

                    strcpy(ext_app_req_msg.req_api_msg,
"get_temp_conversion_rate");
                    ext_app_req_msg.req_recipient =
REQ_RECP_TEMP_TASK;
                    ext_app_req_msg.params = -1;

                    printf("Sending %s request to socket task\n",
ext_app_req_msg.req_api_msg);
```

```
                        ssize_t num_sent_bytes = send(client_sock,
&ext_app_req_msg,
                                    sizeof(struct _socket_req_msg_struct_),
0);
                        if (num_sent_bytes < 0)
                        {
                              perror("send failed");
                        }
                        else
                        {
                              /* Receiving message from parent process */
                              size_t num_read_bytes = read(client_sock,
buffer, sizeof(buffer));
                              printf("Message received in external app :
%s\n", buffer);
                        }

                  }
                  else if(user_option == 7){

                        strcpy(ext_app_req_msg.req_api_msg,
"get_temp_fault_bits");
                        ext_app_req_msg.req_recipient =
REQ_RECP_TEMP_TASK;
                        ext_app_req_msg.params = -1;

                        printf("Sending %s request to socket task\n",
ext_app_req_msg.req_api_msg);
                        ssize_t num_sent_bytes = send(client_sock,
&ext_app_req_msg,
                                    sizeof(struct _socket_req_msg_struct_),
0);
                        if (num_sent_bytes < 0)
                        {
                              perror("send failed");
                        }
                        else
                        {
                              /* Receiving message from parent process */
                              size_t num_read_bytes = read(client_sock,
buffer, sizeof(buffer));
                              printf("Message received in external app :
%s\n", buffer);
                        }

                  }
                  else if(user_option == 8){

                        strcpy(ext_app_req_msg.req_api_msg,
"set_temp_on_off");
                        ext_app_req_msg.req_recipient =
REQ_RECP_TEMP_TASK;

                        int temp_control = 0;
                        printf("Enter option to control temperature sensor
(ON-0 / OFF-1)\n");
                        scanf("%d",&temp_control);
                        ext_app_req_msg.params = temp_control;
```

```
                        printf("Sending %s request to socket task\n",
ext_app_req_msg.req_api_msg);
                        ssize_t num_sent_bytes = send(client_sock,
&ext_app_req_msg,
                                    sizeof(struct _socket_req_msg_struct_),
0);
                        if (num_sent_bytes < 0)
                        {
                              perror("send failed");
                        }
                        else
                        {
                              /* Receiving message from parent process */
                              size_t num_read_bytes = read(client_sock,
buffer, sizeof(buffer));
                              printf("Message received in external app :
%s\n", buffer);
                        }
                  }
                  else if(user_option == 9){

                        strcpy(ext_app_req_msg.req_api_msg,
"set_temp_em");
                        ext_app_req_msg.req_recipient =
REQ_RECP_TEMP_TASK;

                        uint8_t temp_conversion = 0;
                        printf("Enter option to set extended mode
operation of temperature sensor (Normal Mode-0 / Extended mode-1)\n");
                        scanf("%d", &temp_conversion);
                  printf("Temp_Conv: %d\n", temp_conversion);
                        ext_app_req_msg.params = temp_conversion;

                        printf("Sending %s request to socket task\n",
ext_app_req_msg.req_api_msg);
                        ssize_t num_sent_bytes = send(client_sock,
&ext_app_req_msg,
                                    sizeof(struct _socket_req_msg_struct_),
0);
                        if (num_sent_bytes < 0)
                        {
                              perror("send failed");
                        }
                        else
                        {
                              /* Receiving message from parent process */
                              size_t num_read_bytes = read(client_sock,
buffer, sizeof(buffer));
                              printf("Message received in external app :
%s\n", buffer);
                        }
                  }
                  else if(user_option == 10){

                        strcpy(ext_app_req_msg.req_api_msg,
"set_temp_conversion_rate");
                        ext_app_req_msg.req_recipient =
REQ_RECP_TEMP_TASK;

                        int temp_conversion=0;
```

```
                        printf("Enter option to set conversion rate of
temperature sensor (0.2Hz-0 , 1Hz-1, 4Hz(Default)-2, 8hz-3)\n");
                        scanf("%d",&temp_conversion);
                        ext_app_req_msg.params = temp_conversion;

                        printf("Sending %s request to socket task\n",
ext_app_req_msg.req_api_msg);
                        ssize_t num_sent_bytes = send(client_sock,
&ext_app_req_msg,
                                    sizeof(struct _socket_req_msg_struct_),
0);
                        if (num_sent_bytes < 0)
                        {
                              perror("send failed");
                        }
                        else
                        {
                              /* Receiving message from parent process */
                              size_t num_read_bytes = read(client_sock,
buffer, sizeof(buffer));
                              printf("Message received in external app :
%s\n", buffer);
                        }
                  }
                  else if(user_option == 11){

                        strcpy(ext_app_req_msg.req_api_msg,
"set_temp_low_data");
                        ext_app_req_msg.req_recipient =
REQ_RECP_TEMP_TASK;

                        int16_t temp_low=0;
                        printf("Enter option to set low threshold of
temperature sensor\n");
                        scanf("%d",&temp_low);
                        ext_app_req_msg.params = temp_low;

                        printf("Sending %s request to socket task\n",
ext_app_req_msg.req_api_msg);
                        ssize_t num_sent_bytes = send(client_sock,
&ext_app_req_msg,
                                    sizeof(struct _socket_req_msg_struct_),
0);
                        if (num_sent_bytes < 0)
                        {
                              perror("send failed");
                        }
                        else
                        {
                              /* Receiving message from parent process */
                              size_t num_read_bytes = read(client_sock,
buffer, sizeof(buffer));
                              printf("Message received in external app :
%s\n", buffer);
                        }
                  }
                  else if(user_option == 12){
                        strcpy(ext_app_req_msg.req_api_msg,
"set_temp_high_data");
```

```
                        ext_app_req_msg.req_recipient =
REQ_RECP_TEMP_TASK;

                        int16_t temp_high=0;
                        printf("Enter option to set high threshold of
temperature sensor \n");
                        scanf("%d",&temp_high);
                        ext_app_req_msg.params = temp_high;

                        printf("Sending %s request to socket task\n",
ext_app_req_msg.req_api_msg);
                        ssize_t num_sent_bytes = send(client_sock,
&ext_app_req_msg,
                                    sizeof(struct _socket_req_msg_struct_),
0);
                        if (num_sent_bytes < 0)
                        {
                              perror("send failed");
                        }
                        else
                        {
                              /* Receiving message from parent process */
                              size_t num_read_bytes = read(client_sock,
buffer, sizeof(buffer));
                              printf("Message received in external app :
%s\n", buffer);
                        }

                  }
                  else if(user_option == 13){

                        strcpy(ext_app_req_msg.req_api_msg,
"set_temp_fault_bits");
                        ext_app_req_msg.req_recipient =
REQ_RECP_TEMP_TASK;

                        uint8_t temp_fault=0;
                        printf("Enter option to set fault bits of
temperature sensor\n");
                        scanf("%d",&temp_fault);
                        ext_app_req_msg.params = temp_fault;

                        printf("Sending %s request to socket task\n",
ext_app_req_msg.req_api_msg);
                        ssize_t num_sent_bytes = send(client_sock,
&ext_app_req_msg,
                                    sizeof(struct _socket_req_msg_struct_),
0);
                        if (num_sent_bytes < 0)
                        {
                              perror("send failed");
                        }
                        else
                        {
                              /* Receiving message from parent process */
                              size_t num_read_bytes = read(client_sock,
buffer, sizeof(buffer));
                              printf("Message received in external app :
%s\n", buffer);
                        }
```

```
                }
                else if(user_option == 14){

                        strcpy(ext_app_req_msg.req_api_msg,
"get_light_sensor_ctrl_reg");
                        ext_app_req_msg.req_recipient =
REQ_RECP_LIGHT_TASK;
                        ext_app_req_msg.params = -1;

                        printf("Sending %s request to socket task\n",
ext_app_req_msg.req_api_msg);
                        ssize_t num_sent_bytes = send(client_sock,
&ext_app_req_msg,
                                    sizeof(struct _socket_req_msg_struct_),
0);
                        if (num_sent_bytes < 0)
                        {
                              perror("send failed");
                        }
                        else
                        {
                              /* Receiving message from parent process */
                              size_t num_read_bytes = read(client_sock,
buffer, sizeof(buffer));
                              printf("Message received in external app :
%s\n", buffer);
                        }

                }
                else if(user_option == 15){

                        strcpy(ext_app_req_msg.req_api_msg,
"get_lux_data");
                        ext_app_req_msg.req_recipient =
REQ_RECP_LIGHT_TASK;
                        ext_app_req_msg.params = -1;

                        printf("Sending %s request to socket task\n",
ext_app_req_msg.req_api_msg);
                        ssize_t num_sent_bytes = send(client_sock,
&ext_app_req_msg,
                                    sizeof(struct _socket_req_msg_struct_),
0);
                        if (num_sent_bytes < 0)
                        {
                              perror("send failed");
                        }
                        else
                        {
                              /* Receiving message from parent process */
                              size_t num_read_bytes = read(client_sock,
buffer, sizeof(buffer));
                              printf("Message received in external app :
%s\n", buffer);
                        }

                }
                else if(user_option == 16){
```

```
                        strcpy(ext_app_req_msg.req_api_msg,
"get_light_sensor_id");
                        ext_app_req_msg.req_recipient =
REQ_RECP_LIGHT_TASK;
                        ext_app_req_msg.params = -1;

                        printf("Sending %s request to socket task\n",
ext_app_req_msg.req_api_msg);
                        ssize_t num_sent_bytes = send(client_sock,
&ext_app_req_msg,
                                    sizeof(struct _socket_req_msg_struct_),
0);
                        if (num_sent_bytes < 0)
                        {
                              perror("send failed");
                        }
                        else
                        {
                              /* Receiving message from parent process */
                              size_t num_read_bytes = read(client_sock,
buffer, sizeof(buffer));
                              printf("Message received in external app :
%s\n", buffer);
                        }

                  }
                  else if(user_option == 17){

                        strcpy(ext_app_req_msg.req_api_msg,
"get_light_sensor_tim_reg");
                        ext_app_req_msg.req_recipient =
REQ_RECP_LIGHT_TASK;
                        ext_app_req_msg.params = -1;

                        printf("Sending %s request to socket task\n",
ext_app_req_msg.req_api_msg);
                        ssize_t num_sent_bytes = send(client_sock,
&ext_app_req_msg,
                                    sizeof(struct _socket_req_msg_struct_),
0);
                        if (num_sent_bytes < 0)
                        {
                              perror("send failed");
                        }
                        else
                        {
                              /* Receiving message from parent process */
                              size_t num_read_bytes = read(client_sock,
buffer, sizeof(buffer));
                              printf("Message received in external app :
%s\n", buffer);
                        }

                  }
                  else if(user_option == 18){

                        strcpy(ext_app_req_msg.req_api_msg,
"get_light_sensor_int_thresh_reg");
                        ext_app_req_msg.req_recipient =
REQ_RECP_LIGHT_TASK;
```

```
                        ext_app_req_msg.params = -1;

                        printf("Sending %s request to socket task\n",
ext_app_req_msg.req_api_msg);
                        ssize_t num_sent_bytes = send(client_sock,
&ext_app_req_msg,
                                    sizeof(struct _socket_req_msg_struct_),
0);
                        if (num_sent_bytes < 0)
                        {
                              perror("send failed");
                        }
                        else
                        {
                      printf("WAITING ON READ\n");
                              /* Receiving message from parent process */
                              struct _int_thresh_reg_struct_
int_thresh_reg_struct = {0};
                      size_t num_read_bytes = read(client_sock,
&int_thresh_reg_struct, sizeof(struct _int_thresh_reg_struct_));
                              //size_t num_read_bytes = read(client_sock,
buffer, sizeof(buffer));
                      printf("READ SUCCESS\n");

                      printf("Low Threshold : %d\n",
int_thresh_reg_struct.low_thresh);
                      printf("High Threshold : %d\n",
int_thresh_reg_struct.high_thresh);
                        }
                  }
                  else if(user_option == 19){

                        strcpy(ext_app_req_msg.req_api_msg,
"set_light_sensor_ctrl_reg");
                        ext_app_req_msg.req_recipient =
REQ_RECP_LIGHT_TASK;

                        uint8_t reg_value=0;
                        printf("Enter option to set control register value
of light sensor\n");
                        scanf("%d",&reg_value);
                        ext_app_req_msg.params = (int )reg_value;

                        printf("Sending %s request to socket task\n",
ext_app_req_msg.req_api_msg);
                        ssize_t num_sent_bytes = send(client_sock,
&ext_app_req_msg,
                                    sizeof(struct _socket_req_msg_struct_),
0);
                        if (num_sent_bytes < 0)
                        {
                              perror("send failed");
                        }
                        else
                        {
                              /* Receiving message from parent process */
                              size_t num_read_bytes = read(client_sock,
buffer, sizeof(buffer));
                              printf("Message received in external app :
%s\n", buffer);
```

```
                    }
                }
                else if(user_option == 20){

                    strcpy(ext_app_req_msg.req_api_msg,
"set_light_sensor_integration_time");
                    ext_app_req_msg.req_recipient =
REQ_RECP_LIGHT_TASK;

                    uint8_t integration_time=0;
                    printf("Enter option to set integration time of
light sensor\n");
                    scanf("%d",&integration_time);
                    ext_app_req_msg.params = (int )integration_time;

                    printf("Sending %s request to socket task\n",
ext_app_req_msg.req_api_msg);
                    ssize_t num_sent_bytes = send(client_sock,
&ext_app_req_msg,
                                sizeof(struct _socket_req_msg_struct_),
0);
                    if (num_sent_bytes < 0)
                    {
                        perror("send failed");
                    }
                    else
                    {
                        /* Receiving message from parent process */
                        size_t num_read_bytes = read(client_sock,
buffer, sizeof(buffer));
                        printf("Message received in external app :
%s\n", buffer);
                    }
                }
                else if(user_option == 21){

                    strcpy(ext_app_req_msg.req_api_msg,
"set_light_sensor_gain");
                    ext_app_req_msg.req_recipient =
REQ_RECP_LIGHT_TASK;

                    uint8_t gain_value=0;
                    printf("Enter option to set gain value of light
sensor\n");
                    scanf("%d",&gain_value);
                    ext_app_req_msg.params = (int )gain_value;

                    printf("Sending %s request to socket task\n",
ext_app_req_msg.req_api_msg);
                    ssize_t num_sent_bytes = send(client_sock,
&ext_app_req_msg,
                                sizeof(struct _socket_req_msg_struct_),
0);
                    if (num_sent_bytes < 0)
                    {
                        perror("send failed");
                    }
                    else
                    {
                        /* Receiving message from parent process */
```

```
                              size_t num_read_bytes = read(client_sock,
buffer, sizeof(buffer));
                              printf("Message received in external app :
%s\n", buffer);
                        }
                  }
                  else if(user_option == 22){

                        strcpy(ext_app_req_msg.req_api_msg,
"set_interrupt_low_threshold");
                        ext_app_req_msg.req_recipient =
REQ_RECP_LIGHT_TASK;

                        uint16_t low_threshold_value=0;
                        printf("Enter option to set low interrupt
threshold of light sensor\n");
                        scanf("%d",&low_threshold_value);
                        ext_app_req_msg.params = (int
)low_threshold_value;

                        printf("Sending %s request to socket task\n",
ext_app_req_msg.req_api_msg);
                        ssize_t num_sent_bytes = send(client_sock,
&ext_app_req_msg,
                                    sizeof(struct _socket_req_msg_struct_),
0);
                        if (num_sent_bytes < 0)
                        {
                              perror("send failed");
                        }
                        else
                        {
                              /* Receiving message from parent process */
                              size_t num_read_bytes = read(client_sock,
buffer, sizeof(buffer));
                              printf("Message received in external app :
%s\n", buffer);
                        }
                  }
                  else if(user_option == 23){

                        strcpy(ext_app_req_msg.req_api_msg,
"set_interrupt_high_threshold");
                        ext_app_req_msg.req_recipient =
REQ_RECP_LIGHT_TASK;

                        uint16_t high_threshold_value=0;
                        printf("Enter option to set low interrupt
threshold of light sensor\n");
                        scanf("%d",&high_threshold_value);
                        ext_app_req_msg.params = (int
)high_threshold_value;

                        printf("Sending %s request to socket task\n",
ext_app_req_msg.req_api_msg);
                        ssize_t num_sent_bytes = send(client_sock,
&ext_app_req_msg,
                                    sizeof(struct _socket_req_msg_struct_),
0);
                        if (num_sent_bytes < 0)
```

```
                        {
                              perror("send failed");
                        }
                        else
                        {
                              /* Receiving message from parent process */
                              size_t num_read_bytes = read(client_sock,
buffer, sizeof(buffer));
                              printf("Message received in external app :
%s\n", buffer);
                        }
                  }
                  else if(user_option == 24){

                        exit(0);
                  }

            else{
                        printf("Invalid option selected, please select the
correct option.\n");
                  }

            }
            else{
                  printf("Invalid option selected, please select the
correct option.\n");
            }

      }

      return 0;
}
/*************************************************************************
*
* Author:       Pavan Dhareshwar & Sridhar Pavithrapu
* Date:         03/08/2018
* File:         logger_task.h
* Description:  Header file containing the macros, structs/enums, globals
                and function prototypes for source file logger_task.c
*************************************************************************
/

#ifndef _LOGGER_TASK_H_
#define _LOGGER_TASK_H_

/*---------------------------------- INCLUDES ----------------------------
----*/
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <stdint.h>
#include <string.h>
#include <time.h>

#include <unistd.h>
#include <fcntl.h>
#include <pthread.h>
#include <signal.h>

#include <sys/types.h>
```

```
#include <sys/stat.h>
#include <sys/ipc.h>
#include <sys/types.h>
#include <sys/msg.h>

#include <mqueue.h>

#include <netinet/in.h>
#include <arpa/inet.h>
/*---------------------------------- INCLUDES ----------------------------
----*/

/*---------------------------------- MACROS ----------------------------
----*/
// Message queue attribute macros
#define MSG_QUEUE_MAX_NUM_MSGS              5
#define MSG_QUEUE_MAX_MSG_SIZE              1024
#define MSG_QUEUE_NAME                      "/logger_task_mq"

#define LOGGER_FILE_PATH                    "./"
#define LOGGER_FILE_NAME                    "logger_file.txt"

#define LOG_MSG_PAYLOAD_SIZE                256
#define MSG_MAX_LEN                         128

#define MSG_BUFF_MAX_LEN                    1024

#define LOGGER_FILE_PATH_LEN                256
#define LOGGER_FILE_NAME_LEN                64

#define SOCKET_HB_PORT_NUM                  8680
#define SOCKET_HB_LISTEN_QUEUE_SIZE         10

#define MSG_TYPE_TEMP_DATA                  0
#define MSG_TYPE_LUX_DATA                   1
#define MSG_TYPE_SOCK_DATA                  2
#define MSG_TYPE_MAIN_DATA                  3

#define LOGGER_ATTR_LEN                     32

/*---------------------------------- MACROS ----------------------------
----*/

/*---------------------------------- GLOBALS ----------------------------
----*/
mqd_t logger_mq_handle;
int logger_fd;
pthread_t logger_thread_id, socket_hb_thread_id;

sig_atomic_t g_sig_kill_logger_thread, g_sig_kill_sock_hb_thread;

/*---------------------------------- GLOBALS ----------------------------
----*/

/*---------------------------- STRUCTURES/ENUMERATIONS -----------------
----*/
struct _logger_msg_struct_
{
    char message[MSG_MAX_LEN];
    char logger_msg_src_id[LOGGER_ATTR_LEN];
```

```
    char logger_msg_level[LOGGER_ATTR_LEN];
};

/*---------------------------- STRUCTURES/ENUMERATIONS ------------------
----*/

/*---------------------------- FUNCTION PROTOTYPES ----------------------
----*/
/**
 *  @brief Initialize the logger task
 *
 *  This funcition will create the message queue for logger task and
 *  open a file handle of logger file for writing. (If the logger file
 *  already exists, it is deleted and a fresh one is created).
 *
 *  @param void
 *
 *  @return 0  : if sensor initialization is a success
            -1 : if sensor initialization fails
*/
int logger_task_init();

/**
 *  @brief Read from configuration file for the logger task
 *
 *  This function reads the configuration parameters for the logger task
file
 *  and sets-up the logger file as per this configuration
 *
 *  @param file     : name of the config file
 *
 *  @return void
 *
 */
int read_logger_conf_file(char *file);

/**
 *  @brief Create logger and hearbeat socket threads for logger task
 *
 *  The logger task is made multi-threaded with
 *     1. logger thread responsible for reading messages from its message
queue
 *        and logging it to a file.
 *     2. socket heartbeat responsible for communicating with main task,
 *        to log heartbeat every time its requested by main task.
 *
 *  @param void
 *
 *  @return 0  : thread creation success
 *          -1 : thread creation failed
 *
 */
int create_threads(void);

/**
 *  @brief Entry point and executing entity for logger thread
 *
 *  The logger thread starts execution by invoking this
function(start_routine)
 *
```

```
 *  @param arg : argument to start_routine
 *
 *  @return void
 *
 */
void *logger_thread_func(void *arg);

/**
 *  @brief Entry point and executing entity for socket thread
 *
 *  The socket thread for heartbeat starts execution by invoking this
function(start_routine)
 *
 *  @param arg : argument to start_routine
 *
 *  @return void
 *
 */
void *socket_hb_thread_func(void *arg);

/**
 *  @brief Create the socket and initialize
 *
 *  This function create the socket for the given socket id.
 *
 *  @param sock_fd              : socket file descriptor
 *         server_addr_struct   : server address of the socket
 *         port_num             : port number in which the socket is
communicating
 *         listen_qsize         : number of connections the socket is
accepting
 *
 *  @return void
*/
void init_sock(int *sock_fd, struct sockaddr_in *server_addr_struct,
               int port_num, int listen_qsize);


int write_test_msg_to_logger();
void read_test_msg_to_logger(char * buffer);

/**
 *  @brief Read message from logger message queue
 *
 *  This function will read messages from its message queue and log it to
a file
 *
 *  @param void
 *
 *  @return void
*/
void read_from_logger_msg_queue(void);

/**
 *  @brief Cleanup of the logger sensor
 *
 *  This function will close the message queue and the logger file handle
 *
 *  @param void
 *
```

```
 *  @return void
*/
void logger_task_exit(void);

/**
 *  @brief Signal handler for temperature task
 *
 *  This function handles the reception of SIGKILL and SIGINT signal to
the
 *  temperature task and terminates all the threads, closes the I2C file
descriptor
 *  and logger message queue handle and exits.
 *
 *  @param sig_num              : signal number
 *
 *  @return void
*/

void sig_handler(int sig_num);
#endif // _LOGGER_TASK_H_
/*****************************************************************************
*
* Author:       Pavan Dhareshwar & Sridhar Pavithrapu
* Date:         03/08/2018
* File:         logger_task.c
* Description:  Source file describing the functionality and
implementation
*               of logger task.
*****************************************************************************
/

#include "logger_task.h"

int main(void)
{
    printf("In Logger task\n");

    int init_status = logger_task_init();
    if (init_status == -1)
    {
        printf("logger task initialization failed\n");
        exit(1);
    }

    //write_test_msg_to_logger();

    int thread_create_status = create_threads();
    if (thread_create_status)
    {
        printf("Thread creation failed\n");
    }
    else
    {
        printf("Thread creation success\n");
    }

    if (signal(SIGINT, sig_handler) == SIG_ERR)
        printf("SigHandler setup for SIGINT failed\n");

    if (signal(SIGUSR1, sig_handler) == SIG_ERR)
```

```
        printf("SigHandler setup for SIGKILL failed\n");

    g_sig_kill_logger_thread = 0;
    g_sig_kill_sock_hb_thread = 0;

    pthread_join(logger_thread_id, NULL);
    pthread_join(socket_hb_thread_id, NULL);

    logger_task_exit();

    return 0;
}

int logger_task_init()
{
    /* In the logger task init function, we create the message queue */

    /* Set the message queue attributes */
    struct mq_attr logger_mq_attr = { .mq_flags = 0,
                                      .mq_maxmsg =
MSG_QUEUE_MAX_NUM_MSGS,  // Max number of messages on queue
                                      .mq_msgsize =
MSG_QUEUE_MAX_MSG_SIZE  // Max. message size
                                    };


    logger_mq_handle = mq_open(MSG_QUEUE_NAME, O_CREAT | O_RDWR, S_IRWXU,
&logger_mq_attr);
    if (logger_mq_handle < 0)
    {
        perror("Logger message queue create failed");
        return -1;
    }

    printf("Logger message queue successfully created\n");

    char filename[100];
    memset(filename, '\0', sizeof(filename));
    int conf_file_read_status = read_logger_conf_file(filename);
    if (conf_file_read_status != 0)
    {
        printf("Logger task config file read failed. Using default log
file path and name\n");
        sprintf(filename, "%s%s", LOGGER_FILE_PATH, LOGGER_FILE_NAME);
    }

    if (open(filename, O_RDONLY) != -1)
    {
        printf("Logger file exists. Deleting existing file.\n");
        remove(filename);
        sync();
    }

    printf("Trying to create file %s\n", filename);
    logger_fd = creat(filename ,(S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH));
    if (logger_fd == -1)
    {
        perror("Logger file open failed");
        return -1;
    }
```

```
    else
    {
        printf("Logger file open success\n");
    }

    return 0;
}

int read_logger_conf_file(char *file)
{
    FILE *fp_conf_file = fopen("./logger_task_conf_file.txt", "r");
    if (fp_conf_file == NULL)
    {
        perror("file open failed");
        printf("File %s open failed\n", "logger_task_conf_file.txt");
        return -1;
    }

    char logger_file_path[LOGGER_FILE_PATH_LEN];
    char logger_file_name[LOGGER_FILE_NAME_LEN];
    char *buffer;
    size_t num_bytes = 120;
    char equal_delimiter[] = "=";
    ssize_t bytes_read;

    memset(logger_file_path, '\0', sizeof(logger_file_path));
    memset(logger_file_name, '\0', sizeof(logger_file_name));

    buffer = (char *)malloc(num_bytes*sizeof(char));

    while ((bytes_read = getline(&buffer, &num_bytes, fp_conf_file)) != -
1)
    {
        char *token = strtok(buffer, equal_delimiter);

        if (!strcmp(token, "LOGGER_FILE_PATH"))
        {
            token = strtok(NULL, equal_delimiter);
            strcpy(logger_file_path, token);
            int len = strlen(logger_file_path);
            if (logger_file_path[len-1] == '\n')
                logger_file_path[len-1] = '\0';
        }
        else if (!strcmp(token, "LOGGER_FILE_NAME"))
        {
            token = strtok(NULL, equal_delimiter);
            strcpy(logger_file_name, token);
            int len = strlen(logger_file_name);
            if (logger_file_name[len-1] == '\n')
                logger_file_name[len-1] = '\0';
        }
    }

    strcpy(file, logger_file_path);
    strcat(file, logger_file_name);

    if (buffer)
        free(buffer);

    if (fp_conf_file)
```

```
        fclose(fp_conf_file);

    return 0;
}

int create_threads(void)
{
    int logger_t_creat_ret_val = pthread_create(&logger_thread_id, NULL,
&logger_thread_func, NULL);
    if (logger_t_creat_ret_val)
    {
        perror("Sensor thread creation failed");
        return -1;
    }

    int sock_hb_t_creat_ret_val = pthread_create(&socket_hb_thread_id,
NULL, &socket_hb_thread_func, NULL);
    if (sock_hb_t_creat_ret_val)
    {
        perror("Socket heartbeat thread creation failed");
        return -1;
    }

    return 0;
}

void *logger_thread_func(void *arg)
{
    while(!g_sig_kill_logger_thread)
    {
        /* This function will continously read from the logger task
message
        ** queue and write it to logger file */
        read_from_logger_msg_queue();
    }

    pthread_exit(NULL);
}

void *socket_hb_thread_func(void *arg)
{
    int sock_hb_fd;
    struct sockaddr_in sock_hb_address;
    int sock_hb_addr_len = sizeof(sock_hb_address);

    init_sock(&sock_hb_fd, &sock_hb_address, SOCKET_HB_PORT_NUM,
SOCKET_HB_LISTEN_QUEUE_SIZE);


    int accept_conn_id;
    printf("Waiting for request...\n");
    if ((accept_conn_id = accept(sock_hb_fd, (struct sockaddr
*)&sock_hb_address,
                    (socklen_t*)&sock_hb_addr_len)) < 0)
    {
        perror("accept failed");
        //pthread_exit(NULL);
    }

    char recv_buffer[MSG_BUFF_MAX_LEN];
```

```
    char send_buffer[] = "Alive";

    while (!g_sig_kill_sock_hb_thread)
    {
        memset(recv_buffer, '\0', sizeof(recv_buffer));

        size_t num_read_bytes = read(accept_conn_id, &recv_buffer,
sizeof(recv_buffer));

        if (!strcmp(recv_buffer, "heartbeat"))
        {
                 ssize_t num_sent_bytes = send(accept_conn_id,
send_buffer, strlen(send_buffer), 0);
            if (num_sent_bytes < 0)
                perror("send failed");
        }
    }
}


int write_test_msg_to_logger(char *test_msg)
{
    struct _logger_msg_struct_ logger_msg = {0};

    strcpy(logger_msg.message, test_msg);
    logger_msg.msg_len = strlen(test_msg);

    logger_msg.logger_msg_type = MSG_TYPE_TEMP_DATA;

    int msg_priority = 1;
    int num_sent_bytes = mq_send(logger_mq_handle, (char *)&logger_msg,
                                    sizeof(logger_msg), msg_priority);

    if (num_sent_bytes < 0)
        perror("mq_send failed");
           return -1;

      return 0;
}

void read_test_msg_to_logger(char * buffer)
{
    char recv_buffer[MSG_MAX_LEN];
    memset(recv_buffer, '\0', sizeof(recv_buffer));

    int msg_priority;

    int num_recv_bytes;
    num_recv_bytes = mq_receive(logger_mq_handle, (char *)&recv_buffer,
                                    MSG_QUEUE_MAX_MSG_SIZE,
&msg_priority));

      strcpy(buffer, recv_buffer);

}


void init_sock(int *sock_fd, struct sockaddr_in *server_addr_struct,
               int port_num, int listen_qsize)
{
```

```
    int serv_addr_len = sizeof(struct sockaddr_in);

    /* Create the socket */
    if ((*sock_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0)
    {
        perror("socket creation failed");
        pthread_exit(NULL); // Change these return values from
pthread_exit
    }

    int option = 1;
    if(setsockopt(*sock_fd, SOL_SOCKET, (SO_REUSEPORT | SO_REUSEADDR),
(void *)&option, sizeof(option)) < 0)
    {
        perror("setsockopt failed");
        pthread_exit(NULL);
    }

    server_addr_struct->sin_family = AF_INET;
    server_addr_struct->sin_addr.s_addr = INADDR_ANY;
    server_addr_struct->sin_port = htons(port_num);

    if (bind(*sock_fd, (struct sockaddr *)server_addr_struct,
                                            sizeof(struct
sockaddr_in))<0)
    {
        perror("bind failed");
        pthread_exit(NULL);
    }

    if (listen(*sock_fd, listen_qsize) < 0)
    {
        perror("listen failed");
        pthread_exit(NULL);
    }

}

void read_from_logger_msg_queue(void)
{
    char recv_buffer[MSG_MAX_LEN];
    memset(recv_buffer, '\0', sizeof(recv_buffer));

    int msg_priority;

    int num_recv_bytes;
    while ((num_recv_bytes = mq_receive(logger_mq_handle, (char
*)&recv_buffer,
                                    MSG_QUEUE_MAX_MSG_SIZE,
&msg_priority)) != -1)
    {
        if (num_recv_bytes < 0)
        {
            perror("mq_receive failed");
            return;
        }

#if 0
        printf("Message received: %s, msg_src: %s, message level: %s\n",
            (((struct _logger_msg_struct_ *)&recv_buffer)->message),
```

```
            (((struct _logger_msg_struct_ *)&recv_buffer)-
>logger_msg_src_id),
            (((struct _logger_msg_struct_ *)&recv_buffer)-
>logger_msg_level));
#endif

        time_t tval = time(NULL);
        struct tm *cur_time = localtime(&tval);

        char timestamp_str[32];
        memset(timestamp_str, '\0', sizeof(timestamp_str));

        sprintf(timestamp_str, "%02d:%02d:%02d", cur_time->tm_hour,
cur_time->tm_min, cur_time->tm_sec);

        char msg_to_write[LOG_MSG_PAYLOAD_SIZE];
        memset(msg_to_write, '\0', sizeof(msg_to_write));

        sprintf(msg_to_write, "Timestamp: %s | Message_Src: %s |
Message_Type: %s | Message: %s\n",
            timestamp_str, (((struct _logger_msg_struct_ *)&recv_buffer)-
>logger_msg_src_id),
            (((struct _logger_msg_struct_ *)&recv_buffer)-
>logger_msg_level),
            (((struct _logger_msg_struct_ *)&recv_buffer)->message));

        printf("Message to write: %s\n", msg_to_write);
        int num_written_bytes = write(logger_fd, msg_to_write,
strlen(msg_to_write));
    }

}

void sig_handler(int sig_num)
{
    char buffer[MSG_BUFF_MAX_LEN];
    memset(buffer, '\0', sizeof(buffer));

    if (sig_num == SIGINT || sig_num == SIGUSR1)
    {
        if (sig_num == SIGINT)
            printf("Caught signal %s in logger task\n", "SIGINT");
        else if (sig_num == SIGUSR1)
            printf("Caught signal %s in logger task\n", "SIGKILL");

        g_sig_kill_logger_thread = 1;
        g_sig_kill_sock_hb_thread = 1;

        //pthread_join(sensor_thread_id, NULL);
        //pthread_join(socket_thread_id, NULL);
        //pthread_join(socket_hb_thread_id, NULL);

        mq_close(logger_mq_handle);

        exit(0);
    }
}

void logger_task_exit(void)
{
```

```
    int mq_close_status = mq_close(logger_mq_handle);
    if (mq_close_status == -1)
        perror("Logger message queue close failed");

    if (logger_fd)
        close(logger_fd);

}
/*************************************************************************
*************
*
* FileName        :    temp_unit_tests.c
* Description     :    This file contains necessary  test functions for
temperature task.

* File Author Name:    Sridhar Pavithrapu
* Tools used      :    gcc, gedit, cmocka
* References      :    None
*
*************************************************************************
**************/

/* Headers Section */
#include <math.h>
#include <stdlib.h>
#include <stdarg.h>
#include <stddef.h>
#include <setjmp.h>
#include <cmocka.h>
#include "logger_task.h"

/* Macros section */

/**
*   @brief :  test function for checking write to message queue
*
*
*   @param  state   A pointer to the state
*
*   @return None
*/
void test_write_message_queue(void **state){

      /* Test case for checking write to message queue */
      char buffer[MSG_MAX_LEN] = "This is a test case";
      int return_value = write_test_msg_to_logger(buffer);
      assert_int_equal(return_value, 0);

}

/**
*   @brief :  test function for checking read to message queue
*
*
*   @param  state   A pointer to the state
*
*   @return None
*/
void test_read_message_queue(void **state){
```

```
      /* Test case for checking read to message queue */
      char buffer[MSG_MAX_LEN] = "This is a test case";
      char buffer_output[MSG_MAX_LEN];
      int return_value = write_test_msg_to_logger(buffer);
      if(return_value == 0){
            read_test_msg_to_logger(buffer_output);
            assert_string_equal(buffer,buffer_output);
      }
      else{
            assert_int_equal(return_value, 0);
      }
}

/**
*   @brief : main function for all DLL test cases
*
*
*   @return Pass and Fail test cases
*/
int main(int argc, char **argv){

      logger_task_init();

      /* Calling all DLL test case functions */
      const struct CMUnitTest tests[] = {
            cmocka_unit_test(test_write_message_queue),
            cmocka_unit_test(test_read_message_queue),

      };

      return cmocka_run_group_tests(tests, NULL, NULL);
}/*************************************************************************
**
* Author:       Pavan Dhareshwar & Sridhar Pavithrapu
* Date:         03/07/2018
* File:         light_sensor.c
* Description:  Source file describing the functionality and
implementation
*               of light sensor task.
*************************************************************************
/

#include <stdio.h>
#include <stdlib.h>

#include "light_sensor.h"

int main(void){

    int init_ret_val = light_sensor_init();
    if (init_ret_val == -1)
    {
        printf("Light sensor init failed\n");
        exit(1);
    }

    printf("Creating threads\n");
    int thread_create_status = create_threads();
    if (thread_create_status)
    {
```

```
        printf("Thread creation failed\n");
    }
    else
    {
        printf("Thread creation success\n");
    }

    if (signal(SIGINT, sig_handler) == SIG_ERR)
        printf("SigHandler setup for SIGINT failed\n");

    if (signal(SIGUSR1, sig_handler) == SIG_ERR)
        printf("SigHandler setup for SIGKILL failed\n");

    g_sig_kill_sensor_thread = 0;
    g_sig_kill_sock_thread = 0;
    g_sig_kill_sock_hb_thread = 0;

    pthread_join(sensor_thread_id, NULL);
    pthread_join(socket_thread_id, NULL);
    pthread_join(socket_hb_thread_id, NULL);

     light_sensor_exit();

     return 0;
}

int light_sensor_init(void)
{
     /* Open the i2c bus for read and write operation */
    printf("Opening i2c bus %s\n", I2C_DEV_NAME);
    if ((i2c_light_sensor_fd = open(I2C_DEV_NAME,O_RDWR)) < 0) {
           perror("Failed to open i2c bus.");
           /* ERROR HANDLING; you can check errno to see what went wrong
*/
           return -1;
     }

     if (ioctl(i2c_light_sensor_fd,I2C_SLAVE,I2C_SLAVE_ADDR) < 0) {
           perror("Failed to acquire bus access and/or talk to slave.");
           /* ERROR HANDLING; you can check errno to see what went wrong
*/
           return -1;
     }

    printf("Powering on light sensor\n");
    /* Power on the APDS-9301 device */
    power_on_light_sensor();
    printf("Powered on light sensor\n");

    return 0;
}

void power_on_light_sensor(void)
{
    int cmd_ctrl_reg_val = I2C_LIGHT_SENSOR_CMD_CTRL_REG;

     int ctrl_reg_val = I2C_LIGHT_SENSOR_CTRL_REG_VAL;

    write_light_sensor_reg(cmd_ctrl_reg_val, ctrl_reg_val);
```

```
      cmd_ctrl_reg_val = I2C_LIGHT_SENSOR_CMD_TIM_REG;
      ctrl_reg_val = 0X10;
      write_light_sensor_reg(cmd_ctrl_reg_val, ctrl_reg_val);

}

int create_threads(void)
{
    int sens_t_creat_ret_val = pthread_create(&sensor_thread_id, NULL,
&sensor_thread_func, NULL);
    if (sens_t_creat_ret_val)
    {
        perror("Sensor thread creation failed");
        return -1;
    }

    int sock_t_creat_ret_val = pthread_create(&socket_thread_id, NULL,
&socket_thread_func, NULL);
    if (sock_t_creat_ret_val)
    {
        perror("Socket thread creation failed");
        return -1;
    }

      int sock_hb_t_creat_ret_val = pthread_create(&socket_hb_thread_id,
NULL, &socket_hb_thread_func, NULL);
    if (sock_hb_t_creat_ret_val)
    {
        perror("Socket heartbeat thread creation failed");
        return -1;
    }

    return 0;
}

void init_light_socket(struct sockaddr_in *sock_addr_struct)
{

    /* Create the socket */
    if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0)
    {
        perror("socket creation failed");
        pthread_exit(NULL); // Change these return values from
pthread_exit
    }

    int option = 1;
    if(setsockopt(server_fd, SOL_SOCKET, (SO_REUSEPORT | SO_REUSEADDR),
                (void *)&option, sizeof(option)) < 0)
    {
        perror("setsockopt failed");
        pthread_exit(NULL);
    }

    sock_addr_struct->sin_family = AF_INET;
    sock_addr_struct->sin_addr.s_addr = INADDR_ANY;
    sock_addr_struct->sin_port = htons(LIGHT_SENSOR_SERVER_PORT_NUM);

    if (bind(server_fd, (struct sockaddr *)sock_addr_struct,
```

```
                                                sizeof(struct
sockaddr_in))<0)
    {
        perror("bind failed");
        pthread_exit(NULL);
    }

    if (listen(server_fd, LIGHT_SENSOR_LISTEN_QUEUE_SIZE) < 0)
    {
        perror("listen failed");
        pthread_exit(NULL);
    }
}

void *socket_thread_func(void *arg)
{
    struct sockaddr_in server_address;
    int serv_addr_len = sizeof(server_address);

    init_light_socket(&server_address);

    char recv_buffer[MSG_BUFF_MAX_LEN];

    int accept_conn_id;
    printf("Waiting for request...\n");
    if ((accept_conn_id = accept(server_fd, (struct sockaddr
*)&server_address,
                    (socklen_t*)&serv_addr_len)) < 0)
    {
        perror("accept failed");
        //pthread_exit(NULL);
    }

    while (!g_sig_kill_sock_thread)
    {
        memset(recv_buffer, '\0', sizeof(recv_buffer));

        size_t num_read_bytes = read(accept_conn_id, &recv_buffer,
sizeof(recv_buffer));

        printf("[Light_Task] Message req api: %s, req recp: %s, req api
params: %s\n",
                ((struct _socket_req_msg_struct_ *)&recv_buffer)-
>req_api_msg),
                ((((struct _socket_req_msg_struct_ *)&recv_buffer)-
>req_recipient)
                 == REQ_RECP_TEMP_TASK ? "Temp Task" : "Light Task"),
                ((struct _socket_req_msg_struct_ *)&recv_buffer)-
>ptr_param_list != NULL ?
                 ((struct _socket_req_msg_struct_ *)&recv_buffer)-
>ptr_param_list :"NULL"));

        char light_sensor_rsp_msg[64];
        if (!strcmp((((struct _socket_req_msg_struct_ *)&recv_buffer)-
>req_api_msg), "get_lux_data"))
        {
            float lux_data = get_lux_data();
            memset(light_sensor_rsp_msg, '\0',
sizeof(light_sensor_rsp_msg));
```

```
            sprintf(light_sensor_rsp_msg, "Lux Data: %3.2f", lux_data);

            ssize_t num_sent_bytes = send(accept_conn_id,
light_sensor_rsp_msg, strlen(light_sensor_rsp_msg), 0);
            if (num_sent_bytes < 0)
                perror("send failed");

        }
        else if (!strcmp((((struct _socket_req_msg_struct_
*)&recv_buffer)->req_api_msg), "get_light_sensor_id"))
        {
            uint8_t light_sen_id_reg_val = read_id_reg();
            printf("id reg val : %d\n", light_sen_id_reg_val);

            memset(light_sensor_rsp_msg, '\0',
sizeof(light_sensor_rsp_msg));

            sprintf(light_sensor_rsp_msg, "ID reg val: 0x%x",
light_sen_id_reg_val);

            ssize_t num_sent_bytes = send(accept_conn_id,
light_sensor_rsp_msg, strlen(light_sensor_rsp_msg), 0);
            if (num_sent_bytes < 0)
                perror("send failed");

        }
        else if (!strcmp((((struct _socket_req_msg_struct_
*)&recv_buffer)->req_api_msg), "get_light_sensor_ctrl_reg"))
        {
            uint8_t light_sen_ctrl_reg_val = read_ctrl_reg();
            printf("ctrl reg val : %d\n", light_sen_ctrl_reg_val);

            memset(light_sensor_rsp_msg, '\0',
sizeof(light_sensor_rsp_msg));

            sprintf(light_sensor_rsp_msg, "Ctrl reg val: 0x%x",
light_sen_ctrl_reg_val);

            ssize_t num_sent_bytes = send(accept_conn_id,
light_sensor_rsp_msg, strlen(light_sensor_rsp_msg), 0);
            if (num_sent_bytes < 0)
                perror("send failed");
        }
        else if (!strcmp((((struct _socket_req_msg_struct_
*)&recv_buffer)->req_api_msg), "set_light_sensor_ctrl_reg"))
        {
            if (((struct _socket_req_msg_struct_ *)&recv_buffer)-
>ptr_param_list != NULL)
            {
                    #if 0
                         uint8_t cmd_ctrl_reg_val = *(uint8_t
*)(((struct _socket_req_msg_struct_ *)&recv_buffer)->ptr_param_list);
                    #endif
                    uint8_t cmd_ctrl_reg_val = (((struct
_socket_req_msg_struct_ *)&recv_buffer)->ptr_param_list);
                if (write_ctrl_reg(cmd_ctrl_reg_val) == 0)
                {
                    memset(light_sensor_rsp_msg, '\0',
sizeof(light_sensor_rsp_msg));
```

```
                    sprintf(light_sensor_rsp_msg, "OK");

                    ssize_t num_sent_bytes = send(accept_conn_id,
light_sensor_rsp_msg, strlen(light_sensor_rsp_msg), 0);
                    if (num_sent_bytes < 0)
                        perror("send failed");
                }
            }
        }
        else if (!strcmp((((struct _socket_req_msg_struct_
*)&recv_buffer)->req_api_msg), "get_light_sensor_tim_reg"))
        {
            uint8_t light_sen_tim_reg_val = read_timing_reg();
            printf("tim reg val : %d\n", light_sen_tim_reg_val);

            memset(light_sensor_rsp_msg, '\0',
sizeof(light_sensor_rsp_msg));

            sprintf(light_sensor_rsp_msg, "Ctrl reg val: 0x%x",
light_sen_tim_reg_val);

            ssize_t num_sent_bytes = send(accept_conn_id,
light_sensor_rsp_msg, strlen(light_sensor_rsp_msg), 0);
            if (num_sent_bytes < 0)
                perror("send failed");
        }
           #if 0
        else if (!strcmp((((struct _socket_req_msg_struct_
*)&recv_buffer)->req_api_msg), "set_light_sensor_tim_reg"))
        {
            if (((struct _socket_req_msg_struct_ *)&recv_buffer)-
>ptr_param_list != NULL)
            {
                struct _light_sensor_tim_params light_sen_tim_params =
*(struct _light_sensor_tim_params *)(((struct _socket_req_msg_struct_
*)&recv_buffer)->ptr_param_list);
                uint8_t cmd_tim_reg_val =
light_sen_tim_params.tim_reg_val;
                uint8_t cmd_tim_field_to_set =
light_sen_tim_params.tim_reg_field_to_set;
                uint8_t cmd_tim_field_val =
light_sen_tim_params.tim_reg_field_val;

                if (write_timing_reg(cmd_tim_reg_val,
cmd_tim_field_to_set, cmd_tim_field_val) == 0)
                {
                    memset(light_sensor_rsp_msg, '\0',
sizeof(light_sensor_rsp_msg));

                    sprintf(light_sensor_rsp_msg, "OK");
                    ssize_t num_sent_bytes = send(accept_conn_id,
light_sensor_rsp_msg, strlen(light_sensor_rsp_msg), 0);
                    if (num_sent_bytes < 0)
                        perror("send failed");
                    }
            }
        }
           #endif
           else if (!strcmp((((struct _socket_req_msg_struct_
*)&recv_buffer)->req_api_msg), "set_light_sensor_integration_time"))
```

```
        {
            if (((struct _socket_req_msg_struct_ *)&recv_buffer)-
>ptr_param_list != NULL)
            {

                uint8_t cmd_tim_reg_val = read_timing_reg();
                uint8_t cmd_tim_field_val = (((struct
_socket_req_msg_struct_ *)&recv_buffer)->ptr_param_list);

                if (write_timing_reg(cmd_tim_reg_val, 0x3,
cmd_tim_field_val) == 0)
                {
                    memset(light_sensor_rsp_msg, '\0',
sizeof(light_sensor_rsp_msg));

                    sprintf(light_sensor_rsp_msg, "OK");
                    ssize_t num_sent_bytes = send(accept_conn_id,
light_sensor_rsp_msg, strlen(light_sensor_rsp_msg), 0);
                    if (num_sent_bytes < 0)
                        perror("send failed");
                    }
            }
        }
           else if (!strcmp((((struct _socket_req_msg_struct_
*)&recv_buffer)->req_api_msg), "set_light_sensor_gain"))
        {
            if (((struct _socket_req_msg_struct_ *)&recv_buffer)-
>ptr_param_list != NULL)
            {

                uint8_t cmd_tim_reg_val = read_timing_reg();
                uint8_t cmd_tim_field_val = (((struct
_socket_req_msg_struct_ *)&recv_buffer)->ptr_param_list);

                if (write_timing_reg(cmd_tim_reg_val, 0x10,
cmd_tim_field_val) == 0)
                {
                    memset(light_sensor_rsp_msg, '\0',
sizeof(light_sensor_rsp_msg));

                    sprintf(light_sensor_rsp_msg, "OK");
                    ssize_t num_sent_bytes = send(accept_conn_id,
light_sensor_rsp_msg, strlen(light_sensor_rsp_msg), 0);
                    if (num_sent_bytes < 0)
                        perror("send failed");
                    }
            }
        }
           else if (!strcmp((((struct _socket_req_msg_struct_
*)&recv_buffer)->req_api_msg), "set_interrupt_low_threshold"))
        {
            if (((struct _socket_req_msg_struct_ *)&recv_buffer)-
>ptr_param_list != NULL)
            {


                uint16_t low_thresh = (((struct _socket_req_msg_struct_
*)&recv_buffer)->ptr_param_list);

                write_intr_low_thresh_reg(low_thresh);
```

```
                sprintf(light_sensor_rsp_msg, "OK");
                ssize_t num_sent_bytes = send(accept_conn_id,
light_sensor_rsp_msg, strlen(light_sensor_rsp_msg), 0);
                if (num_sent_bytes < 0)
                    perror("send failed");
            }
        }
           else if (!strcmp((((struct _socket_req_msg_struct_
*)&recv_buffer)->req_api_msg), "set_interrupt_high_threshold"))
        {
            if (((struct _socket_req_msg_struct_ *)&recv_buffer)-
>ptr_param_list != NULL)
            {


                uint16_t high_thresh = (((struct _socket_req_msg_struct_
*)&recv_buffer)->ptr_param_list);

                write_intr_high_thresh_reg(high_thresh);

                sprintf(light_sensor_rsp_msg, "OK");
                ssize_t num_sent_bytes = send(accept_conn_id,
light_sensor_rsp_msg, strlen(light_sensor_rsp_msg), 0);
                if (num_sent_bytes < 0)
                    perror("send failed");
            }
        }

        else if (!strcmp((((struct _socket_req_msg_struct_
*)&recv_buffer)->req_api_msg), "get_light_sensor_int_thresh_reg"))
        {
#if 0
            uint8_t cmd_thresh_low_low_reg =
I2C_LIGHT_SENSOR_CMD_THRESH_LOW_LOW_REG;
            uint8_t cmd_thresh_low_high_reg =
I2C_LIGHT_SENSOR_CMD_THRESH_LOW_HIGH_REG;
            uint8_t cmd_thresh_high_low_reg =
I2C_LIGHT_SENSOR_CMD_THRESH_HIGH_LOW_REG;
            uint8_t cmd_thresh_high_high_reg =
I2C_LIGHT_SENSOR_CMD_THRESH_HIGH_HIGH_REG;

            int8_t light_sen_thresh_low_low_reg_val =
read_light_sensor_reg(cmd_thresh_low_low_reg);
            printf("thresh low low reg val : %d\n",
light_sen_thresh_low_low_reg_val);

            int8_t light_sen_thresh_low_high_reg_val =
read_light_sensor_reg(cmd_thresh_low_high_reg);
            printf("thresh low high reg val : %d\n",
light_sen_thresh_low_high_reg_val);

            int8_t light_sen_thresh_high_low_reg_val =
read_light_sensor_reg(cmd_thresh_high_low_reg);
            printf("thresh high low reg val : %d\n",
light_sen_thresh_high_low_reg_val);

            int8_t light_sen_thresh_high_high_reg_val =
read_light_sensor_reg(cmd_thresh_high_high_reg);
```

```
            printf("thresh high high reg val : %d\n",
light_sen_thresh_high_high_reg_val);

            memset(light_sensor_rsp_msg, '\0',
sizeof(light_sensor_rsp_msg));

            struct _int_thresh_reg_struct_ int_thresh_reg_struct;
            int_thresh_reg_struct.thresh_low_low =
light_sen_thresh_low_low_reg_val;
            int_thresh_reg_struct.thresh_low_high =
light_sen_thresh_low_high_reg_val;
            int_thresh_reg_struct.thresh_high_low =
light_sen_thresh_high_low_reg_val;
            int_thresh_reg_struct.thresh_high_high =
light_sen_thresh_high_high_reg_val;
#endif
            uint16_t low_thresh, high_thresh;
            read_intr_thresh_reg(&low_thresh, &high_thresh);

            struct _int_thresh_reg_struct_ int_thresh_reg_struct;
            int_thresh_reg_struct.low_thresh = low_thresh;
            int_thresh_reg_struct.high_thresh = high_thresh;
            ssize_t num_sent_bytes = send(accept_conn_id,
&int_thresh_reg_struct,
                                          sizeof(struct
_int_thresh_reg_struct_), 0);
            if (num_sent_bytes < 0)
                perror("send failed");
        }
#if 0
        else if (!strcmp((((struct _socket_req_msg_struct_
*)&recv_buffer)->req_api_msg), "set_light_sensor_int_thresh_reg"))
        {
            if (((struct _socket_req_msg_struct_ *)&recv_buffer)-
>ptr_param_list != NULL)
            {
                struct _int_thresh_reg_struct_ *p_int_thresh_reg_struct =
                    (struct _int_thresh_reg_struct_ *)(((struct
_socket_req_msg_struct_ *)&recv_buffer)->ptr_param_list);
      #if 0
                uint8_t cmd_thresh_low_low_reg =
I2C_LIGHT_SENSOR_CMD_THRESH_LOW_LOW_REG;
                uint8_t cmd_thresh_low_low_reg_val =
(uint8_t)p_int_thresh_reg_struct->thresh_low_low;
                write_light_sensor_reg(cmd_thresh_low_low_reg,
cmd_thresh_low_low_reg_val);

                uint8_t cmd_thresh_low_high_reg =
I2C_LIGHT_SENSOR_CMD_THRESH_LOW_HIGH_REG;
                uint8_t cmd_thresh_low_high_reg_val =
(uint8_t)p_int_thresh_reg_struct->thresh_low_high;
                write_light_sensor_reg(cmd_thresh_low_high_reg,
cmd_thresh_low_high_reg_val);

                uint8_t cmd_thresh_high_low_reg =
I2C_LIGHT_SENSOR_CMD_THRESH_HIGH_LOW_REG;
                uint8_t cmd_thresh_high_low_reg_val =
(uint8_t)p_int_thresh_reg_struct->thresh_high_low;
                write_light_sensor_reg(cmd_thresh_high_low_reg,
cmd_thresh_high_low_reg_val);
```

```
                uint8_t cmd_thresh_high_high_reg =
I2C_LIGHT_SENSOR_CMD_THRESH_HIGH_HIGH_REG;
                uint8_t cmd_thresh_high_high_reg_val =
(uint8_t)p_int_thresh_reg_struct->thresh_high_high;
                write_light_sensor_reg(cmd_thresh_high_high_reg,
cmd_thresh_high_high_reg_val);
      #endif

                uint16_t low_thresh = p_int_thresh_reg_struct-
>low_thresh;
                uint16_t high_thresh = p_int_thresh_reg_struct-
>high_thresh;

                write_intr_thresh_reg(low_thresh, high_thresh);

                sprintf(light_sensor_rsp_msg, "OK");
                ssize_t num_sent_bytes = send(accept_conn_id,
light_sensor_rsp_msg, strlen(light_sensor_rsp_msg), 0);
                if (num_sent_bytes < 0)
                    perror("send failed");
            }
        }
#endif
        else
        {
            printf("Invalid request from socket task\n");
        }
    }

    pthread_exit(NULL);
}

void *sensor_thread_func(void *arg)
{
    while (!g_sig_kill_sensor_thread)
    {
        float sensor_lux_data = get_lux_data();

        printf("Sensor lux data: %3.2f\n", sensor_lux_data);

        log_lux_data(sensor_lux_data);

        sleep(5);
    }

    pthread_exit(NULL);
}

void init_sock(int *sock_fd, struct sockaddr_in *server_addr_struct,
               int port_num, int listen_qsize)
{
    int serv_addr_len = sizeof(struct sockaddr_in);

    /* Create the socket */
    if ((*sock_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0)
    {
        perror("socket creation failed");
        pthread_exit(NULL); // Change these return values from
pthread_exit
```

```
    }

    int option = 1;
    if(setsockopt(*sock_fd, SOL_SOCKET, (SO_REUSEPORT | SO_REUSEADDR),
(void *)&option, sizeof(option)) < 0)
    {
        perror("setsockopt failed");
        pthread_exit(NULL);
    }

    server_addr_struct->sin_family = AF_INET;
    server_addr_struct->sin_addr.s_addr = INADDR_ANY;
    server_addr_struct->sin_port = htons(port_num);

    if (bind(*sock_fd, (struct sockaddr *)server_addr_struct,
                                            sizeof(struct
sockaddr_in))<0)
    {
        perror("bind failed");
        pthread_exit(NULL);
    }

    if (listen(*sock_fd, listen_qsize) < 0)
    {
        perror("listen failed");
        pthread_exit(NULL);
    }

}


void *socket_hb_thread_func(void *arg)
{
    int sock_hb_fd;
    struct sockaddr_in sock_hb_address;
    int sock_hb_addr_len = sizeof(sock_hb_address);

    init_sock(&sock_hb_fd, &sock_hb_address, SOCKET_HB_PORT_NUM,
SOCKET_HB_LISTEN_QUEUE_SIZE);


    int accept_conn_id;
    printf("Waiting for request...\n");
    if ((accept_conn_id = accept(sock_hb_fd, (struct sockaddr
*)&sock_hb_address,
                    (socklen_t*)&sock_hb_addr_len)) < 0)
    {
        perror("accept failed");
        //pthread_exit(NULL);
    }

    char recv_buffer[MSG_BUFF_MAX_LEN];
    char send_buffer[] = "Alive";

    while (!g_sig_kill_sock_hb_thread)
    {
        memset(recv_buffer, '\0', sizeof(recv_buffer));

        size_t num_read_bytes = read(accept_conn_id, &recv_buffer,
sizeof(recv_buffer));
```

```
        if (!strcmp(recv_buffer, "heartbeat"))
        {
                 ssize_t num_sent_bytes = send(accept_conn_id,
send_buffer, strlen(send_buffer), 0);
            if (num_sent_bytes < 0)
                perror("send failed");
        }
    }

    pthread_exit(NULL);
}

float get_lux_data(void)
{
    float sensor_lux_val = 0;

    uint16_t adc_ch0_data, adc_ch1_data;

    get_adc_channel_data(0, &adc_ch0_data);
    get_adc_channel_data(1, &adc_ch1_data);

    sensor_lux_val = calculate_lux_value(adc_ch0_data, adc_ch1_data);

    printf("Sensor lux value: %3.2f\n", sensor_lux_val);

    return sensor_lux_val;
}

void get_adc_channel_data(int channel_num, uint16_t *ch_data)
{
    if (channel_num == 0)
    {
        uint8_t cmd_data0_low_reg = I2C_LIGHT_SENSOR_CMD_DATA0LOW_REG;
        uint8_t cmd_data0_high_reg = I2C_LIGHT_SENSOR_CMD_DATA0HIGH_REG;

        int8_t ch_data_low = read_light_sensor_reg(cmd_data0_low_reg);
        //printf("data0_low : %d\n", ch_data_low);

        int8_t ch_data_high = read_light_sensor_reg(cmd_data0_high_reg);
        //printf("data0_high : %d\n", ch_data_high);

        *ch_data = ch_data_high << 8 | ch_data_low;
    }
    else if (channel_num == 1)
    {
        uint8_t cmd_data1_low_reg = I2C_LIGHT_SENSOR_CMD_DATA1LOW_REG;
        uint8_t cmd_data1_high_reg = I2C_LIGHT_SENSOR_CMD_DATA1HIGH_REG;

        int8_t ch_data_low = read_light_sensor_reg(cmd_data1_low_reg);
        //printf("data1_low : %d\n", ch_data_low);

        int8_t ch_data_high = read_light_sensor_reg(cmd_data1_high_reg);
        //printf("data1_high : %d\n", ch_data_high);

        *ch_data = ch_data_high << 8 | ch_data_low;
    }
    else
    {
        printf("Channel number %d invalid\n", channel_num);
```

```
    }
}

float calculate_lux_value(uint16_t ch0_data, uint16_t ch1_data)
{
    float sensor_lux_val = 0;

    if (ch0_data == 0 || ch1_data == 0)
        return 0;

    /* Mapping between ADC channel data and the sensor lux formula used
    **       CH1/CH0                                  Sensor lux formula
    **
    **  0 < CH1/CH0 ≤ 0.50           Sensor Lux = (0.0304 x CH0) - (0.062
x CH0 x ((CH1/CH0)^1.4))
    **  0.50 < CH1/CH0 ≤ 0.61        Sensor Lux = (0.0224 x CH0) - (0.031
x CH1)
    **  0.61 < CH1/CH0 ≤ 0.80        Sensor Lux = (0.0128 x CH0) -
(0.0153 x CH1)
    **  0.80 < CH1/CH0 ≤ 1.30        Sensor Lux = (0.00146 x CH0) -
(0.00112 x CH1)
    **  CH1/CH0>1.30                 Sensor Lux = 0
    **
    */

    float adc_count_ratio = (float)(ch1_data/ch0_data);
    if ( 0 < adc_count_ratio <= 0.5)
    {
        sensor_lux_val = ((0.0304 * ch0_data) - (0.062 * ch0_data *
pow(adc_count_ratio, 1.4)));
    }
    else if (0.5 < adc_count_ratio <= 0.61)
    {
        sensor_lux_val = ((0.0224 * ch0_data) - (0.031 * ch1_data));
    }
    else if (0.61 < adc_count_ratio <= 0.8)
    {
        sensor_lux_val = ((0.0128 * ch0_data) - (0.0153 * ch1_data));
    }
    else if (0.8 < adc_count_ratio <= 1.3)
    {
        sensor_lux_val = ((0.00146 * ch0_data) - (0.00112 * ch1_data));
    }
    else if (adc_count_ratio > 1.3)
    {
        sensor_lux_val = 0;
    }

    return sensor_lux_val;
}

int write_light_sensor_reg(int cmd_reg_val, int target_reg_val)
{

    /* Write the command register to specify the following two
information
    **       1. Target register address for subsequent write operation
    **       2. If I2C write operation is a word or byte operation
    */
      if (wrapper_write(i2c_light_sensor_fd, &cmd_reg_val, 1) != 1)
```

```
      {
            perror("Failed to write to the i2c bus.");
        return -1;
    }

    if(wrapper_write(i2c_light_sensor_fd, &target_reg_val, 1) != 1){
            perror("Failed to write to the i2c bus.");
        return -1;
    }

    return 0;
}

int8_t read_light_sensor_reg(uint8_t read_reg_val)
{
    /* Write the read register to specify the initiate a read operation
*/
      if(wrapper_write(i2c_light_sensor_fd, &read_reg_val, 1) != 1){
            printf("Failed to write to the i2c bus.\n");
        return -1;
    }

    /* Read the value */
    int read_val;
      if (wrapper_read(i2c_light_sensor_fd, &read_val, 1) != 1) {
            perror("adc data read error");
        return -1;
      }
    //printf("**** read val for %d: %d\n", read_reg_val, read_val);

    int8_t ret_val = (int8_t)read_val;

    return ret_val;
}

void log_lux_data(float lux_data)
{
    int msg_priority;

    /* Set the message queue attributes */
    struct mq_attr logger_mq_attr = { .mq_flags = 0,
                                      .mq_maxmsg =
MSG_QUEUE_MAX_NUM_MSGS,  // Max number of messages on queue
                                      .mq_msgsize =
MSG_QUEUE_MAX_MSG_SIZE  // Max. message size
                                    };

    logger_mq_handle = mq_open(MSG_QUEUE_NAME, O_RDWR, S_IRWXU,
&logger_mq_attr);

    char lux_data_msg[128];
    memset(lux_data_msg, '\0', sizeof(lux_data_msg));

    sprintf(lux_data_msg, "Lux Value: %3.2f", lux_data);

    struct _logger_msg_struct_ logger_msg;
    memset(&logger_msg, '\0', sizeof(logger_msg));
    strcpy(logger_msg.message, lux_data_msg);
    strcpy(logger_msg.logger_msg_src_id, "Light");
    logger_msg.logger_msg_src_id[strlen("Light") + 1] = '\0';
```

```
    strncpy(logger_msg.logger_msg_level, "Info", strlen("Info"));
    logger_msg.logger_msg_level[strlen("Info") + 1] = '\0';

    msg_priority = 1;
    int num_sent_bytes = mq_send(logger_mq_handle, (char *)&logger_msg,
                            sizeof(logger_msg), msg_priority);
    if (num_sent_bytes < 0)
        perror("mq_send failed");
}

void sig_handler(int sig_num)
{
    char buffer[MSG_BUFF_MAX_LEN];
    memset(buffer, '\0', sizeof(buffer));

    if (sig_num == SIGINT || sig_num == SIGUSR1)
    {
        if (sig_num == SIGINT)
            printf("Caught signal %s in light task\n", "SIGINT");
        else if (sig_num == SIGUSR1)
            printf("Caught signal %s in light task\n", "SIGKILL");

        g_sig_kill_sensor_thread = 1;
        g_sig_kill_sock_thread = 1;
        g_sig_kill_sock_hb_thread = 1;

        //pthread_join(sensor_thread_id, NULL);
        //pthread_join(socket_thread_id, NULL);
        //pthread_join(socket_hb_thread_id, NULL);

        mq_close(logger_mq_handle);

        if (i2c_light_sensor_fd != -1)
            close(i2c_light_sensor_fd);

        exit(0);
    }
}

void write_cmd_reg(uint8_t cmd_reg_val)
{

}

uint8_t read_ctrl_reg(void)
{
    uint8_t cmd_ctrl_reg = I2C_LIGHT_SENSOR_CMD_CTRL_REG;

    int8_t light_sen_ctrl_reg_val = read_light_sensor_reg(cmd_ctrl_reg);
    if (light_sen_ctrl_reg_val != -1)
        return (uint8_t)light_sen_ctrl_reg_val;
    else
        return 0xFF; /* Sending 0xFF in case of error */

}

int write_ctrl_reg(uint8_t ctrl_reg_val)
{
    uint8_t cmd_ctrl_reg = I2C_LIGHT_SENSOR_CMD_CTRL_REG;
```

```
    if (write_light_sensor_reg(cmd_ctrl_reg, ctrl_reg_val) == 0)
    {
        return 0;
    }
    else
    {
        return -1;
    }
}

uint8_t read_timing_reg(void)
{
    uint8_t cmd_tim_reg = I2C_LIGHT_SENSOR_CMD_TIM_REG;

    int8_t light_sen_tim_reg_val = read_light_sensor_reg(cmd_tim_reg);
    if (light_sen_tim_reg_val != -1)
        return (uint8_t)light_sen_tim_reg_val;
    else
        return 0xFF; /* Sending 0xFF in case of error */

}

int write_timing_reg(uint8_t tim_reg_val, uint8_t field_to_set, uint8_t
field_val)
{
    uint8_t cmd_tim_reg = I2C_LIGHT_SENSOR_CMD_TIM_REG;
    int ret_val = -1;

      uint8_t time_reg_val_copy = tim_reg_val;
      time_reg_val_copy &= ~field_to_set;
      time_reg_val_copy |= (field_val << 4);

      if (write_light_sensor_reg(cmd_tim_reg, time_reg_val_copy) == 0)
      {
            ret_val = 0;
      }
      else
      {
            ret_val = -1;
      }
      return ret_val;

      #if 0
    if (field_to_set & 0x3 == 0x3)
    {
        /* Setting integration time */
        uint8_t time_reg_val_copy = tim_reg_val;
        time_reg_val_copy &= ~0x3;
        time_reg_val_copy |= field_val;

        if (write_light_sensor_reg(cmd_tim_reg, time_reg_val_copy) == 0)
        {
            ret_val = 0;
        }
        else
        {
            ret_val = -1;
        }
        return ret_val;
    }
```

```
    if (field_to_set & 0x10 == 0x10)
    {
        /* Setting integration gain */
        uint8_t time_reg_val_copy = tim_reg_val;
        time_reg_val_copy &= ~0x10;
        time_reg_val_copy |= (field_val << 4);

        if (write_light_sensor_reg(cmd_tim_reg, time_reg_val_copy) == 0)
        {
            ret_val = 0;
        }
        else
        {
            ret_val = -1;
        }
        return ret_val;

    }
      #endif

}

int enable_disable_intr_ctrl_reg(uint8_t int_ctrl_reg_val)
{
    uint8_t cmd_intr_ctrl_reg = I2C_LIGHT_SENSOR_CMD_INT_REG;

    if (write_light_sensor_reg(cmd_intr_ctrl_reg, int_ctrl_reg_val) == 0)
    {
        return 0;
    }
    else
    {
        return -1;
    }

}

uint8_t read_id_reg(void)
{
    uint8_t cmd_id_reg = I2C_LIGHT_SENSOR_CMD_ID_REG;

    int8_t light_sen_id_reg_val = read_light_sensor_reg(cmd_id_reg);
    if (light_sen_id_reg_val != -1)
        return (uint8_t)light_sen_id_reg_val;
    else
        return 0xFF; /* Sending 0xFF in case of error */

}

void read_intr_thresh_reg(uint16_t *low_thresh, uint16_t *high_thresh)
{
    uint8_t cmd_thresh_low_low_reg =
I2C_LIGHT_SENSOR_CMD_THRESH_LOW_LOW_REG;
    uint8_t cmd_thresh_low_high_reg =
I2C_LIGHT_SENSOR_CMD_THRESH_LOW_HIGH_REG;
    uint8_t cmd_thresh_high_low_reg =
I2C_LIGHT_SENSOR_CMD_THRESH_HIGH_LOW_REG;
    uint8_t cmd_thresh_high_high_reg =
I2C_LIGHT_SENSOR_CMD_THRESH_HIGH_HIGH_REG;
```

```
    int8_t light_sen_thresh_low_low_reg_val =
read_light_sensor_reg(cmd_thresh_low_low_reg);
    printf("thresh low low reg val : %d\n",
light_sen_thresh_low_low_reg_val);

    int8_t light_sen_thresh_low_high_reg_val =
read_light_sensor_reg(cmd_thresh_low_high_reg);
    printf("thresh low high reg val : %d\n",
light_sen_thresh_low_high_reg_val);

    int8_t light_sen_thresh_high_low_reg_val =
read_light_sensor_reg(cmd_thresh_high_low_reg);
    printf("thresh high low reg val : %d\n",
light_sen_thresh_high_low_reg_val);

    int8_t light_sen_thresh_high_high_reg_val =
read_light_sensor_reg(cmd_thresh_high_high_reg);
    printf("thresh high high reg val : %d\n",
light_sen_thresh_high_high_reg_val);

    *low_thresh = (light_sen_thresh_low_high_reg_val << 8 |
light_sen_thresh_low_low_reg_val);
    *high_thresh = (light_sen_thresh_high_high_reg_val << 8 |
light_sen_thresh_high_low_reg_val);
}

void read_intr_low_thresh_reg(uint16_t *low_thresh)
{
      uint8_t cmd_thresh_low_low_reg =
I2C_LIGHT_SENSOR_CMD_THRESH_LOW_LOW_REG;
    uint8_t cmd_thresh_low_high_reg =
I2C_LIGHT_SENSOR_CMD_THRESH_LOW_HIGH_REG;


    int8_t light_sen_thresh_low_low_reg_val =
read_light_sensor_reg(cmd_thresh_low_low_reg);
    printf("thresh low low reg val : %d\n",
light_sen_thresh_low_low_reg_val);

    int8_t light_sen_thresh_low_high_reg_val =
read_light_sensor_reg(cmd_thresh_low_high_reg);
    printf("thresh low high reg val : %d\n",
light_sen_thresh_low_high_reg_val);

    *low_thresh = (light_sen_thresh_low_high_reg_val << 8 |
light_sen_thresh_low_low_reg_val);

}

void read_intr_high_thresh_reg(uint16_t *high_thresh)
{
    uint8_t cmd_thresh_high_low_reg =
I2C_LIGHT_SENSOR_CMD_THRESH_HIGH_LOW_REG;
    uint8_t cmd_thresh_high_high_reg =
I2C_LIGHT_SENSOR_CMD_THRESH_HIGH_HIGH_REG;

    int8_t light_sen_thresh_high_low_reg_val =
read_light_sensor_reg(cmd_thresh_high_low_reg);
    printf("thresh high low reg val : %d\n",
light_sen_thresh_high_low_reg_val);
```

```
    int8_t light_sen_thresh_high_high_reg_val =
read_light_sensor_reg(cmd_thresh_high_high_reg);
    printf("thresh high high reg val : %d\n",
light_sen_thresh_high_high_reg_val);

    *high_thresh = (light_sen_thresh_high_high_reg_val << 8 |
light_sen_thresh_high_low_reg_val);

}

void write_intr_high_thresh_reg(uint16_t high_thresh)
{
      uint8_t cmd_thresh_high_low_reg =
I2C_LIGHT_SENSOR_CMD_THRESH_HIGH_LOW_REG;
    uint8_t cmd_thresh_high_low_reg_val = (uint8_t)high_thresh & 0xFF;
    write_light_sensor_reg(cmd_thresh_high_low_reg,
cmd_thresh_high_low_reg_val);

    uint8_t cmd_thresh_high_high_reg =
I2C_LIGHT_SENSOR_CMD_THRESH_HIGH_HIGH_REG;
    uint8_t cmd_thresh_high_high_reg_val = (uint8_t)((high_thresh >> 8) &
0xFF);
    write_light_sensor_reg(cmd_thresh_high_high_reg,
cmd_thresh_high_high_reg_val);
}

void write_intr_low_thresh_reg(uint16_t low_thresh)
{
      uint8_t cmd_thresh_low_low_reg =
I2C_LIGHT_SENSOR_CMD_THRESH_LOW_LOW_REG;
    uint8_t cmd_thresh_low_low_reg_val = (uint8_t)low_thresh & 0xFF;
    write_light_sensor_reg(cmd_thresh_low_low_reg,
cmd_thresh_low_low_reg_val);

    uint8_t cmd_thresh_low_high_reg =
I2C_LIGHT_SENSOR_CMD_THRESH_LOW_HIGH_REG;
    uint8_t cmd_thresh_low_high_reg_val = (uint8_t)((low_thresh >> 8) &
0xFF);
    write_light_sensor_reg(cmd_thresh_low_high_reg,
cmd_thresh_low_high_reg_val);
}

#if 0
void write_intr_thresh_reg(uint16_t low_thresh, uint16_t high_thresh)
{
    uint8_t cmd_thresh_low_low_reg =
I2C_LIGHT_SENSOR_CMD_THRESH_LOW_LOW_REG;
    uint8_t cmd_thresh_low_low_reg_val = (uint8_t)low_thresh & 0xFF;
    write_light_sensor_reg(cmd_thresh_low_low_reg,
cmd_thresh_low_low_reg_val);

    uint8_t cmd_thresh_low_high_reg =
I2C_LIGHT_SENSOR_CMD_THRESH_LOW_HIGH_REG;
    uint8_t cmd_thresh_low_high_reg_val = (uint8_t)((low_thresh >> 8) &
0xFF);
    write_light_sensor_reg(cmd_thresh_low_high_reg,
cmd_thresh_low_high_reg_val);
```

```
    uint8_t cmd_thresh_high_low_reg =
I2C_LIGHT_SENSOR_CMD_THRESH_HIGH_LOW_REG;
    uint8_t cmd_thresh_high_low_reg_val = (uint8_t)high_thresh & 0xFF;
    write_light_sensor_reg(cmd_thresh_high_low_reg,
cmd_thresh_high_low_reg_val);

    uint8_t cmd_thresh_high_high_reg =
I2C_LIGHT_SENSOR_CMD_THRESH_HIGH_HIGH_REG;
    uint8_t cmd_thresh_high_high_reg_val = (uint8_t)((high_thresh >> 8) &
0xFF);
    write_light_sensor_reg(cmd_thresh_high_high_reg,
cmd_thresh_high_high_reg_val);

}
#endif

void light_sensor_exit(void)
{
    /* Close i2c bus */
    if (i2c_light_sensor_fd != -1)
        close(i2c_light_sensor_fd);
}
/**************************************************************************
******
* Author:       Pavan Dhareshwar & Sridhar Pavithrapu
* Date:         03/07/2018
* File:         wrapper.c
* Description:  Source file describing the functionality and
implementation
*               of wrapper for synchronization of light and temperature
tasks.
**************************************************************************
*****/

/*---------------------------------- INCLUDES ---------------------------
----*/
#include "wrapper.h"

sem_t *get_named_semaphore_handle(void)
{
    sem_t *sem;
    if ((sem = sem_open("wrapper_sem", O_CREAT, 0644, 1)) == SEM_FAILED)
    {
        perror("sem_open failed");
        return SEM_FAILED;
    }
    else
    {
        printf("Named semaphore created successfully\n");
        return sem;
    }
}

ssize_t wrapper_write(int fd, void *buf, size_t count){

      ssize_t return_value = 0;

#if 1
    sem_t *wrapper_sem = get_named_semaphore_handle();
    if (wrapper_sem == SEM_FAILED)
```

```
    {
        return -1000;
    }

     if(sem_wait(wrapper_sem) == 0)
    {
            return_value = write(fd, buf, count);
    }
     else{
           perror("sem_wait error in wrapper\n");
     }

     if(sem_post(wrapper_sem) != 0){

           perror("sem_post error in wrapper\n");
     }
#else
    return_value = write(fd, buf, count);
#endif
     return return_value;
}

ssize_t wrapper_read(int fd, void *buf, size_t count){

     ssize_t return_value = 0;

#if 1
    sem_t *wrapper_sem = get_named_semaphore_handle();
    if (wrapper_sem == SEM_FAILED)
    {
        return -1000;
    }

    if(sem_wait(wrapper_sem) == 0){

            return_value = read(fd, buf, count);
    }
     else{
           perror("sem_wait error in wrapper\n");
     }

     if(sem_post(wrapper_sem) != 0){

           perror("sem_post error in wrapper\n");
     }
#else
    return_value = read(fd, buf, count);
#endif

     return return_value;
}
#ifndef _WRAPPER_H_
#define _WRAPPER_H_

#include <semaphore.h>
#include <errno.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
```

```
#include <stdint.h>
#include <linux/i2c-dev.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

sem_t *get_named_semaphore_handle(void);

ssize_t wrapper_write(int fd, void *buf, size_t count);
ssize_t wrapper_read(int fd, void *buf, size_t count);

#endif
/**************************************************************************
*************
*
* FileName        :    temp_unit_tests.c
* Description     :    This file contains necessary  test functions for
temperature task.

* File Author Name:    Sridhar Pavithrapu
* Tools used      :    gcc, gedit, cmocka
* References      :    None
*
**************************************************************************
**************/

/* Headers Section */
#include <math.h>
#include <stdlib.h>
#include <stdarg.h>
#include <stddef.h>
#include <setjmp.h>
#include <cmocka.h>
#include "light_sensor.h"

/* Macros section */


/**
*   @brief :  test function for checking gain of light sensor
*
*
*   @param  state   A pointer to the state
*
*   @return None
*/
void test_light_gain_check(void **state){

      /* Test case for checking gain of light sensor */
      uint8_t cmd_tim_reg_val = read_timing_reg();
      uint8_t cmd_tim_field_val = 0XAF;

      write_timing_reg(cmd_tim_reg_val, 0x10, cmd_tim_field_val);

      uint8_t light_sen_tim_reg_val = read_timing_reg();
      assert_int_equal(light_sen_tim_reg_val, cmd_tim_field_val);

}
```

```
/**
*   @brief :  test function for checking integration time of light sensor
*
*
*   @param  state   A pointer to the state
*
*   @return None
*/
void test_light_integration_time_check(void **state){

      /* Test case for checking integration time of light sensor */
      uint8_t cmd_tim_reg_val = read_timing_reg();
      uint8_t cmd_tim_field_val = 0X1F;

      write_timing_reg(cmd_tim_reg_val, 0x3, cmd_tim_field_val);

      uint8_t light_sen_tim_reg_val = read_timing_reg();
      assert_int_equal(light_sen_tim_reg_val, cmd_tim_field_val);
}

/**
*   @brief :  test function for checking low threshold of light sensor
*
*
*   @param  state   A pointer to the state
*
*   @return None
*/
void test_light_low_threshold_check(void **state){

      /* Test case for checking low threshold of light sensor */
      uint16_t data = 0X9876;
      write_intr_low_thresh_reg(data);
      uint16_t *return_value;
      read_temp_high_low_register(return_value);
      assert_int_equal(return_value, data);
}

/**
*   @brief : test function for checking high threshold of light sensor
*
*
*   @param  state   A pointer to the state
*
*   @return None
*/
void test_light_high_threshold_check(void **state){

      /* Test case for checking temperature high threshold of light
sensor */
      uint16_t data = 0X7676;
      write_intr_high_thresh_reg(data);
      uint16_t *return_value;
      read_temp_high_high_register(return_value);
      assert_int_equal(return_value, data);

}


/**
```

```
*   @brief : main function for all DLL test cases
*
*
*   @return Pass and Fail test cases
*/
int main(int argc, char **argv){

      /* Calling all DLL test case functions */
      const struct CMUnitTest tests[] = {
            cmocka_unit_test(test_light_gain_check),
            cmocka_unit_test(test_light_integration_time_check),
            cmocka_unit_test(test_light_low_threshold_check),
            cmocka_unit_test(test_light_high_threshold_check),

      };

      return cmocka_run_group_tests(tests, NULL, NULL);
}/*************************************************************************
**
* Author:       Pavan Dhareshwar & Sridhar Pavithrapu
* Date:         03/07/2018
* File:         light_sensor.h
* Description:  Header file containing the macros, structs/enums, globals
                and function prototypes for source file light_sensor.c
*************************************************************************
/

#ifndef _LIGHT_SENSOR_TASK_H_
#define _LIGHT_SENSOR_TASK_H_

/*---------------------------------- INCLUDES ----------------------------
----*/
#include <errno.h>
#include <stdint.h>
#include <string.h>
#include <math.h>

#include <unistd.h>
#include <fcntl.h>
#include <signal.h>

#include <linux/i2c-dev.h>

#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/msg.h>
#include <sys/ipc.h>

#include <mqueue.h>

#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/socket.h>

#include "wrapper.h"

/*---------------------------------- INCLUDES ----------------------------
----*/
```

```
/*---------------------------------- MACROS ----------------------------
----*/

#define I2C_SLAVE_ADDR                0b0111001       // Slave address -
0x39
#define I2C_DEV_NAME                  "/dev/i2c-2"

#define I2C_LIGHT_SENSOR_CMD_CTRL_REG                0x80
#define I2C_LIGHT_SENSOR_CMD_TIM_REG                 0x81

#define I2C_LIGHT_SENSOR_CMD_THRESH_LOW_LOW_REG      0x82
#define I2C_LIGHT_SENSOR_CMD_THRESH_LOW_HIGH_REG     0x83
#define I2C_LIGHT_SENSOR_CMD_THRESH_HIGH_LOW_REG     0x84
#define I2C_LIGHT_SENSOR_CMD_THRESH_HIGH_HIGH_REG    0x85

#define I2C_LIGHT_SENSOR_CMD_INT_REG                 0x86
#define I2C_LIGHT_SENSOR_CMD_ID_REG                  0x8A

#define I2C_LIGHT_SENSOR_CMD_DATA0LOW_REG            0x8C
#define I2C_LIGHT_SENSOR_CMD_DATA0HIGH_REG           0x8D
#define I2C_LIGHT_SENSOR_CMD_DATA1LOW_REG            0x8E
#define I2C_LIGHT_SENSOR_CMD_DATA1HIGH_REG           0x8F

#define I2C_LIGHT_SENSOR_CTRL_REG_VAL                0x3

#define MSG_QUEUE_NAME                               "/logger_task_mq"
#define MSG_QUEUE_MAX_NUM_MSGS                       5
#define MSG_QUEUE_MAX_MSG_SIZE                       1024

#define MSG_MAX_LEN                                  128

#define LIGHT_SENSOR_SERVER_PORT_NUM                 8086
#define LIGHT_SENSOR_LISTEN_QUEUE_SIZE               5

#define MSG_BUFF_MAX_LEN                             1024

#define SOCK_REQ_MSG_API_MSG_LEN                     64

#define SOCKET_HB_PORT_NUM                           8660
#define SOCKET_HB_LISTEN_QUEUE_SIZE                  5

#define MSG_TYPE_TEMP_DATA                           0
#define MSG_TYPE_LUX_DATA                            1
#define MSG_TYPE_SOCK_DATA                           2
#define MSG_TYPE_MAIN_DATA                           3

#define LOGGER_ATTR_LEN                              32

/*---------------------------------- MACROS ----------------------------
----*/

/*---------------------------------- GLOBALS ---------------------------
----*/
int i2c_light_sensor_fd;
int server_fd, accept_conn_id;
int sensor_thread_id, socket_thread_id, socket_hb_thread_id;

mqd_t logger_mq_handle;
```

```
sig_atomic_t g_sig_kill_sensor_thread, g_sig_kill_sock_thread,
g_sig_kill_sock_hb_thread;
/*---------------------------------- GLOBALS ----------------------------
----*/

/*---------------------------- STRUCTURES/ENUMERATIONS ------------------
----*/

struct _logger_msg_struct_
{
    char message[MSG_MAX_LEN];
    char logger_msg_src_id[LOGGER_ATTR_LEN];
    char logger_msg_level[LOGGER_ATTR_LEN];
};

enum _req_recipient_
{
    REQ_RECP_TEMP_TASK,
    REQ_RECP_LIGHT_TASK
};

struct _socket_req_msg_struct_
{
    char req_api_msg[SOCK_REQ_MSG_API_MSG_LEN];
    enum _req_recipient_ req_recipient;
    void *ptr_param_list;
};

#if 0
struct _int_thresh_reg_struct_
{
    uint8_t thresh_low_low;
    uint8_t thresh_low_high;
    uint8_t thresh_high_low;
    uint8_t thresh_high_high;
};
#endif
struct _int_thresh_reg_struct_
{
    uint16_t low_thresh;
    uint16_t high_thresh;
};

struct _light_sensor_tim_params
{
    uint8_t tim_reg_val;
    uint8_t tim_reg_field_to_set;
    uint8_t tim_reg_field_val;

};
/*---------------------------- STRUCTURES/ENUMERATIONS ------------------
----*/

/*---------------------------- FUNCTION PROTOTYPES ----------------------
----*/
/**
 *  @brief Initialize the light sensor
 *
 *  This function will open the i2c bus for read and write operation and
 *  initialize the communication with the peripheral.
```

```
 *
 *  @param void
 *
 *  @return 0  : if sensor initialization is a success
 *          -1 : if sensor initialization fails
*/
int light_sensor_init();

/**
 *  @brief Power on the light sensor
 *
 *  This function will configure the control register to power on the
light
 *  sensor.
 *
 *  @param void
 *
 *  @return void
 *
 */
void power_on_light_sensor(void);

/**
 *  @brief Create sensor,socket and hearbeat socket threads for light
task
 *
 *  The light task is made multi-threaded with
 *     1. sensor thread responsible for communicating via I2C interface
 *        with the light sensor to get light data and a socket
 *        thread.
 *     2. socket thread responsible for communicating with socket thread
and
 *        serve request from external application forwarded via socket
task.
  *      3. socket heartbeat responsible for communicating with main
task,
 *           to log heartbeat every time its requested by main task.
 *
 *  @param void
 *
 *  @return 0  : thread creation success
 *          -1 : thread creation failed
 *
 */
int create_threads(void);

/**
 *  @brief Initialize light task socket
 *
 *  This function will create, bind and make the socket listen for
incoming
 *  connections.
 *
 *  @param sock_addr_struct : pointer to sockaddr_in structure
 *
 *  @return void
 *
 */
void init_light_socket(struct sockaddr_in *sock_addr_struct);
```

```
/**
 *  @brief Entry point and executing entity for sensor thread
 *
 *  The sensor thread starts execution by invoking this
function(start_routine)
 *
 *  @param arg : argument to start_routine
 *
 *  @return void
 *
 */
void *sensor_thread_func(void *arg);

/**
 *  @brief Entry point and executing entity for socket thread
 *
 *  The socket thread starts execution by invoking this
function(start_routine)
 *
 *  @param arg : argument to start_routine
 *
 *  @return void
 *
 */
void *socket_thread_func(void *arg);

/**
 *  @brief Entry point and executing entity for socket thread
 *
 *  The socket thread for heartbeat starts execution by invoking this
function(start_routine)
 *
 *  @param arg : argument to start_routine
 *
 *  @return void
 *
 */
void *socket_hb_thread_func(void *arg);

/**
 *  @brief Get lux data from light sensor
 *
 *  This function will get the illuminance (ambient light level) in lux
and
 *  return this value.
 *
 *  @param void
 *
 *  @return float lux data
*/
float get_lux_data();

/**
 *  @brief Write light sensor register
 *
 *  This function will write to light sensor data specifed by @param(
 *  cmd_reg_val) with a value specified by @param(target_reg_val)
 *
 *  @param cmd_reg_val     : command register value
 *  @param target_reg_val  : value to be written to target register
```

```
 *
 *  @return 0   : if register write is successful
 *          -1  : if register write fails
*/
int write_light_sensor_reg(int cmd_reg_val, int target_reg_val);

/**
 *  @brief Read light sensor register
 *
 *  This function will read light sensor data specifed by @param(
 *  read_reg_val)
 *
 *  @param read_reg_val     : register to be read
 *
 *  @return reg_val   : if register read is successful
 *          -1        : if register read fails
*/
int8_t read_light_sensor_reg(uint8_t read_reg_val);

/**
 *  @brief Get the ADC channel data
 *
 *  This function will read the ADC data for channel specified by @param(
 *  channel_num) and populate them @param(ch_data_low) and
@param(ch_data_high)
 *
 *  @param channel_num     : ADC channel number to be read
 *  @param ch_data         : pointer to ADC data
 *
 *  @return void
*/
void get_adc_channel_data(int channel_num, uint16_t *ch_data);

/**
 *  @brief Calculate the lux value
 *
 *  This function calculates the illuminance value
 *
 *  @param ch0_data     : ADC channel 0 data
 *  @param ch1_data     : ADC channel 1 data
 *
 *  @return lux_val
*/
float calculate_lux_value(uint16_t ch0_data, uint16_t ch1_data);

/**
 *  @brief Log the lux value
 *
 *  This function writes the lux value calculated to logger message queue
 *
 *  @param lux_data     : lux_data
 *
 *  @return void
*/
void log_lux_data(float lux_data);

/**
 *  @brief Cleanup of the light sensor
 *
 *  This function will close the i2c bus for read and write operation and
```

```
 *  perform any cleanup required
 *
 *  @param void
 *
 *  @return void
*/
void light_sensor_exit(void);

/**
 *  @brief Create the socket and initialize
 *
 *  This function create the socket for the given socket id.
 *
 *  @param sock_fd                      : socket file descriptor
 *             server_addr_struct  : server address of the socket
 *             port_num                 : port number in which the socket
is communicating
 *             listen_qsize        : number of connections the socket is
accepting
 *
 *  @return void
*/
void init_sock(int *sock_fd, struct sockaddr_in *server_addr_struct,
               int port_num, int listen_qsize);

/**
 *  @brief Signal handler for temperature task
 *
 *  This function handles the reception of SIGKILL and SIGINT signal to
the
 *  temperature task and terminates all the threads, closes the I2C file
descriptor
 *  and logger message queue handle and exits.
 *
 *  @param sig_num              : signal number
 *
 *  @return void
*/

void sig_handler(int sig_num);

/**
 *  @brief Write command register of light sensor
 *
 *  This function will write to command register of light sensor
 *
 *  @param cmd_reg_val    : value to be written
 *
 *  @return 0   : success
 *          -1  : failure
 */
void write_cmd_reg(uint8_t cmd_reg_val);

/**
 *  @brief Read control register of light sensor
 *
 *  This function will read the control register of light sensor
 *
 *  @param void
 *
```

```
 *  @return ctrl_reg_val
*/
uint8_t read_ctrl_reg(void);

/**
 *  @brief Write control register of light sensor
 *
 *  This function will write to control register of light sensor
 *
 *  @param ctrl_reg_val    : value to be written
 *
 *  @return 0   : success
 *          -1  : failure
 */
int write_ctrl_reg(uint8_t ctrl_reg_val);

/**
 *  @brief Read timing register of light sensor
 *
 *  This function will read the timing register of light sensor
 *
 *  @param void
 *
 *  @return tim_reg_val
*/
uint8_t read_timing_reg(void);

/**
 *  @brief Write timing register of light sensor
 *
 *  This function will write to timing register of light sensor
 *
 *  @param tim_reg_val     : value to be written
 *  @param field_to_set    : timing register field to be set
 *  @param field_val       : field value
 *
 *  @return 0   : success
 *          -1  : failure
 */
int write_timing_reg(uint8_t tim_reg_val, uint8_t field_to_set, uint8_t
field_val);

/**
 *  @brief Enable or disable interrupt register of light sensor
 *
 *  This function will enable or diable the interrupt control register of
 *  light sensor
 *
 *  @param int_ctrl_reg_val     : value to be written
 *
 *  @return 0   : success
 *          -1  : failure
*/
int enable_disable_intr_ctrl_reg(uint8_t int_ctrl_reg_val);

/**
 *  @brief Read identification register of light sensor
 *
 *  This function will read the identification register of light sensor
 *
```

```
 *  @param void
 *
 *  @return tim_reg_val
*/
uint8_t read_id_reg(void);

/**
 *  @brief Read interrupt threshold register of light sensor
 *
 *  This function will read the interrupt threshold register of light
sensor
 *
 *  @param low_thresh            : pointer to low threshold value
 *  @param high_thresh           : pointer to high threshold value
 *
 *  @return void
*/
void read_intr_thresh_reg(uint16_t *low_thresh, uint16_t *high_thresh);

/**
 *  @brief Write interrupt threshold register of light sensor
 *
 *  This function will write the interrupt threshold register of light
sensor
 *
 *  @param low_thresh            : low threshold value to be written
 *  @param high_thresh           : high threshold value to be written
 *
 *  @return void
*/
void write_intr_thresh_reg(uint16_t low_thresh, uint16_t high_thresh);

void write_intr_high_thresh_reg(uint16_t high_thresh);
void write_intr_low_thresh_reg(uint16_t low_thresh);
void read_intr_low_thresh_reg(uint16_t *low_thresh);
void read_intr_low_thresh_reg(uint16_t *low_thresh);
/*---------------------------- FUNCTION PROTOTYPES ----------------------
----*/

#endif
/**************************************************************************
*
* Author:       Pavan Dhareshwar & Sridhar Pavithrapu
* Date:         03/07/2018
* File:         temperature_sensor.h
* Description:  Header file containing the macros, structs/enums, globals
                and function prototypes for source file
temperature_sensor.c
**************************************************************************
/

#ifndef _TEMPERATURE_SENSOR_TASK_H_
#define _TEMPERATURE_SENSOR_TASK_H_


/*---------------------------------- INCLUDES ---------------------------
----*/

#include <errno.h>
#include <string.h>
```

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <linux/i2c-dev.h>
#include <fcntl.h>

#include <unistd.h>
#include <signal.h>

#include <netinet/in.h>
#include <arpa/inet.h>

#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/msg.h>
#include <sys/ipc.h>
#include <sys/socket.h>

#include <mqueue.h>
#include "wrapper.h"

/*---------------------------------- GLOBALS ----------------------------
----*/
char i2c_name[10];
int sensor_thread_id, socket_thread_id, socket_hb_thread_id;
int file_descriptor;


int temp_sensor_initialized;

sig_atomic_t g_sig_kill_sensor_thread, g_sig_kill_sock_thread,
g_sig_kill_sock_hb_thread;
mqd_t logger_mq_handle;

/*---------------------------------- MACROS ----------------------------
----*/

#define I2C_SLAVE_ADDR                    0b01001000
#define I2C_SLAVE_DEV_NAME              "/dev/i2c-2"


#define I2C_TEMP_SENSOR_TEMP_DATA_REG  0b00000000 // Temperature data
register (read-only)
#define I2C_TEMP_SENSOR_CONFIG_REG         0b00000001   // command
register
#define I2C_TEMP_SENSOR_TLOW_REG      0b00000010   // T_low register
#define I2C_TEMP_SENSOR_THIGH_REG     0b00000011   // T_high register

#define SERVER_PORT_NUM                 8081
#define SERVER_LISTEN_QUEUE_SIZE        5

#define MSG_BUFF_MAX_LEN                1024
#define MSG_MAX_LEN                     128

#define MSG_QUEUE_NAME                  "/logger_task_mq"
#define MSG_QUEUE_MAX_NUM_MSGS          5
#define MSG_QUEUE_MAX_MSG_SIZE          1024

#define SOCK_REQ_MSG_API_MSG_LEN        64
```

```
#define SOCKET_HB_PORT_NUM              8650
#define SOCKET_HB_LISTEN_QUEUE_SIZE     5

#define LOGGER_ATTR_LEN                 32

/*---------------------------- STRUCTURES/ENUMERATIONS ------------------
----*/

typedef enum{

      TEMP_CELSIUS = 0,
      TEMP_KELVIN = 1,
      TEMP_FARENHEIT = 2

}tempformat_e;

struct _logger_msg_struct_
{
    char message[MSG_MAX_LEN];
    char logger_msg_src_id[LOGGER_ATTR_LEN];
    char logger_msg_level[LOGGER_ATTR_LEN];
};

enum _req_recipient_
{
    REQ_RECP_TEMP_TASK,
    REQ_RECP_LIGHT_TASK
};

struct _socket_req_msg_struct_
{
    char req_api_msg[SOCK_REQ_MSG_API_MSG_LEN];
    enum _req_recipient_ req_recipient;
    void *ptr_param_list;
};

/*---------------------------- FUNCTION PROTOTYPES ----------------------
----*/
/**
 *  @brief Write pointer register of temperature sensor
 *
 *  This function will open the i2c bus write operation of pointer
register
 *  of Temperature sensor.
 *
 *  @param value : value to be written into pointer register
 *
 *  @return void
*/
void write_pointer_register(uint8_t value);

/**
 *  @brief Write temperature high and low register of temperature sensor
 *
 *  This function will open the i2c bus write operation of temperature
high and
 *  low register of Temperature sensor.
 *
```

```
 *  @param sensor_register  : register address of either temperature high
or low register
 *            data                    : value to be written into
register
 *
 *  @return void
*/
void write_temp_high_low_register(int sensor_register, int16_t data );

/**
 *  @brief Write config register of temperature sensor
 *
 *  This function will open the i2c bus write operation of config
register of Temperature sensor.
 *
 *  @param data                  : value to be written into register
 *
 *  @return void
*/
void write_config_register_on_off(uint8_t data );

/**
 *  @brief Write config register of temperature sensor
 *
 *  This function will open the i2c bus write operation of config
register for em bits of Temperature sensor.
 *
 *  @param data                  : value to be written for em bits of
config register
 *
 *  @return void
*/
void write_config_register_em(uint8_t data );

/**
 *  @brief Write config register of temperature sensor
 *
 *  This function will open the i2c bus write operation of config
register for conversion rate of Temperature sensor.
 *
 *  @param data                  : value to be written for conversion
rate of config register
 *
 *  @return void
*/
void write_config_register_conversion_rate(uint8_t data );

/**
 *  @brief Write config register of temperature sensor
 *
 *  This function will open the i2c bus write operation of default values
into config register of Temperature sensor.
 *
 *  @param data                  : void
 *
 *  @return void
*/
void write_config_register_default( );

/**
```

```
 *  @brief Read temperature high and low register of temperature sensor
 *
 *  This function will open the i2c bus for read of temperature high and
 *  low register of Temperature sensor.
 *
 *  @param sensor_register  : register address of either temperature high
or low register
 *            data                    : value to be read from register
 *
 *  @return reg_val   : if register read is successful
 *          -1        : if register read fails
*/
int16_t read_temp_high_low_register(int sensor_register);

/**
 *  @brief Read temperature config of temperature sensor
 *
 *  This function will open the i2c bus for read config
 *  register of Temperature sensor.
 *
 *  @param void
 *
 *  @return reg_val   : if register read is successful
 *          -1        : if register read fails
*/
uint16_t read_temp_config_register();

/**
 *  @brief Read temperature data of temperature sensor
 *
 *  This function will open the i2c bus for read temperature data
 *  register of Temperature sensor.
 *
 *  @param void
 *
 *  @return temp_value  : if register read is successful
            -1 : if sensor initialization fails
*/
float read_temperature_data_register(int format);

/**
 *  @brief Initialize the temperature sensor
 *
 *  This function will open the i2c bus for read and write operation and
 *  initialize the communication with the peripheral.
 *
 *  @param void
 *
 *  @return 0  : if sensor initialization is a success
            -1 : if sensor initialization fails
*/
int temp_sensor_init();

/**
 *  @brief Log the temperature value
 *
 *  This function writes the temperature value calculated to logger
message queue
 *
 *  @param temp_data     : temperature data to be logged
```

```
 *
 *  @return void
*/
void log_temp_data(float temp_data);

/**
 *  @brief Entry point and executing entity for sensor thread
 *
 *  The sensor thread starts execution by invoking this
function(start_routine)
 *
 *  @param arg : argument to start_routine
 *
 *  @return void
 *
 */
void *sensor_thread_func(void *arg);

/**
 *  @brief Entry point and executing entity for socket thread
 *
 *  The socket thread starts execution by invoking this
function(start_routine)
 *
 *  @param arg : argument to start_routine
 *
 *  @return void
 *
 */
void *socket_thread_func(void *arg);

/**
 *  @brief Entry point and executing entity for socket thread
 *
 *  The socket thread for heartbeat starts execution by invoking this
function(start_routine)
 *
 *  @param arg : argument to start_routine
 *
 *  @return void
 *
 */
void *socket_hb_thread_func(void *arg);

/**
 *  @brief Create sensor,socket and heartbeat threads for temperature
task
 *
 *  The temperature task is made multi-threaded with
 *     1. sensor thread responsible for communicating via I2C interface
 *        with the temperature sensor to get temperature data and a
socket
 *        thread.
 *     2. socket thread responsible for communicating with socket thread
and
 *        serve request from external application forwarded via socket
task.
 *       3. socket heartbeat responsible for communicating with main
task,
 *           to log heartbeat every time its requested by main task.
```

```
 *
 *  @param void
 *
 *  @return 0  : thread creation success
 *          -1 : thread creation failed
 *
 */
int create_threads(void);

/**
 *  @brief Create the socket and initialize
 *
 *  This function create the socket for the given socket id.
 *
 *  @param sock_fd                      : socket file descriptor
 *             server_addr_struct  : server address of the socket
 *             port_num                 : port number in which the socket
is communicating
 *             listen_qsize        : number of connections the socket is
accepting
 *
 *  @return void
*/
void init_sock(int *sock_fd, struct sockaddr_in *server_addr_struct,
               int port_num, int listen_qsize);

/**
 *  @brief Signal handler for temperature task
 *
 *  This function handles the reception of SIGKILL and SIGINT signal to
the
 *  temperature task and terminates all the threads, closes the I2C file
descriptor
 *  and logger message queue handle and exits.
 *
 *  @param sig_num                      : signal number
 *
 *  @return void
*/

void sig_handler(int sig_num);
uint8_t read_config_register_em();
uint8_t read_config_register_conversion_rate();

#endif // #ifndef _TEMPERATURE_SENSOR_TASK_H_
/*************************************************************************
*
* Author:       Pavan Dhareshwar & Sridhar Pavithrapu
* Date:         03/07/2018
* File:         temperature_sensor.c
* Description:  Source file describing the functionality and
implementation
*               of temperature sensor task.
*************************************************************************
/

#include "temperature_sensor.h"
#include "wrapper.h"

int default_config_byte_one = 0X60;
```

```
int default_config_byte_two = 0XA0;

void write_pointer_register(uint8_t value){

      if (wrapper_write(file_descriptor, &value, 1) != 1) {
            perror("wrapper_write pointer register error\n");
      }
}

void write_temp_high_low_register(int sensor_register, int16_t data ){

      /* Writing to the pointer register for reading T_High/T_low
register */
      write_pointer_register(sensor_register);

      /* Writing the T_High/T_low register value */
      if (wrapper_write(file_descriptor, &data, 2) != 2) {
            perror("T-low register wrapper_write error");
      }
}


void write_config_register_on_off(uint8_t data ){

      /* Writing to the pointer register for configuration register */
      write_pointer_register(I2C_TEMP_SENSOR_CONFIG_REG);
      if((data == 0) || (data == 1)){
            default_config_byte_one |= data;

            /* Writing data to the configuration register */
            if (wrapper_write(file_descriptor, &default_config_byte_one,
1) != 1) {
                  perror("Configuration register wrapper_write error for
first byte");
            }

            if (wrapper_write(file_descriptor, &default_config_byte_two,
1) != 1) {
                  perror("Configuration register wrapper_write error for
second byte");
            }
      }
}

void write_config_register_em(uint8_t data ){

      /* Writing to the pointer register for configuration register */
      write_pointer_register(I2C_TEMP_SENSOR_CONFIG_REG);
      if((data == 0) || (data == 1)){

        uint16_t config_reg_data;
        config_reg_data = read_temp_config_register();
        printf("CONFIG_REG_DATA: %d\n", config_reg_data);

        config_reg_data = config_reg_data & (uint16_t)(~0x10);

        config_reg_data |= (uint16_t)(data << 4);

        uint8_t config_high_data = (uint8_t)(config_reg_data >> 8);
        uint8_t config_low_data = (uint8_t)(config_reg_data & 0XFF);
```

```
            //default_config_byte_two |= (data << 4);

            /* Writing data to the configuration register */
            if (wrapper_write(file_descriptor, &config_high_data, 1) != 1)
{
                  perror("Configuration register wrapper_write error for
first byte");
            }

            if (wrapper_write(file_descriptor, &config_low_data, 1) != 1)
{
                  perror("Configuration register wrapper_write error for
second byte");
            }
      }
}

void write_config_register_conversion_rate(uint8_t data ){

      /* Writing to the pointer register for configuration register */
      write_pointer_register(I2C_TEMP_SENSOR_CONFIG_REG);
      if((data >= 0) || (data <= 3)){
        uint16_t config_reg_data;
        config_reg_data = read_temp_config_register();
        config_reg_data = config_reg_data & (uint16_t)(~0xC0);

        config_reg_data |= (uint16_t)(data << 6);

        uint8_t config_high_data = (uint8_t)(config_reg_data >> 8);
        uint8_t config_low_data = (uint8_t)(config_reg_data & 0XFF);

            /* Writing data to the configuration register */
            if (wrapper_write(file_descriptor, &config_high_data, 1) != 1)
{
                  perror("Configuration register wrapper_write error for
first byte");
            }

            if (wrapper_write(file_descriptor, &config_low_data, 1) != 1)
{
                  perror("Configuration register wrapper_write error for
second byte");
            }
      }
}

void write_config_register_fault_bits(uint8_t data ){

      /* Writing to the pointer register for configuration register */
      write_pointer_register(I2C_TEMP_SENSOR_CONFIG_REG);
      if((data >= 0) || (data <= 3)){
        uint16_t config_reg_data;
        config_reg_data = read_temp_config_register();
        config_reg_data = config_reg_data & (uint16_t)(~0x1800);

        config_reg_data |= (uint16_t)(data << 11);

        uint8_t config_high_data = (uint8_t)(config_reg_data >> 8);
        uint8_t config_low_data = (uint8_t)(config_reg_data & 0XFF);
```

```
            /* Writing data to the configuration register */
            if (wrapper_write(file_descriptor, &config_high_data, 1) != 1)
{
                  perror("Configuration register wrapper_write error for
first byte");
            }

            if (wrapper_write(file_descriptor, &config_low_data, 1) != 1)
{
                  perror("Configuration register wrapper_write error for
second byte");
            }
      }
}

uint8_t read_config_register_fault_bits(){

    /* Reading fault bits of temperature config register */
    uint16_t config_value = read_temp_config_register();
    uint8_t return_value = (uint8_t)((config_value & 0x1800) >> 11);
    return return_value;

}

uint8_t read_config_register_em(){

    /* Reading em-bit of temperature config register */
    uint16_t config_value = read_temp_config_register();
#define TEMP_CONF_REG_EM_BM       0x10

    uint8_t return_value = (config_value & TEMP_CONF_REG_EM_BM) >> 4;
    return return_value;

}

uint8_t read_config_register_conversion_rate(){

    /* Reading conversion rate of temperature config register */
    uint16_t config_value = read_temp_config_register();
    uint8_t return_value = (uint8_t)((config_value & 0x00C0) >> 6);
    return return_value;
}

void write_config_register_default( ){

      /* Writing to the pointer register for configuration register */
      write_pointer_register(I2C_TEMP_SENSOR_CONFIG_REG);

      /* Writing data to the configuration register */
      if (wrapper_write(file_descriptor, &default_config_byte_one, 1) !=
1) {
            perror("Configuration register wrapper_write error for first
byte");
      }

      if (wrapper_write(file_descriptor, &default_config_byte_two, 1) !=
1) {
            perror("Configuration register wrapper_write error for second
byte");
```

```
      }
}

int16_t read_temp_high_low_register(int sensor_register){

      int16_t tlow_output_value;
    int8_t *ptr_tlow_val = (int8_t *)&tlow_output_value;
      int8_t data[2]={0};

      /* Writing to the pointer register for reading Tlow register */
      write_pointer_register(sensor_register);

      /* Reading the Tlow register value */
      if (wrapper_read(file_descriptor, data, 1) != 1) {
            perror("T-low register wrapper_read error");
      }

    printf("data[0]: %d, data[1]:%d\n", data[0], data[1]);

      tlow_output_value = ((int16_t)data[0] | ((int16_t)((data[1] & 0XF)
<< 8)));
      printf("T-low register value is: %d \n", tlow_output_value);

      return tlow_output_value;

}

uint16_t read_temp_config_register(){

      uint16_t temp_config_value;
      uint8_t data[2]={0};

      /* Writing to the pointer register for reading THigh register */
      write_pointer_register(I2C_TEMP_SENSOR_CONFIG_REG);

      /* Reading the THigh register value */
      if (wrapper_read(file_descriptor, data, 2) != 2) {
            perror("Temperature configuration register wrapper_read
error");
      }

    printf("data[0]: %d, data[1]:%d\n", data[0], data[1]);

      temp_config_value = (((int16_t)data[0])<<8 | ((int16_t)data[1]));
      printf("Temperature configuration register value is: %f \n",
temp_config_value);
      return temp_config_value;

}

float read_temperature_data_register(int format){

      float temperature_value;
      uint8_t data[3]={0};

      /* Writing to the pointer register for reading temperature data
register */
      write_pointer_register(I2C_TEMP_SENSOR_TEMP_DATA_REG);

      /* Reading the temperature data register value */
```

```
      if (wrapper_read(file_descriptor, data, 2) != 2) {
            perror("Temperature data register wrapper_read error");
      }

      if(format == TEMP_CELSIUS){
            temperature_value = (data[0]<<4 | (data[1] >> 4 & 0XF)) *
0.0625;
            printf("Temperature value is: %3.2f degree Celsius \n",
temperature_value);
      }
      else if(format == TEMP_KELVIN){
            temperature_value = (data[0]<<4 | (data[1] >> 4 & 0XF)) *
0.0625;
            temperature_value += 273.15;
            printf("Temperature value is: %3.2f degree Kelvin \n",
temperature_value);
      }
      else if(format == TEMP_FARENHEIT){
            temperature_value = (data[0]<<4 | (data[1] >> 4 & 0XF)) *
0.0625;
            temperature_value = ((temperature_value * 9)/5 + 32);
            printf("Temperature value is: %3.2f degree Fahrenheit \n",
temperature_value);

      }
      else{
            printf("Invalid format\n");
      }
      return temperature_value;

}

int temp_sensor_init()
{
      if ((file_descriptor = open(I2C_SLAVE_DEV_NAME, O_RDWR)) < 0) {
        perror("Failed to open the bus.");
        /* ERROR HANDLING; you can check errno to see what went wrong */
            return -1;
      }

      if (ioctl(file_descriptor,I2C_SLAVE,I2C_SLAVE_ADDR) < 0) {
            perror("Failed to acquire bus access and/or talk to slave");
            /* ERROR HANDLING; you can check errno to see what went wrong
*/
            return -1;
      }

    if (temp_sensor_initialized == 0)
        temp_sensor_initialized = 1;

    return 0;
}

void *sensor_thread_func(void *arg)
{

      write_config_register_default();
      float temp_value;

    while (!g_sig_kill_sensor_thread)
```

```
    {
        //temp_value = read_temperature_data_register(TEMP_CELSIUS);

        //log_temp_data(temp_value);

        sleep(10);
    }

    pthread_exit(NULL);

    return NULL;
}

void log_temp_data(float temp_data)
{
    int msg_priority;

    /* Set the message queue attributes */
    struct mq_attr logger_mq_attr = { .mq_flags = 0,
                                      .mq_maxmsg =
MSG_QUEUE_MAX_NUM_MSGS,  // Max number of messages on queue
                                      .mq_msgsize =
MSG_QUEUE_MAX_MSG_SIZE  // Max. message size
    };

    logger_mq_handle = mq_open(MSG_QUEUE_NAME, O_RDWR, S_IRWXU,
&logger_mq_attr);

    char temp_data_msg[MSG_MAX_LEN];
    memset(temp_data_msg, '\0', sizeof(temp_data_msg));

    sprintf(temp_data_msg, "Temp Value: %3.2f", temp_data);

    struct _logger_msg_struct_ logger_msg;
    memset(&logger_msg, '\0', sizeof(logger_msg));
    strcpy(logger_msg.message, temp_data_msg);
    strncpy(logger_msg.logger_msg_src_id, "Temp", strlen("Temp"));
    logger_msg.logger_msg_src_id[strlen("Temp")] = '\0';
    strncpy(logger_msg.logger_msg_level, "Info", strlen("Info"));
    logger_msg.logger_msg_level[strlen("Info")] = '\0';

    msg_priority = 2;
    int num_sent_bytes = mq_send(logger_mq_handle, (char *)&logger_msg,
            sizeof(logger_msg), msg_priority);
    if (num_sent_bytes < 0)
        perror("mq_send failed");
}

void init_sock(int *sock_fd, struct sockaddr_in *server_addr_struct,
               int port_num, int listen_qsize)
{
    int serv_addr_len = sizeof(struct sockaddr_in);

    /* Create the socket */
    if ((*sock_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0)
    {
        perror("socket creation failed");
        pthread_exit(NULL); // Change these return values from
pthread_exit
    }
```

```
    int option = 1;
    if(setsockopt(*sock_fd, SOL_SOCKET, (SO_REUSEPORT | SO_REUSEADDR),
(void *)&option, sizeof(option)) < 0)
    {
        perror("setsockopt failed");
        pthread_exit(NULL);
    }

    server_addr_struct->sin_family = AF_INET;
    server_addr_struct->sin_addr.s_addr = INADDR_ANY;
    server_addr_struct->sin_port = htons(port_num);

    if (bind(*sock_fd, (struct sockaddr *)server_addr_struct,
                                            sizeof(struct
sockaddr_in))<0)
    {
        perror("bind failed");
        pthread_exit(NULL);
    }

    if (listen(*sock_fd, listen_qsize) < 0)
    {
        perror("listen failed");
        pthread_exit(NULL);
    }

}

void *socket_thread_func(void *arg)
{
    int server_fd;
    struct sockaddr_in server_address;
    int serv_addr_len = sizeof(server_address);

    init_sock(&server_fd, &server_address, SERVER_PORT_NUM,
SERVER_LISTEN_QUEUE_SIZE);

    int accept_conn_id;
    printf("Waiting for request...\n");
    if ((accept_conn_id = accept(server_fd, (struct sockaddr
*)&server_address,
                    (socklen_t*)&serv_addr_len)) < 0)
    {
        perror("accept failed");
        //pthread_exit(NULL);
    }

    char recv_buffer[MSG_BUFF_MAX_LEN];

    while (!g_sig_kill_sock_thread)
    {
        memset(recv_buffer, '\0', sizeof(recv_buffer));

        size_t num_read_bytes = read(accept_conn_id, &recv_buffer,
sizeof(recv_buffer));

        printf("[Temp_Task] Message req api: %s, req recp: %s, req api
params: %s\n",
```

```
                (((struct _socket_req_msg_struct_ *)&recv_buffer)-
>req_api_msg),
                ((((struct _socket_req_msg_struct_ *)&recv_buffer)-
>req_recipient)
                 == REQ_RECP_TEMP_TASK ? "Temp Task" : "Light Task"),
                ((struct _socket_req_msg_struct_ *)&recv_buffer)-
>ptr_param_list != NULL ?
                 ((struct _socket_req_msg_struct_ *)&recv_buffer)-
>ptr_param_list :"NULL"));

        if (!strcmp((((struct _socket_req_msg_struct_ *)&recv_buffer)-
>req_api_msg), "get_temp_data"))
        {
            float temp_data =
read_temperature_data_register(TEMP_CELSIUS);
            char temp_data_msg[64];
            memset(temp_data_msg, '\0', sizeof(temp_data_msg));

            sprintf(temp_data_msg, "Temp Data: %3.2f", temp_data);

            ssize_t num_sent_bytes = send(accept_conn_id, temp_data_msg,
strlen(temp_data_msg), 0);
            if (num_sent_bytes < 0)
                perror("send failed");
        }
           else if (!strcmp((((struct _socket_req_msg_struct_
*)&recv_buffer)->req_api_msg), "get_temp_low_data"))
        {
            int16_t temp_data =
read_temp_high_low_register(I2C_TEMP_SENSOR_TLOW_REG);
            char temp_data_msg[64];
            memset(temp_data_msg, '\0', sizeof(temp_data_msg));

            sprintf(temp_data_msg, "Tlow Data: %d", temp_data);

            ssize_t num_sent_bytes = send(accept_conn_id, temp_data_msg,
strlen(temp_data_msg), 0);
            if (num_sent_bytes < 0)
                perror("send failed");
        }
           else if (!strcmp((((struct _socket_req_msg_struct_
*)&recv_buffer)->req_api_msg), "get_temp_high_data"))
        {
            int16_t temp_data =
read_temp_high_low_register(I2C_TEMP_SENSOR_THIGH_REG);
            char temp_data_msg[64];
            memset(temp_data_msg, '\0', sizeof(temp_data_msg));

            sprintf(temp_data_msg, "T_High Data: %d", temp_data);

            ssize_t num_sent_bytes = send(accept_conn_id, temp_data_msg,
strlen(temp_data_msg), 0);
            if (num_sent_bytes < 0)
                perror("send failed");
        }
           else if (!strcmp((((struct _socket_req_msg_struct_
*)&recv_buffer)->req_api_msg), "get_temp_em"))
        {
            uint8_t temp_data = read_config_register_em();
            char temp_data_msg[64];
```

```
            memset(temp_data_msg, '\0', sizeof(temp_data_msg));

            sprintf(temp_data_msg, "T_High Data: %d", temp_data);

            ssize_t num_sent_bytes = send(accept_conn_id, temp_data_msg,
strlen(temp_data_msg), 0);
            if (num_sent_bytes < 0)
                perror("send failed");
        }
           else if (!strcmp((((struct _socket_req_msg_struct_
*)&recv_buffer)->req_api_msg), "get_temp_conversion_rate"))
        {
            uint8_t temp_data = read_config_register_conversion_rate();
            char temp_data_msg[64];
            memset(temp_data_msg, '\0', sizeof(temp_data_msg));

            sprintf(temp_data_msg, "T_High Data: %d", temp_data);

            ssize_t num_sent_bytes = send(accept_conn_id, temp_data_msg,
strlen(temp_data_msg), 0);
            if (num_sent_bytes < 0)
                perror("send failed");
        }
           else if (!strcmp((((struct _socket_req_msg_struct_
*)&recv_buffer)->req_api_msg), "get_temp_conf_data"))
        {
            uint16_t temp_data = read_temp_config_register();
            char temp_data_msg[64];
            memset(temp_data_msg, '\0', sizeof(temp_data_msg));

            sprintf(temp_data_msg, "Conf Data: %d", temp_data);

            ssize_t num_sent_bytes = send(accept_conn_id, temp_data_msg,
strlen(temp_data_msg), 0);
            if (num_sent_bytes < 0)
                perror("send failed");
        }
           else if (!strcmp((((struct _socket_req_msg_struct_
*)&recv_buffer)->req_api_msg), "set_temp_on_off"))
        {
            if((((struct _socket_req_msg_struct_ *)&recv_buffer)-
>ptr_param_list) != NULL){

                        uint8_t data = *(uint8_t *)(((struct
_socket_req_msg_struct_ *)&recv_buffer)->ptr_param_list);
                        write_config_register_on_off(data);
                        char temp_data_msg[64];
                        memset(temp_data_msg, '\0',
sizeof(temp_data_msg));

                        sprintf(temp_data_msg, "%s", "Set success");

                        ssize_t num_sent_bytes = send(accept_conn_id,
temp_data_msg, strlen(temp_data_msg), 0);
                        if (num_sent_bytes < 0)
                              perror("send failed");
                }
        }
           else if (!strcmp((((struct _socket_req_msg_struct_
*)&recv_buffer)->req_api_msg), "set_temp_em"))
```

```
        {
            if((((struct _socket_req_msg_struct_ *)&recv_buffer)-
>ptr_param_list) != NULL){

                        uint8_t data = *(uint8_t *)(((struct
_socket_req_msg_struct_ *)&recv_buffer)->ptr_param_list);
                        printf("DATA:::::: %d\n", data);
                write_config_register_em(data);
                        char temp_data_msg[64];
                        memset(temp_data_msg, '\0',
sizeof(temp_data_msg));

                        sprintf(temp_data_msg, "%s", "Set success");

                        ssize_t num_sent_bytes = send(accept_conn_id,
temp_data_msg, strlen(temp_data_msg), 0);
                        if (num_sent_bytes < 0)
                              perror("send failed");
                }
        }
           else if (!strcmp((((struct _socket_req_msg_struct_
*)&recv_buffer)->req_api_msg), "set_temp_conversion_rate"))
        {
            if((((struct _socket_req_msg_struct_ *)&recv_buffer)-
>ptr_param_list) != NULL){

                        uint8_t data = *(uint8_t *)(((struct
_socket_req_msg_struct_ *)&recv_buffer)->ptr_param_list);
                        write_config_register_conversion_rate(data);
                        char temp_data_msg[64];
                        memset(temp_data_msg, '\0',
sizeof(temp_data_msg));

                        sprintf(temp_data_msg, "%s", "Set success");

                        ssize_t num_sent_bytes = send(accept_conn_id,
temp_data_msg, strlen(temp_data_msg), 0);
                        if (num_sent_bytes < 0)
                              perror("send failed");
                }
        }
           else if (!strcmp((((struct _socket_req_msg_struct_
*)&recv_buffer)->req_api_msg), "set_temp_high_data"))
        {
            if((((struct _socket_req_msg_struct_ *)&recv_buffer)-
>ptr_param_list) != NULL){

                        int16_t data = *(uint8_t *)(((struct
_socket_req_msg_struct_ *)&recv_buffer)->ptr_param_list);

      write_temp_high_low_register(I2C_TEMP_SENSOR_THIGH_REG,data);
                        char temp_data_msg[64];
                        memset(temp_data_msg, '\0',
sizeof(temp_data_msg));

                        sprintf(temp_data_msg, "%s", "Set success");

                        ssize_t num_sent_bytes = send(accept_conn_id,
temp_data_msg, strlen(temp_data_msg), 0);
                        if (num_sent_bytes < 0)
```

```
                        perror("send failed");
                }
        }
           else if (!strcmp((((struct _socket_req_msg_struct_
*)&recv_buffer)->req_api_msg), "set_temp_low_data"))
        {
            if((((struct _socket_req_msg_struct_ *)&recv_buffer)-
>ptr_param_list) != NULL){

                        int16_t data = *(uint8_t *)(((struct
_socket_req_msg_struct_ *)&recv_buffer)->ptr_param_list);

      write_temp_high_low_register(I2C_TEMP_SENSOR_TLOW_REG,data);
                        char temp_data_msg[64];
                        memset(temp_data_msg, '\0',
sizeof(temp_data_msg));

                        sprintf(temp_data_msg, "%s", "Set success");

                        ssize_t num_sent_bytes = send(accept_conn_id,
temp_data_msg, strlen(temp_data_msg), 0);
                        if (num_sent_bytes < 0)
                             perror("send failed");
                }
        }
           else if (!strcmp((((struct _socket_req_msg_struct_
*)&recv_buffer)->req_api_msg), "set_temp_fault_bits"))
        {
            if((((struct _socket_req_msg_struct_ *)&recv_buffer)-
>ptr_param_list) != NULL){

                        uint8_t data = *(uint8_t *)(((struct
_socket_req_msg_struct_ *)&recv_buffer)->ptr_param_list);
                        write_config_register_fault_bits(data);
                        char temp_data_msg[64];
                        memset(temp_data_msg, '\0',
sizeof(temp_data_msg));

                        sprintf(temp_data_msg, "%s", "Set success");

                        ssize_t num_sent_bytes = send(accept_conn_id,
temp_data_msg, strlen(temp_data_msg), 0);
                        if (num_sent_bytes < 0)
                             perror("send failed");
                }
        }
           else if (!strcmp((((struct _socket_req_msg_struct_
*)&recv_buffer)->req_api_msg), "get_temp_fault_bits"))
        {
            uint8_t temp_data = read_config_register_fault_bits();
            char temp_data_msg[64];
            memset(temp_data_msg, '\0', sizeof(temp_data_msg));

            printf("Fault Bits: %d", temp_data);
            sprintf(temp_data_msg, "Fault Bits: %d", temp_data);

            ssize_t num_sent_bytes = send(accept_conn_id, temp_data_msg,
strlen(temp_data_msg), 0);
            if (num_sent_bytes < 0)
                perror("send failed");
```

```
        }
    }

    printf("Calling pthread_exit in sock thread\n");
    pthread_exit(NULL);

    return NULL;
}

void *socket_hb_thread_func(void *arg)
{
    int sock_hb_fd;
    struct sockaddr_in sock_hb_address;
    int sock_hb_addr_len = sizeof(sock_hb_address);

    init_sock(&sock_hb_fd, &sock_hb_address, SOCKET_HB_PORT_NUM,
SOCKET_HB_LISTEN_QUEUE_SIZE);

    int accept_conn_id;
    printf("Waiting for request...\n");
    if ((accept_conn_id = accept(sock_hb_fd, (struct sockaddr
*)&sock_hb_address,
                    (socklen_t*)&sock_hb_addr_len)) < 0)
    {
        perror("accept failed");
        //pthread_exit(NULL);
    }

    char recv_buffer[MSG_BUFF_MAX_LEN];
    char send_buffer[20];
    memset(send_buffer, '\0', sizeof(send_buffer));

    while (!g_sig_kill_sock_hb_thread)
    {
        memset(recv_buffer, '\0', sizeof(recv_buffer));

        size_t num_read_bytes = read(accept_conn_id, &recv_buffer,
sizeof(recv_buffer));

        if (!strcmp(recv_buffer, "heartbeat"))
        {
            strcpy(send_buffer, "Alive");
                 ssize_t num_sent_bytes = send(accept_conn_id,
send_buffer, strlen(send_buffer), 0);
            if (num_sent_bytes < 0)
                perror("send failed");
        }
        else if (!strcmp(recv_buffer, "startup_check"))
        {
            /* For the sake of start-up check, because we have the
temperature sensor initialized
            ** by the time this thread is spawned. So we perform a
"get_temp_data" call to see if
            ** everything is working fine */
            if (temp_sensor_initialized == 1)
                strcpy(send_buffer, "Initialized");
            else
                strcpy(send_buffer, "Uninitialized");
```

```
                ssize_t num_sent_bytes = send(accept_conn_id,
send_buffer, strlen(send_buffer), 0);
            if (num_sent_bytes < 0)
                perror("send failed");
        }
    }

    printf("Calling pthread_exit in sock hb thread\n");
    pthread_exit(NULL);

    return NULL;
}

void sig_handler(int sig_num)
{
      char buffer[MSG_BUFF_MAX_LEN];
      memset(buffer, '\0', sizeof(buffer));

      if (sig_num == SIGINT || sig_num == SIGUSR1)
      {
        if (sig_num == SIGINT)
            printf("Caught signal %s in temperature task\n", "SIGINT");
        else if (sig_num == SIGUSR1)
            printf("Caught signal %s in temperature task\n", "SIGKILL");

        g_sig_kill_sensor_thread = 1;
        g_sig_kill_sock_thread = 1;
        g_sig_kill_sock_hb_thread = 1;

        //pthread_join(sensor_thread_id, NULL);
        //pthread_join(socket_thread_id, NULL);
        //pthread_join(socket_hb_thread_id, NULL);

        mq_close(logger_mq_handle);

        close(file_descriptor);

        exit(0);
    }
}

int create_threads()
{
    int sens_t_creat_ret_val = pthread_create(&sensor_thread_id, NULL,
&sensor_thread_func, NULL);
    if (sens_t_creat_ret_val)
    {
        perror("Sensor thread creation failed");
        return -1;
    }

    int sock_t_creat_ret_val = pthread_create(&socket_thread_id, NULL,
&socket_thread_func, NULL);
    if (sock_t_creat_ret_val)
    {
        perror("Socket thread creation failed");
        return -1;
    }
```

```
    int sock_hb_t_creat_ret_val = pthread_create(&socket_hb_thread_id,
NULL, &socket_hb_thread_func, NULL);
    if (sock_hb_t_creat_ret_val)
    {
        perror("Socket heartbeat thread creation failed");
        return -1;
    }

    return 0;
}

int main()
{
    temp_sensor_initialized = 0;

    int temp_sensor_init_status = temp_sensor_init();
    if (temp_sensor_init_status == -1)
    {
        printf("Temperature sensor init failed\n");
        exit(1);
    }
    else
    {
        printf("Temperature sensor init success\n");
    }

    int thread_create_status = create_threads();
    if (thread_create_status)
    {
        printf("Thread creation failed\n");
    }
    else
    {
        printf("Thread creation success\n");
    }

    if (signal(SIGINT, sig_handler) == SIG_ERR)
        printf("SigHandler setup for SIGINT failed\n");

    if (signal(SIGUSR1, sig_handler) == SIG_ERR)
        printf("SigHandler setup for SIGKILL failed\n");

    g_sig_kill_sensor_thread = 0;
    g_sig_kill_sock_thread = 0;
    g_sig_kill_sock_hb_thread = 0;

    pthread_join(sensor_thread_id, NULL);
    pthread_join(socket_thread_id, NULL);
    pthread_join(socket_hb_thread_id, NULL);

    close(file_descriptor);

    return 0;
}
/*************************************************************************
******
* Author:       Pavan Dhareshwar & Sridhar Pavithrapu
* Date:         03/07/2018
* File:         wrapper.c
```

```
* Description:  Source file describing the functionality and
implementation
*               of wrapper for synchronization of light and temperature
tasks.
*****************************************************************************
*****/

/*---------------------------------- INCLUDES ---------------------------
----*/
#include "wrapper.h"

sem_t *get_named_semaphore_handle(void)
{
    sem_t *sem;
    if ((sem = sem_open("wrapper_sem", O_CREAT, 0644, 1)) == SEM_FAILED)
    {
        perror("sem_open failed");
        return SEM_FAILED;
    }
    else
    {
        printf("Named semaphore created successfully\n");
        return sem;
    }
}

ssize_t wrapper_write(int fd, void *buf, size_t count){

     ssize_t return_value = 0;

#if 1
    sem_t *wrapper_sem = get_named_semaphore_handle();
    if (wrapper_sem == SEM_FAILED)
    {
        return -1000;
    }

     if(sem_wait(wrapper_sem) == 0)
    {
            return_value = write(fd, buf, count);
    }
     else{
          perror("sem_wait error in wrapper\n");
     }

     if(sem_post(wrapper_sem) != 0){

          perror("sem_post error in wrapper\n");
     }
#else
    return_value = write(fd, buf, count);
#endif
     return return_value;
}

ssize_t wrapper_read(int fd, void *buf, size_t count){

     ssize_t return_value = 0;

#if 1
```

```
    sem_t *wrapper_sem = get_named_semaphore_handle();
    if (wrapper_sem == SEM_FAILED)
    {
        return -1000;
    }

    if(sem_wait(wrapper_sem) == 0){

            return_value = read(fd, buf, count);
    }
     else{
           perror("sem_wait error in wrapper\n");
     }

     if(sem_post(wrapper_sem) != 0){

           perror("sem_post error in wrapper\n");
     }
#else
    return_value = read(fd, buf, count);
#endif

     return return_value;
}
#ifndef _WRAPPER_H_
#define _WRAPPER_H_

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

sem_t *get_named_semaphore_handle(void);

ssize_t wrapper_write(int fd, void *buf, size_t count);
ssize_t wrapper_read(int fd, void *buf, size_t count);

#endif
/*****************************************************************************
*************
*
* FileName        :    temp_unit_tests.c
* Description     :    This file contains necessary  test functions for
temperature task.

* File Author Name:    Sridhar Pavithrapu
* Tools used      :    gcc, gedit, cmocka
* References      :    None
*
*****************************************************************************
*************/
```

```
/* Headers Section */
#include <math.h>
#include <stdlib.h>
#include <stdarg.h>
#include <stddef.h>
#include <setjmp.h>
#include <cmocka.h>
#include "temperature_sensor.h"

/* Macros section */


/**
*   @brief : test function for writing and reading temperature
*                  configuration register.
*
*
*   @param  state   A pointer to the state
*
*   @return None
*/
void test_temperature_config_register(void **state){

      /* Test case for checking temperature configuration register */
      write_config_register_default();
      uint16_t return_value = read_temp_config_register();
      assert_int_equal(return_value, ((default_config_byte_one << 8) |
default_config_byte_two));

}

/**
*   @brief : test function for checking temperature
*                  high threshold value register.
*
*
*   @param  state   A pointer to the state
*
*   @return None
*/
void test_temperature_threshold_high(void **state){

      /* Test case for checking temperature high threshold value register
*/
      uint16_t data = 0X9876;
      write_temp_high_low_register(I2C_TEMP_SENSOR_THIGH_REG,data);
      uint16_t return_value =
read_temp_high_low_register(I2C_TEMP_SENSOR_THIGH_REG);
      assert_int_equal(return_value, data);
}

/**
*   @brief : test function for checking temperature
*                  low threshold value register.
*
*
*   @param  state   A pointer to the state
*
*   @return None
*/
```

```
void test_temperature_threshold_low(void **state){

      /* Test case for checking temperature low threshold value register
*/
      uint16_t data = 0X7676;
      write_temp_high_low_register(I2C_TEMP_SENSOR_TLOW_REG,data);
      uint16_t return_value =
read_temp_high_low_register(I2C_TEMP_SENSOR_TLOW_REG);
      assert_int_equal(return_value, data);
}

/**
*   @brief : test function for checking em-bit of config register
*
*
*   @param  state   A pointer to the state
*
*   @return None
*/
void test_temperature_config_em(void **state){

      /* Test case for checking em-bit of config register */
      int data = 1;
      write_config_register_em(data);
      uint16_t return_value = read_config_register_em();
      assert_int_equal(return_value, data);

}

/**
*   @brief : test function for checking conversion rate of config
register
*
*
*   @param  state   A pointer to the state
*
*   @return None
*/
void test_temperature_config_conversion_rate(void **state){

      /* Test case for checking conversion rate of config register */
      int data = 1;
      write_config_register_conversion_rate(data);
      uint16_t return_value = read_config_register_conversion_rate();
      assert_int_equal(return_value, data);

}

/**
*   @brief : test function for checking negative scenario conversion rate
of config register
*
*
*   @param  state   A pointer to the state
*
*   @return None
*/
void test_temperature_config_conversion_rate_false(void **state){
```

```
      /* Test case for checking negative scenario conversion rate of
config register */
      int data = 1;
      write_config_register_conversion_rate(data);
      uint16_t return_value = read_config_register_conversion_rate();
      assert_int_equal(return_value, data);

}



/**
*   @brief : main function for all DLL test cases
*
*
*   @return Pass and Fail test cases
*/
int main(int argc, char **argv){

      /* Calling all DLL test case functions */
      const struct CMUnitTest tests[] = {
            cmocka_unit_test(test_temperature_config_register),
            cmocka_unit_test(test_temperature_threshold_high),
            cmocka_unit_test(test_temperature_threshold_low),
            cmocka_unit_test(test_temperature_config_em),
            cmocka_unit_test(test_temperature_config_conversion_rate),

      cmocka_unit_test(test_temperature_config_conversion_rate_false),

      };

      return cmocka_run_group_tests(tests, NULL, NULL);
}/*********************************************************************
**
* Author:       Pavan Dhareshwar & Sridhar Pavithrapu
* Date:         03/08/2018
* File:         logger_task.h
* Description:  Header file containing the macros, structs/enums, globals
                and function prototypes for source file logger_task.c
*********************************************************************
/

#ifndef _LOGGER_TASK_H_
#define _LOGGER_TASK_H_

/*---------------------------------- INCLUDES ---------------------------
----*/
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <stdint.h>
#include <string.h>
#include <time.h>

#include <unistd.h>
#include <fcntl.h>
#include <pthread.h>
#include <signal.h>

#include <sys/types.h>
```

```
#include <sys/stat.h>
#include <sys/ipc.h>
#include <sys/types.h>
#include <sys/msg.h>

#include <mqueue.h>

#include <netinet/in.h>
#include <arpa/inet.h>
/*---------------------------------- INCLUDES ----------------------------
----*/

/*----------------------------------- MACROS -----------------------------
----*/
// Message queue attribute macros
#define MSG_QUEUE_MAX_NUM_MSGS              5
#define MSG_QUEUE_MAX_MSG_SIZE              1024
#define MSG_QUEUE_NAME                      "/logger_task_mq"

#define LOGGER_FILE_PATH                    "./"
#define LOGGER_FILE_NAME                    "logger_file.txt"

#define LOG_MSG_PAYLOAD_SIZE                256
#define MSG_MAX_LEN                         128

#define MSG_BUFF_MAX_LEN                    1024

#define LOGGER_FILE_PATH_LEN                256
#define LOGGER_FILE_NAME_LEN                64

#define SOCKET_HB_PORT_NUM                  8680
#define SOCKET_HB_LISTEN_QUEUE_SIZE         10

#define MSG_TYPE_TEMP_DATA                  0
#define MSG_TYPE_LUX_DATA                   1
#define MSG_TYPE_SOCK_DATA                  2
#define MSG_TYPE_MAIN_DATA                  3

#define LOGGER_ATTR_LEN                     32

/*----------------------------------- MACROS -----------------------------
----*/

/*---------------------------------- GLOBALS -----------------------------
----*/
mqd_t logger_mq_handle;
int logger_fd;
pthread_t logger_thread_id, socket_hb_thread_id;

sig_atomic_t g_sig_kill_logger_thread, g_sig_kill_sock_hb_thread;

/*---------------------------------- GLOBALS -----------------------------
----*/

/*--------------------------- STRUCTURES/ENUMERATIONS -------------------
----*/
struct _logger_msg_struct_
{
    char message[MSG_MAX_LEN];
    char logger_msg_src_id[LOGGER_ATTR_LEN];
```

```
    char logger_msg_level[LOGGER_ATTR_LEN];
};

/*---------------------------- STRUCTURES/ENUMERATIONS ------------------
----*/

/*---------------------------- FUNCTION PROTOTYPES ----------------------
----*/
/**
 *  @brief Initialize the logger task
 *
 *  This funcition will create the message queue for logger task and
 *  open a file handle of logger file for writing. (If the logger file
 *  already exists, it is deleted and a fresh one is created).
 *
 *  @param void
 *
 *  @return 0  : if sensor initialization is a success
            -1 : if sensor initialization fails
*/
int logger_task_init();

/**
 *  @brief Read from configuration file for the logger task
 *
 *  This function reads the configuration parameters for the logger task
file
 *  and sets-up the logger file as per this configuration
 *
 *  @param file     : name of the config file
 *
 *  @return void
 *
 */
int read_logger_conf_file(char *file);

/**
 *  @brief Create logger and hearbeat socket threads for logger task
 *
 *  The logger task is made multi-threaded with
 *     1. logger thread responsible for reading messages from its message
queue
 *        and logging it to a file.
 *     2. socket heartbeat responsible for communicating with main task,
 *        to log heartbeat every time its requested by main task.
 *
 *  @param void
 *
 *  @return 0  : thread creation success
 *          -1 : thread creation failed
 *
 */
int create_threads(void);

/**
 *  @brief Entry point and executing entity for logger thread
 *
 *  The logger thread starts execution by invoking this
function(start_routine)
 *
```

```
 *  @param arg : argument to start_routine
 *
 *  @return void
 *
 */
void *logger_thread_func(void *arg);

/**
 *  @brief Entry point and executing entity for socket thread
 *
 *  The socket thread for heartbeat starts execution by invoking this
function(start_routine)
 *
 *  @param arg : argument to start_routine
 *
 *  @return void
 *
 */
void *socket_hb_thread_func(void *arg);

/**
 *  @brief Create the socket and initialize
 *
 *  This function create the socket for the given socket id.
 *
 *  @param sock_fd              : socket file descriptor
 *         server_addr_struct   : server address of the socket
 *         port_num             : port number in which the socket is
communicating
 *         listen_qsize         : number of connections the socket is
accepting
 *
 *  @return void
*/
void init_sock(int *sock_fd, struct sockaddr_in *server_addr_struct,
               int port_num, int listen_qsize);


int write_test_msg_to_logger();
void read_test_msg_to_logger(char * buffer);

/**
 *  @brief Read message from logger message queue
 *
 *  This function will read messages from its message queue and log it to
a file
 *
 *  @param void
 *
 *  @return void
*/
void read_from_logger_msg_queue(void);

/**
 *  @brief Cleanup of the logger sensor
 *
 *  This function will close the message queue and the logger file handle
 *
 *  @param void
 *
```

```
 *  @return void
*/
void logger_task_exit(void);

/**
 *  @brief Signal handler for temperature task
 *
 *  This function handles the reception of SIGKILL and SIGINT signal to
the
 *  temperature task and terminates all the threads, closes the I2C file
descriptor
 *  and logger message queue handle and exits.
 *
 *  @param sig_num              : signal number
 *
 *  @return void
*/

void sig_handler(int sig_num);
#endif // _LOGGER_TASK_H_
/*************************************************************************
*
* Author:       Pavan Dhareshwar & Sridhar Pavithrapu
* Date:         03/08/2018
* File:         logger_task.c
* Description:  Source file describing the functionality and
implementation
*               of logger task.
*************************************************************************
/

#include "logger_task.h"

int main(void)
{
    printf("In Logger task\n");

    int init_status = logger_task_init();
    if (init_status == -1)
    {
        printf("logger task initialization failed\n");
        exit(1);
    }

    //write_test_msg_to_logger();

    int thread_create_status = create_threads();
    if (thread_create_status)
    {
        printf("Thread creation failed\n");
    }
    else
    {
        printf("Thread creation success\n");
    }

    if (signal(SIGINT, sig_handler) == SIG_ERR)
        printf("SigHandler setup for SIGINT failed\n");

    if (signal(SIGUSR1, sig_handler) == SIG_ERR)
```

```
        printf("SigHandler setup for SIGKILL failed\n");

    g_sig_kill_logger_thread = 0;
    g_sig_kill_sock_hb_thread = 0;

    pthread_join(logger_thread_id, NULL);
    pthread_join(socket_hb_thread_id, NULL);

    logger_task_exit();

    return 0;
}

int logger_task_init()
{
    /* In the logger task init function, we create the message queue */

    /* Set the message queue attributes */
    struct mq_attr logger_mq_attr = { .mq_flags = 0,
                                      .mq_maxmsg =
MSG_QUEUE_MAX_NUM_MSGS,  // Max number of messages on queue
                                      .mq_msgsize =
MSG_QUEUE_MAX_MSG_SIZE  // Max. message size
                                    };


    logger_mq_handle = mq_open(MSG_QUEUE_NAME, O_CREAT | O_RDWR, S_IRWXU,
&logger_mq_attr);
    if (logger_mq_handle < 0)
    {
        perror("Logger message queue create failed");
        return -1;
    }

    printf("Logger message queue successfully created\n");

    char filename[100];
    memset(filename, '\0', sizeof(filename));
    int conf_file_read_status = read_logger_conf_file(filename);
    if (conf_file_read_status != 0)
    {
        printf("Logger task config file read failed. Using default log
file path and name\n");
        sprintf(filename, "%s%s", LOGGER_FILE_PATH, LOGGER_FILE_NAME);
    }

    if (open(filename, O_RDONLY) != -1)
    {
        printf("Logger file exists. Deleting existing file.\n");
        remove(filename);
        sync();
    }

    printf("Trying to create file %s\n", filename);
    logger_fd = creat(filename ,(S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH));
    if (logger_fd == -1)
    {
        perror("Logger file open failed");
        return -1;
    }
```

```
    else
    {
        printf("Logger file open success\n");
    }

    return 0;
}

int read_logger_conf_file(char *file)
{
    FILE *fp_conf_file = fopen("./logger_task_conf_file.txt", "r");
    if (fp_conf_file == NULL)
    {
        perror("file open failed");
        printf("File %s open failed\n", "logger_task_conf_file.txt");
        return -1;
    }

    char logger_file_path[LOGGER_FILE_PATH_LEN];
    char logger_file_name[LOGGER_FILE_NAME_LEN];
    char *buffer;
    size_t num_bytes = 120;
    char equal_delimiter[] = "=";
    ssize_t bytes_read;

    memset(logger_file_path, '\0', sizeof(logger_file_path));
    memset(logger_file_name, '\0', sizeof(logger_file_name));

    buffer = (char *)malloc(num_bytes*sizeof(char));

    while ((bytes_read = getline(&buffer, &num_bytes, fp_conf_file)) != -
1)
    {
        char *token = strtok(buffer, equal_delimiter);

        if (!strcmp(token, "LOGGER_FILE_PATH"))
        {
            token = strtok(NULL, equal_delimiter);
            strcpy(logger_file_path, token);
            int len = strlen(logger_file_path);
            if (logger_file_path[len-1] == '\n')
                logger_file_path[len-1] = '\0';
        }
        else if (!strcmp(token, "LOGGER_FILE_NAME"))
        {
            token = strtok(NULL, equal_delimiter);
            strcpy(logger_file_name, token);
            int len = strlen(logger_file_name);
            if (logger_file_name[len-1] == '\n')
                logger_file_name[len-1] = '\0';
        }
    }

    strcpy(file, logger_file_path);
    strcat(file, logger_file_name);

    if (buffer)
        free(buffer);

    if (fp_conf_file)
```

```
        fclose(fp_conf_file);

    return 0;
}

int create_threads(void)
{
    int logger_t_creat_ret_val = pthread_create(&logger_thread_id, NULL,
&logger_thread_func, NULL);
    if (logger_t_creat_ret_val)
    {
        perror("Sensor thread creation failed");
        return -1;
    }

    int sock_hb_t_creat_ret_val = pthread_create(&socket_hb_thread_id,
NULL, &socket_hb_thread_func, NULL);
    if (sock_hb_t_creat_ret_val)
    {
        perror("Socket heartbeat thread creation failed");
        return -1;
    }

    return 0;
}

void *logger_thread_func(void *arg)
{
    while(!g_sig_kill_logger_thread)
    {
        /* This function will continously read from the logger task
message
        ** queue and write it to logger file */
        read_from_logger_msg_queue();
    }

    pthread_exit(NULL);
}

void *socket_hb_thread_func(void *arg)
{
    int sock_hb_fd;
    struct sockaddr_in sock_hb_address;
    int sock_hb_addr_len = sizeof(sock_hb_address);

    init_sock(&sock_hb_fd, &sock_hb_address, SOCKET_HB_PORT_NUM,
SOCKET_HB_LISTEN_QUEUE_SIZE);


    int accept_conn_id;
    printf("Waiting for request...\n");
    if ((accept_conn_id = accept(sock_hb_fd, (struct sockaddr
*)&sock_hb_address,
                    (socklen_t*)&sock_hb_addr_len)) < 0)
    {
        perror("accept failed");
        //pthread_exit(NULL);
    }

    char recv_buffer[MSG_BUFF_MAX_LEN];
```

```
    char send_buffer[] = "Alive";

    while (!g_sig_kill_sock_hb_thread)
    {
        memset(recv_buffer, '\0', sizeof(recv_buffer));

        size_t num_read_bytes = read(accept_conn_id, &recv_buffer,
sizeof(recv_buffer));

        if (!strcmp(recv_buffer, "heartbeat"))
        {
                 ssize_t num_sent_bytes = send(accept_conn_id,
send_buffer, strlen(send_buffer), 0);
            if (num_sent_bytes < 0)
                perror("send failed");
        }
    }
}


int write_test_msg_to_logger(char *test_msg)
{
    struct _logger_msg_struct_ logger_msg = {0};

    strcpy(logger_msg.message, test_msg);
    logger_msg.msg_len = strlen(test_msg);

    logger_msg.logger_msg_type = MSG_TYPE_TEMP_DATA;

    int msg_priority = 1;
    int num_sent_bytes = mq_send(logger_mq_handle, (char *)&logger_msg,
                                    sizeof(logger_msg), msg_priority);

    if (num_sent_bytes < 0)
        perror("mq_send failed");
           return -1;

      return 0;
}

void read_test_msg_to_logger(char * buffer)
{
    char recv_buffer[MSG_MAX_LEN];
    memset(recv_buffer, '\0', sizeof(recv_buffer));

    int msg_priority;

    int num_recv_bytes;
    num_recv_bytes = mq_receive(logger_mq_handle, (char *)&recv_buffer,
                                    MSG_QUEUE_MAX_MSG_SIZE,
&msg_priority));

      strcpy(buffer, recv_buffer);

}


void init_sock(int *sock_fd, struct sockaddr_in *server_addr_struct,
               int port_num, int listen_qsize)
{
```

```
    int serv_addr_len = sizeof(struct sockaddr_in);

    /* Create the socket */
    if ((*sock_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0)
    {
        perror("socket creation failed");
        pthread_exit(NULL); // Change these return values from
pthread_exit
    }

    int option = 1;
    if(setsockopt(*sock_fd, SOL_SOCKET, (SO_REUSEPORT | SO_REUSEADDR),
(void *)&option, sizeof(option)) < 0)
    {
        perror("setsockopt failed");
        pthread_exit(NULL);
    }

    server_addr_struct->sin_family = AF_INET;
    server_addr_struct->sin_addr.s_addr = INADDR_ANY;
    server_addr_struct->sin_port = htons(port_num);

    if (bind(*sock_fd, (struct sockaddr *)server_addr_struct,
                                            sizeof(struct
sockaddr_in))<0)
    {
        perror("bind failed");
        pthread_exit(NULL);
    }

    if (listen(*sock_fd, listen_qsize) < 0)
    {
        perror("listen failed");
        pthread_exit(NULL);
    }

}

void read_from_logger_msg_queue(void)
{
    char recv_buffer[MSG_MAX_LEN];
    memset(recv_buffer, '\0', sizeof(recv_buffer));

    int msg_priority;

    int num_recv_bytes;
    while ((num_recv_bytes = mq_receive(logger_mq_handle, (char
*)&recv_buffer,
                                    MSG_QUEUE_MAX_MSG_SIZE,
&msg_priority)) != -1)
    {
        if (num_recv_bytes < 0)
        {
            perror("mq_receive failed");
            return;
        }

#if 0
        printf("Message received: %s, msg_src: %s, message level: %s\n",
            (((struct _logger_msg_struct_ *)&recv_buffer)->message),
```

```
            (((struct _logger_msg_struct_ *)&recv_buffer)-
>logger_msg_src_id),
            (((struct _logger_msg_struct_ *)&recv_buffer)-
>logger_msg_level));
#endif

        time_t tval = time(NULL);
        struct tm *cur_time = localtime(&tval);

        char timestamp_str[32];
        memset(timestamp_str, '\0', sizeof(timestamp_str));

        sprintf(timestamp_str, "%02d:%02d:%02d", cur_time->tm_hour,
cur_time->tm_min, cur_time->tm_sec);

        char msg_to_write[LOG_MSG_PAYLOAD_SIZE];
        memset(msg_to_write, '\0', sizeof(msg_to_write));

        sprintf(msg_to_write, "Timestamp: %s | Message_Src: %s |
Message_Type: %s | Message: %s\n",
            timestamp_str, (((struct _logger_msg_struct_ *)&recv_buffer)-
>logger_msg_src_id),
            (((struct _logger_msg_struct_ *)&recv_buffer)-
>logger_msg_level),
            (((struct _logger_msg_struct_ *)&recv_buffer)->message));

        printf("Message to write: %s\n", msg_to_write);
        int num_written_bytes = write(logger_fd, msg_to_write,
strlen(msg_to_write));
    }

}

void sig_handler(int sig_num)
{
    char buffer[MSG_BUFF_MAX_LEN];
    memset(buffer, '\0', sizeof(buffer));

    if (sig_num == SIGINT || sig_num == SIGUSR1)
    {
        if (sig_num == SIGINT)
            printf("Caught signal %s in logger task\n", "SIGINT");
        else if (sig_num == SIGUSR1)
            printf("Caught signal %s in logger task\n", "SIGKILL");

        g_sig_kill_logger_thread = 1;
        g_sig_kill_sock_hb_thread = 1;

        //pthread_join(sensor_thread_id, NULL);
        //pthread_join(socket_thread_id, NULL);
        //pthread_join(socket_hb_thread_id, NULL);

        mq_close(logger_mq_handle);

        exit(0);
    }
}

void logger_task_exit(void)
{
```

```
    int mq_close_status = mq_close(logger_mq_handle);
    if (mq_close_status == -1)
        perror("Logger message queue close failed");

    if (logger_fd)
        close(logger_fd);

}
/*************************************************************************
*************
*
* FileName        :    temp_unit_tests.c
* Description     :    This file contains necessary  test functions for
temperature task.

* File Author Name:    Sridhar Pavithrapu
* Tools used      :    gcc, gedit, cmocka
* References      :    None
*
*************************************************************************
**************/

/* Headers Section */
#include <math.h>
#include <stdlib.h>
#include <stdarg.h>
#include <stddef.h>
#include <setjmp.h>
#include <cmocka.h>
#include "logger_task.h"

/* Macros section */

/**
*   @brief :  test function for checking write to message queue
*
*
*   @param  state   A pointer to the state
*
*   @return None
*/
void test_write_message_queue(void **state){

      /* Test case for checking write to message queue */
      char buffer[MSG_MAX_LEN] = "This is a test case";
      int return_value = write_test_msg_to_logger(buffer);
      assert_int_equal(return_value, 0);

}

/**
*   @brief :  test function for checking read to message queue
*
*
*   @param  state   A pointer to the state
*
*   @return None
*/
void test_read_message_queue(void **state){
```

```
	/* Test case for checking read to message queue */
	char buffer[MSG_MAX_LEN] = "This is a test case";
	char buffer_output[MSG_MAX_LEN];
	int return_value = write_test_msg_to_logger(buffer);
	if(return_value == 0){
		read_test_msg_to_logger(buffer_output);
		assert_string_equal(buffer,buffer_output);
	}
	else{
		assert_int_equal(return_value, 0);
	}
}

/**
*   @brief : main function for all DLL test cases
*
*
*   @return Pass and Fail test cases
*/
int main(int argc, char **argv){

	logger_task_init();

	/* Calling all DLL test case functions */
	const struct CMUnitTest tests[] = {
		cmocka_unit_test(test_write_message_queue),
		cmocka_unit_test(test_read_message_queue),

	};

	return cmocka_run_group_tests(tests, NULL, NULL);
}/*************************************************************************
**
* Author:       Pavan Dhareshwar & Sridhar Pavithrapu
* Date:         03/07/2018
* File:         light_sensor.h
* Description:  Header file containing the macros, structs/enums, globals
                and function prototypes for source file light_sensor.c
*************************************************************************
/

#ifndef _SOCKET_TASK_H_
#define _SOCKET_TASK_H_

/*---------------------------------- INCLUDES ----------------------------
----*/
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>

#include <unistd.h>
#include <pthread.h>
#include <signal.h>

#include <sys/socket.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
```

```
#include <netinet/in.h>
#include <arpa/inet.h>

#include <mqueue.h>

/*---------------------------------- INCLUDES ----------------------------
----*/

/*----------------------------------- MACROS -----------------------------
----*/
#define SERVER_PORT_NUM                         8500
#define SERVER_LISTEN_QUEUE_SIZE          100

#define TEMPERATURE_TASK_PORT_NUM                 8081
#define TEMPERATURE_TASK_QUEUE_SIZE               100

#define LIGHT_TASK_PORT_NUM                8086
#define LIGHT_TASK_QUEUE_SIZE                   100

#define SENSOR_TASK_SOCK_IP_ADDR            "127.0.0.1"

#define BUFF_SIZE                           1024

#define SOCK_REQ_MSG_API_MSG_LEN            64

#define MSG_MAX_LEN                         128
#define MSG_BUFF_MAX_LEN                    1024

#define MSG_QUEUE_NAME                      "/logger_task_mq"
#define MSG_QUEUE_MAX_NUM_MSGS              5
#define MSG_QUEUE_MAX_MSG_SIZE              1024

#define SOCKET_HB_PORT_NUM                  8670
#define SOCKET_HB_LISTEN_QUEUE_SIZE         10

#define MSG_TYPE_TEMP_DATA                  0
#define MSG_TYPE_LUX_DATA                   1
#define MSG_TYPE_SOCK_DATA                  2
#define MSG_TYPE_MAIN_DATA                  3

#define LOGGER_ATTR_LEN                     32

/*----------------------------------- MACROS -----------------------------
----*/

/*---------------------------------- GLOBALS -----------------------------
----*/
int server_sockfd, temp_sockfd, light_sockfd;
struct sockaddr_in server_addr, temp_sock_addr, light_sock_addr;
pthread_t socket_thread_id, socket_hb_thread_id;

sig_atomic_t g_sig_kill_sock_thread, g_sig_kill_sock_hb_thread;

mqd_t logger_mq_handle;

int socket_task_initialized;
/*---------------------------------- GLOBALS -----------------------------
----*/
```

```
/*---------------------------- STRUCTURES/ENUMERATIONS ------------------
----*/
enum _req_recipient_
{
    REQ_RECP_TEMP_TASK,
    REQ_RECP_LIGHT_TASK
};

struct _socket_req_msg_struct_
{
    char req_api_msg[SOCK_REQ_MSG_API_MSG_LEN];
    enum _req_recipient_ req_recipient;
    int params;
};

struct _logger_msg_struct_
{
    char message[MSG_MAX_LEN];
    char logger_msg_src_id[LOGGER_ATTR_LEN];
    char logger_msg_level[LOGGER_ATTR_LEN];
};

/*---------------------------- STRUCTURES/ENUMERATIONS ------------------
----*/

/*---------------------------- FUNCTION PROTOTYPES ----------------------
----*/

/**
 *  @brief Initialize server socket
 *
 *  For an external application to communicate with the system, the
socket
 *  task creates and listens on a socket for messages exposed to the
external
 *  application. This function creates, binds and makes the socket task
listen
 *  on this socket for messages from external application
 *
 *  @param sock_addr_struct        : sockaddr_in structre pointer
 *  @param port_num                : port number associated with the
socket
 *  @param listen_queue_size       : backlog argument for listen system
call
 *
 *  @return void
 *
 */
void initialize_server_socket(struct sockaddr_in *sock_addr_struct,
                              int port_num, int listen_queue_size);

/**
 *  @brief Initialize sensor interface socket
 *
 *  For the socket task to forward the request from the external
application, the
 *  socket task creates and uses sockets with temperature and light task
to send
 *  the requests and get the response back from the respective sensor
task. This
```

```
 *  function creates and initializes the socket listen on this socket on
the
 *  socket task side for communication with the respective sensor task.
 *
 *  @param sock_fd                 : pointer socket file descriptor
 *  @param sock_addr_struct        : sockaddr_in structure pointer
 *  @param port_num                : port number associated with the
socket
 *
 *  @return void
 *
 */
void initialize_sensor_task_socket(int *sock_fd, struct sockaddr_in
*sock_addr_struct,
                                   int port_num);

/**
 *  @brief Create logger and hearbeat socket threads for logger task
 *
 *  The socket task is made multi-threaded with
 *     1. socket thread responsible for communicating with external
application
 *        and handling request-response for external application
 *     2. socket heartbeat responsible for communicating with main task,
 *        to log heartbeat every time its requested by main task.
 *
 *  @param void
 *
 *  @return 0  : thread creation success
 *          -1 : thread creation failed
 *
 */
int create_threads(void);

/**
 *  @brief Entry point and executing entity for socket thread
 *
 *  The socket thread starts execution by invoking this
function(start_routine)
 *
 *  @param arg : argument to start_routine
 *
 *  @return void
 *
 */
void *socket_thread_func(void *args);

/**
 *  @brief Entry point and executing entity for socket thread
 *
 *  The socket thread for heartbeat starts execution by invoking this
function(start_routine)
 *
 *  @param arg : argument to start_routine
 *
 *  @return void
 *
 */
void *socket_hb_thread_func(void *arg);
```

```
/**
 *  @brief Create the socket and initialize
 *
 *  This function create the socket for the given socket id.
 *
 *  @param sock_fd              : socket file descriptor
 *         server_addr_struct   : server address of the socket
 *         port_num             : port number in which the socket is
communicating
 *         listen_qsize         : number of connections the socket is
accepting
 *
 *  @return void
*/
void init_sock(int *sock_fd, struct sockaddr_in *server_addr_struct,
               int port_num, int listen_qsize);

/**
 *  @brief Log the socket task request
 *
 *  This function writes the socket task request to logger message queue
 *
 *  @param req_msg     : pointer to request message string
 *
 *  @return void
*/
void log_req_msg(char *req_msg);

/**
 *  @brief Signal handler for temperature task
 *
 *  This function handles the reception of SIGKILL and SIGINT signal to
the
 *  temperature task and terminates all the threads, closes the I2C file
descriptor
 *  and logger message queue handle and exits.
 *
 *  @param sig_num              : signal number
 *
 *  @return void
*/

void sig_handler(int sig_num);
/*---------------------------- FUNCTION PROTOTYPES ----------------------
----*/

#endif // _SOCKET_TASK_H_
#include "socket_task.h"

int main(void)
{
    socket_task_initialized = 0;

    /* Creating a socket that is exposed to the external application */
    initialize_server_socket(&server_addr, SERVER_PORT_NUM,
SERVER_LISTEN_QUEUE_SIZE);

    initialize_sensor_task_socket(&temp_sockfd, &temp_sock_addr,
TEMPERATURE_TASK_PORT_NUM);
```

```
    if (connect(temp_sockfd, (struct sockaddr *)&temp_sock_addr,
sizeof(temp_sock_addr)) < 0)
    {
        perror("connect failed");
        printf("\nConnection Failed for temperature task \n");
        return -1;
    }

    initialize_sensor_task_socket(&light_sockfd, &light_sock_addr,
LIGHT_TASK_PORT_NUM);

    if (connect(light_sockfd, (struct sockaddr *)&light_sock_addr,
sizeof(light_sock_addr)) < 0)
    {
        perror("connect failed");
        printf("\nConnection Failed for light task \n");
        return -1;
    }

    int thread_create_status = create_threads();
    if (thread_create_status)
    {
        printf("Thread creation failed\n");
    }
    else
    {
        printf("Thread creation success\n");
    }

    socket_task_initialized = 1;

    if (signal(SIGINT, sig_handler) == SIG_ERR)
        printf("SigHandler setup for SIGINT failed\n");

    if (signal(SIGUSR1, sig_handler) == SIG_ERR)
        printf("SigHandler setup for SIGKILL failed\n");

    g_sig_kill_sock_thread = 0;
    g_sig_kill_sock_hb_thread = 0;

    pthread_join(socket_thread_id, NULL);
    pthread_join(socket_hb_thread_id, NULL);

      return 0;
}

void initialize_server_socket(struct sockaddr_in *sock_addr_struct,
                              int port_num, int listen_queue_size)
{
      /* Create server socket */
      if ((server_sockfd = socket(AF_INET, SOCK_STREAM, 0)) == 0)
      {
            perror("socket failed");
            exit(EXIT_FAILURE);
      }

    int option = 1;
    if(setsockopt(server_sockfd, SOL_SOCKET, (SO_REUSEPORT |
SO_REUSEADDR),
                    (char*)&option,sizeof(option)) < 0)
```

```
    {
        printf("setsockopt failed\n");
        close(server_sockfd);
        exit(EXIT_FAILURE);
    }

    sock_addr_struct->sin_family = AF_INET;
      sock_addr_struct->sin_addr.s_addr = INADDR_ANY;
      sock_addr_struct->sin_port = htons(port_num);

      if (bind(server_sockfd, (struct sockaddr *)sock_addr_struct,
                                        sizeof(struct sockaddr_in))<0)
      {
            perror("bind failed");
            exit(EXIT_FAILURE);
      }

      if (listen(server_sockfd, listen_queue_size) < 0)
      {
            perror("listen");
            exit(EXIT_FAILURE);
      }
}

void initialize_sensor_task_socket(int *sock_fd, struct sockaddr_in
*sock_addr_struct, int port_num)
{
      memset(sock_addr_struct, '0', sizeof(struct sockaddr_in));
      sock_addr_struct->sin_family = AF_INET;
      sock_addr_struct->sin_port = htons(port_num);

    if ((*sock_fd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    {
        perror("socket creation failed");
        exit(EXIT_FAILURE);
    }

    int option = 1;
    if(setsockopt(*sock_fd, SOL_SOCKET, (SO_REUSEPORT | SO_REUSEADDR),
                (char*)&option,sizeof(option)) < 0)
    {
        perror("setsockopt failed");
        close(*sock_fd);
        exit(EXIT_FAILURE);
    }

    // Convert IPv4 and IPv6 addresses from text to binary form
    if(inet_pton(AF_INET, SENSOR_TASK_SOCK_IP_ADDR, &(sock_addr_struct-
>sin_addr))<=0)
    {
        perror("inet_pton failed");
        printf("\nInvalid address/ Address not supported for temperature
task\n");
        exit(EXIT_FAILURE);
    }
}

int create_threads(void)
{
```

```
    int sock_t_creat_ret_val = pthread_create(&socket_thread_id, NULL,
&socket_thread_func, NULL);
    if (sock_t_creat_ret_val)
    {
        perror("Socket thread creation failed");
        return -1;
    }

    int sock_hb_t_creat_ret_val = pthread_create(&socket_hb_thread_id,
NULL, &socket_hb_thread_func, NULL);
    if (sock_hb_t_creat_ret_val)
    {
        perror("Socket heartbeat thread creation failed");
        return -1;
    }

    return 0;
}

void *socket_thread_func(void *args)
{
      int serv_addr_len = sizeof(server_addr);
      char buffer[BUFF_SIZE];

    int accept_conn_id;
    /* Wait for request from external application */
    if ((accept_conn_id = accept(server_sockfd, (struct sockaddr
*)&server_addr,
                    (socklen_t *)&serv_addr_len)) < 0)
    {
        perror("accept");
    }

    char recv_buffer[BUFF_SIZE];
      while(!g_sig_kill_sock_thread)
    {
        memset(recv_buffer, '\0', sizeof(recv_buffer));
        int num_recv_bytes = recv(accept_conn_id, recv_buffer,
sizeof(recv_buffer), 0);
        if (num_recv_bytes < 0)
        {
            printf("recv failed in socket task\n");
            perror("recv failed");
        }
        else
        {

            if (*(((struct _socket_req_msg_struct_ *)&recv_buffer)-
>req_api_msg) != '\0')
            {
                printf("Message req api: %s, req recp: %s, req api
params: %d\n",
                        (((struct _socket_req_msg_struct_
*)&recv_buffer)->req_api_msg),
                        ((((struct _socket_req_msg_struct_
*)&recv_buffer)->req_recipient)
                         == REQ_RECP_TEMP_TASK ? "Temp Task" : "Light
Task"),
                        (((struct _socket_req_msg_struct_
*)&recv_buffer)->params));
```

```
                //log_req_msg((((struct _socket_req_msg_struct_
*)&recv_buffer)->req_api_msg));

                size_t sent_bytes;
                memset(buffer, '\0', sizeof(buffer));
                //strncpy(buffer, "Hello!", strlen("Hello!"));

                uint8_t data = (uint8_t)(((struct _socket_req_msg_struct_
*)&recv_buffer)->params);

                if ((((struct _socket_req_msg_struct_ *)&recv_buffer)-
>req_recipient)
                        == REQ_RECP_TEMP_TASK)
                {
                    printf("Sending request to temperature task\n");
                    sent_bytes = send(temp_sockfd, recv_buffer,
sizeof(recv_buffer), 0);

                    ssize_t num_recv_bytes = recv(temp_sockfd, buffer,
sizeof(buffer), 0);
                    if (num_recv_bytes < 0)
                        perror("recv failed");
                    //strncpy(buffer, "Hello!", strlen("Hello!"));
                    sent_bytes = send(accept_conn_id, buffer,
strlen(buffer), 0);
                }
                else
                {
                    printf("Sending request to light task\n");
                    sent_bytes = send(light_sockfd, recv_buffer,
sizeof(recv_buffer), 0);

                    ssize_t num_recv_bytes = recv(light_sockfd, buffer,
sizeof(buffer), 0);
                    if (num_recv_bytes < 0)
                        perror("recv failed");
                    printf("Received %d bytes in socket task\n",
num_recv_bytes);

                    sent_bytes = send(accept_conn_id, buffer,
num_recv_bytes, 0);
                    if (sent_bytes < 0)
                        perror("send failed");
                    else
                        printf("Sent %d bytes from socket task to test
app\n", sent_bytes);
                }

                //sent_bytes = send(accept_conn_id, buffer,
sizeof(buffer), 0 );
            }
        }
    }

    pthread_exit(NULL);
}

void *socket_hb_thread_func(void *arg)
{
```

```
    int sock_hb_fd;
    struct sockaddr_in sock_hb_address;
    int sock_hb_addr_len = sizeof(sock_hb_address);

    init_sock(&sock_hb_fd, &sock_hb_address, SOCKET_HB_PORT_NUM,
SOCKET_HB_LISTEN_QUEUE_SIZE);

    int accept_conn_id;
    printf("Waiting for request...\n");
    if ((accept_conn_id = accept(sock_hb_fd, (struct sockaddr
*)&sock_hb_address,
                    (socklen_t*)&sock_hb_addr_len)) < 0)
    {
        perror("accept failed");
        //pthread_exit(NULL);
    }

    char recv_buffer[MSG_BUFF_MAX_LEN];
    char send_buffer[] = "Alive";

    while (!g_sig_kill_sock_hb_thread)
    {
        memset(recv_buffer, '\0', sizeof(recv_buffer));

        size_t num_read_bytes = read(accept_conn_id, &recv_buffer,
sizeof(recv_buffer));

        if (!strcmp(recv_buffer, "heartbeat"))
        {
            ssize_t num_sent_bytes = send(accept_conn_id, send_buffer,
strlen(send_buffer), 0);
            if (num_sent_bytes < 0)
                perror("send failed");
        }
        else if (!strcmp(recv_buffer, "startup_check"))
        {
            /* For the sake of start-up check, because we have the
temperature sensor initialized
            ** by the time this thread is spawned. So we perform a
"get_temp_data" call to see if
            ** everything is working fine */
            if (socket_task_initialized == 1)
                strcpy(send_buffer, "Initialized");
            else
                strcpy(send_buffer, "Uninitialized");

            ssize_t num_sent_bytes = send(accept_conn_id, send_buffer,
strlen(send_buffer), 0);
            if (num_sent_bytes < 0)
                perror("send failed");
        }
    }

    pthread_exit(NULL);

}

void init_sock(int *sock_fd, struct sockaddr_in *server_addr_struct,
               int port_num, int listen_qsize)
{
```

```
    int serv_addr_len = sizeof(struct sockaddr_in);

    /* Create the socket */
    if ((*sock_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0)
    {
        perror("socket creation failed");
        pthread_exit(NULL); // Change these return values from
pthread_exit
    }

    int option = 1;
    if(setsockopt(*sock_fd, SOL_SOCKET, (SO_REUSEPORT | SO_REUSEADDR),
(void *)&option, sizeof(option)) < 0)
    {
        perror("setsockopt failed");
        pthread_exit(NULL);
    }

    server_addr_struct->sin_family = AF_INET;
    server_addr_struct->sin_addr.s_addr = INADDR_ANY;
    server_addr_struct->sin_port = htons(port_num);

    if (bind(*sock_fd, (struct sockaddr *)server_addr_struct,
                                            sizeof(struct
sockaddr_in))<0)
    {
        perror("bind failed");
        pthread_exit(NULL);
    }

    if (listen(*sock_fd, listen_qsize) < 0)
    {
        perror("listen failed");
        pthread_exit(NULL);
    }

}

void log_req_msg(char *req_msg)
{
    int msg_priority;

    /* Set the message queue attributes */
    struct mq_attr logger_mq_attr = { .mq_flags = 0,
                                      .mq_maxmsg =
MSG_QUEUE_MAX_NUM_MSGS,  // Max number of messages on queue
                                      .mq_msgsize =
MSG_QUEUE_MAX_MSG_SIZE  // Max. message size
                                    };

    logger_mq_handle = mq_open(MSG_QUEUE_NAME, O_RDWR, S_IRWXU,
&logger_mq_attr);

    char sock_data_msg[MSG_MAX_LEN];
    memset(sock_data_msg, '\0', sizeof(sock_data_msg));

    sprintf(sock_data_msg, "Req Msg: %s", req_msg);

    struct _logger_msg_struct_ logger_msg;
    memset(&logger_msg, '\0', sizeof(logger_msg));
```

```
    strncpy(logger_msg.logger_msg_src_id, "Socket", strlen("Socket"));
    logger_msg.logger_msg_src_id[strlen("Socket")] = '\0';
    strncpy(logger_msg.logger_msg_level, "Info", strlen("Info"));
    logger_msg.logger_msg_level[strlen("Info")] = '\0';

    msg_priority = 1;
    int num_sent_bytes = mq_send(logger_mq_handle, (char *)&logger_msg,
                          sizeof(logger_msg), msg_priority);
    if (num_sent_bytes < 0)
        perror("mq_send failed");
}

void sig_handler(int sig_num)
{
    char buffer[MSG_BUFF_MAX_LEN];
    memset(buffer, '\0', sizeof(buffer));

    if (sig_num == SIGINT || sig_num == SIGUSR1)
    {
        if (sig_num == SIGINT)
            printf("Caught signal %s in socket task\n", "SIGINT");
        else if (sig_num == SIGUSR1)
            printf("Caught signal %s in socket task\n", "SIGKILL");

        g_sig_kill_sock_thread = 1;
        g_sig_kill_sock_hb_thread = 1;

        //pthread_join(sensor_thread_id, NULL);
        //pthread_join(socket_thread_id, NULL);
        //pthread_join(socket_hb_thread_id, NULL);

        mq_close(logger_mq_handle);

        exit(0);
    }
}
/*************************************************************************
*
* Author:       Pavan Dhareshwar & Sridhar Pavithrapu
* Date:         03/07/2018
* File:         light_sensor.c
* Description:  Source file describing the functionality and
implementation
*               of light sensor task.
*************************************************************************
/

#include <stdio.h>
#include <stdlib.h>

#include "light_sensor.h"

int main(void){

    light_sensor_initialized = 0;

    int init_ret_val = light_sensor_init();
    if (init_ret_val == -1)
    {
        printf("Light sensor init failed\n");
```

```
        exit(1);
    }

    printf("Creating threads\n");
    int thread_create_status = create_threads();
    if (thread_create_status)
    {
        printf("Thread creation failed\n");
    }
    else
    {
        printf("Thread creation success\n");
    }

    if (signal(SIGINT, sig_handler) == SIG_ERR)
        printf("SigHandler setup for SIGINT failed\n");

    if (signal(SIGUSR1, sig_handler) == SIG_ERR)
        printf("SigHandler setup for SIGKILL failed\n");

    g_sig_kill_sensor_thread = 0;
    g_sig_kill_sock_thread = 0;
    g_sig_kill_sock_hb_thread = 0;

    pthread_join(sensor_thread_id, NULL);
    pthread_join(socket_thread_id, NULL);
    pthread_join(socket_hb_thread_id, NULL);

     light_sensor_exit();

     return 0;
}

int light_sensor_init(void)
{
     /* Open the i2c bus for read and write operation */
    printf("Opening i2c bus %s\n", I2C_DEV_NAME);
    if ((i2c_light_sensor_fd = open(I2C_DEV_NAME,O_RDWR)) < 0) {
           perror("Failed to open i2c bus.");
           /* ERROR HANDLING; you can check errno to see what went wrong
*/
           return -1;
     }

     if (ioctl(i2c_light_sensor_fd,I2C_SLAVE,I2C_SLAVE_ADDR) < 0) {
           perror("Failed to acquire bus access and/or talk to slave.");
           /* ERROR HANDLING; you can check errno to see what went wrong
*/
           return -1;
     }

    printf("Powering on light sensor\n");
    /* Power on the APDS-9301 device */
    power_on_light_sensor();
    printf("Powered on light sensor\n");

    if (light_sensor_initialized == 0)
        light_sensor_initialized = 1;

    return 0;
```

```
}

void power_on_light_sensor(void)
{
    int cmd_ctrl_reg_val = I2C_LIGHT_SENSOR_CMD_CTRL_REG;

      int ctrl_reg_val = I2C_LIGHT_SENSOR_CTRL_REG_VAL;

    write_light_sensor_reg(cmd_ctrl_reg_val, ctrl_reg_val);

      cmd_ctrl_reg_val = I2C_LIGHT_SENSOR_CMD_TIM_REG;
      ctrl_reg_val = 0X10;
      write_light_sensor_reg(cmd_ctrl_reg_val, ctrl_reg_val);

}

int create_threads(void)
{
    int sens_t_creat_ret_val = pthread_create(&sensor_thread_id, NULL,
&sensor_thread_func, NULL);
    if (sens_t_creat_ret_val)
    {
        perror("Sensor thread creation failed");
        return -1;
    }

    int sock_t_creat_ret_val = pthread_create(&socket_thread_id, NULL,
&socket_thread_func, NULL);
    if (sock_t_creat_ret_val)
    {
        perror("Socket thread creation failed");
        return -1;
    }

      int sock_hb_t_creat_ret_val = pthread_create(&socket_hb_thread_id,
NULL, &socket_hb_thread_func, NULL);
    if (sock_hb_t_creat_ret_val)
    {
        perror("Socket heartbeat thread creation failed");
        return -1;
    }

    return 0;
}

void init_light_socket(struct sockaddr_in *sock_addr_struct)
{

    /* Create the socket */
    if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0)
    {
        perror("socket creation failed");
        pthread_exit(NULL); // Change these return values from
pthread_exit
    }

    int option = 1;
    if(setsockopt(server_fd, SOL_SOCKET, (SO_REUSEPORT | SO_REUSEADDR),
                (void *)&option, sizeof(option)) < 0)
    {
```

```
        perror("setsockopt failed");
        pthread_exit(NULL);
    }

    sock_addr_struct->sin_family = AF_INET;
    sock_addr_struct->sin_addr.s_addr = INADDR_ANY;
    sock_addr_struct->sin_port = htons(LIGHT_SENSOR_SERVER_PORT_NUM);

    if (bind(server_fd, (struct sockaddr *)sock_addr_struct,
                                            sizeof(struct
sockaddr_in))<0)
    {
        perror("bind failed");
        pthread_exit(NULL);
    }

    if (listen(server_fd, LIGHT_SENSOR_LISTEN_QUEUE_SIZE) < 0)
    {
        perror("listen failed");
        pthread_exit(NULL);
    }
}

void *socket_thread_func(void *arg)
{
    struct sockaddr_in server_address;
    int serv_addr_len = sizeof(server_address);

    init_light_socket(&server_address);

    char recv_buffer[MSG_BUFF_MAX_LEN];

    int accept_conn_id;
    printf("Waiting for request...\n");
    if ((accept_conn_id = accept(server_fd, (struct sockaddr
*)&server_address,
                    (socklen_t*)&serv_addr_len)) < 0)
    {
        perror("accept failed");
        //pthread_exit(NULL);
    }

    while (!g_sig_kill_sock_thread)
    {
        memset(recv_buffer, '\0', sizeof(recv_buffer));

        size_t num_read_bytes = read(accept_conn_id, &recv_buffer,
sizeof(recv_buffer));

        printf("[Light_Task] Message req api: %s, req recp: %s, req api
params: %d\n",
                (((struct _socket_req_msg_struct_ *)&recv_buffer)-
>req_api_msg),
                ((((struct _socket_req_msg_struct_ *)&recv_buffer)-
>req_recipient)
                 == REQ_RECP_TEMP_TASK ? "Temp Task" : "Light Task"),
                (((struct _socket_req_msg_struct_ *)&recv_buffer)-
>param));

        char light_sensor_rsp_msg[64];
```

```
        if (!strcmp((((struct _socket_req_msg_struct_ *)&recv_buffer)-
>req_api_msg), "get_lux_data"))
        {
            float lux_data = get_lux_data();
            memset(light_sensor_rsp_msg, '\0',
sizeof(light_sensor_rsp_msg));

            sprintf(light_sensor_rsp_msg, "Lux Data: %3.2f", lux_data);

            ssize_t num_sent_bytes = send(accept_conn_id,
light_sensor_rsp_msg, strlen(light_sensor_rsp_msg), 0);
            if (num_sent_bytes < 0)
                perror("send failed");

        }
        else if (!strcmp((((struct _socket_req_msg_struct_
*)&recv_buffer)->req_api_msg), "get_light_sensor_id"))
        {
            uint8_t light_sen_id_reg_val = read_id_reg();
            printf("id reg val : %d\n", light_sen_id_reg_val);

            memset(light_sensor_rsp_msg, '\0',
sizeof(light_sensor_rsp_msg));

            sprintf(light_sensor_rsp_msg, "ID reg val: 0x%x",
light_sen_id_reg_val);

            ssize_t num_sent_bytes = send(accept_conn_id,
light_sensor_rsp_msg, strlen(light_sensor_rsp_msg), 0);
            if (num_sent_bytes < 0)
                perror("send failed");

        }
        else if (!strcmp((((struct _socket_req_msg_struct_
*)&recv_buffer)->req_api_msg), "get_light_sensor_ctrl_reg"))
        {
            uint8_t light_sen_ctrl_reg_val = read_ctrl_reg();
            printf("ctrl reg val : %d\n", light_sen_ctrl_reg_val);

            memset(light_sensor_rsp_msg, '\0',
sizeof(light_sensor_rsp_msg));

            sprintf(light_sensor_rsp_msg, "Ctrl reg val: 0x%x",
light_sen_ctrl_reg_val);

            ssize_t num_sent_bytes = send(accept_conn_id,
light_sensor_rsp_msg, strlen(light_sensor_rsp_msg), 0);
            if (num_sent_bytes < 0)
                perror("send failed");
        }
        else if (!strcmp((((struct _socket_req_msg_struct_
*)&recv_buffer)->req_api_msg), "set_light_sensor_ctrl_reg"))
        {
            //if (((struct _socket_req_msg_struct_ *)&recv_buffer)-
>ptr_param_list != NULL)
            {
                    #if 0
                        uint8_t cmd_ctrl_reg_val = *(uint8_t
*)(((struct _socket_req_msg_struct_ *)&recv_buffer)->ptr_param_list);
                    #endif
```

```
                    uint8_t cmd_ctrl_reg_val = (uint8_t)(((struct
_socket_req_msg_struct_ *)&recv_buffer)->param);
                if (write_ctrl_reg(cmd_ctrl_reg_val) == 0)
                {
                    memset(light_sensor_rsp_msg, '\0',
sizeof(light_sensor_rsp_msg));

                    sprintf(light_sensor_rsp_msg, "OK");

                    ssize_t num_sent_bytes = send(accept_conn_id,
light_sensor_rsp_msg, strlen(light_sensor_rsp_msg), 0);
                    if (num_sent_bytes < 0)
                        perror("send failed");
                }
            }
        }
        else if (!strcmp((((struct _socket_req_msg_struct_
*)&recv_buffer)->req_api_msg), "get_light_sensor_tim_reg"))
        {
            uint8_t light_sen_tim_reg_val = read_timing_reg();
            printf("tim reg val : %d\n", light_sen_tim_reg_val);

            memset(light_sensor_rsp_msg, '\0',
sizeof(light_sensor_rsp_msg));

            sprintf(light_sensor_rsp_msg, "Ctrl reg val: 0x%x",
light_sen_tim_reg_val);

            ssize_t num_sent_bytes = send(accept_conn_id,
light_sensor_rsp_msg, strlen(light_sensor_rsp_msg), 0);
            if (num_sent_bytes < 0)
                perror("send failed");
        }
           #if 0
        else if (!strcmp((((struct _socket_req_msg_struct_
*)&recv_buffer)->req_api_msg), "set_light_sensor_tim_reg"))
        {
            if (((struct _socket_req_msg_struct_ *)&recv_buffer)-
>ptr_param_list != NULL)
            {
                struct _light_sensor_tim_params light_sen_tim_params =
*(struct _light_sensor_tim_params *)(((struct _socket_req_msg_struct_
*)&recv_buffer)->ptr_param_list);
                uint8_t cmd_tim_reg_val =
light_sen_tim_params.tim_reg_val;
                uint8_t cmd_tim_field_to_set =
light_sen_tim_params.tim_reg_field_to_set;
                uint8_t cmd_tim_field_val =
light_sen_tim_params.tim_reg_field_val;

                if (write_timing_reg(cmd_tim_reg_val,
cmd_tim_field_to_set, cmd_tim_field_val) == 0)
                {
                    memset(light_sensor_rsp_msg, '\0',
sizeof(light_sensor_rsp_msg));

                    sprintf(light_sensor_rsp_msg, "OK");
                    ssize_t num_sent_bytes = send(accept_conn_id,
light_sensor_rsp_msg, strlen(light_sensor_rsp_msg), 0);
                    if (num_sent_bytes < 0)
```

```
                        perror("send failed");
                    }
                }
            }
            #endif
            else if (!strcmp((((struct _socket_req_msg_struct_
*)&recv_buffer)->req_api_msg), "set_light_sensor_integration_time"))
            {
                //if (((struct _socket_req_msg_struct_ *)&recv_buffer)-
>ptr_param_list != NULL)
                {

                    uint8_t cmd_tim_reg_val = read_timing_reg();
                    uint8_t cmd_tim_field_val = (uint8_t)(((struct
_socket_req_msg_struct_ *)&recv_buffer)->param);

                    if (write_timing_reg(cmd_tim_reg_val, 0x3,
cmd_tim_field_val) == 0)
                    {
                        memset(light_sensor_rsp_msg, '\0',
sizeof(light_sensor_rsp_msg));

                        sprintf(light_sensor_rsp_msg, "OK");
                        ssize_t num_sent_bytes = send(accept_conn_id,
light_sensor_rsp_msg, strlen(light_sensor_rsp_msg), 0);
                        if (num_sent_bytes < 0)
                            perror("send failed");
                        }
                }
            }
            else if (!strcmp((((struct _socket_req_msg_struct_
*)&recv_buffer)->req_api_msg), "set_light_sensor_gain"))
            {
                //if (((struct _socket_req_msg_struct_ *)&recv_buffer)-
>ptr_param_list != NULL)
                {

                    uint8_t cmd_tim_reg_val = read_timing_reg();
                    uint8_t cmd_tim_field_val = (uint8_t)(((struct
_socket_req_msg_struct_ *)&recv_buffer)->param);

                    if (write_timing_reg(cmd_tim_reg_val, 0x10,
cmd_tim_field_val) == 0)
                    {
                        memset(light_sensor_rsp_msg, '\0',
sizeof(light_sensor_rsp_msg));

                        sprintf(light_sensor_rsp_msg, "OK");
                        ssize_t num_sent_bytes = send(accept_conn_id,
light_sensor_rsp_msg, strlen(light_sensor_rsp_msg), 0);
                        if (num_sent_bytes < 0)
                            perror("send failed");
                        }
                }
            }
            else if (!strcmp((((struct _socket_req_msg_struct_
*)&recv_buffer)->req_api_msg), "set_interrupt_low_threshold"))
            {
                //if (((struct _socket_req_msg_struct_ *)&recv_buffer)-
>ptr_param_list != NULL)
```

```
            {


                uint16_t low_thresh = (uint16_t)(((struct
_socket_req_msg_struct_ *)&recv_buffer)->param);

                write_intr_low_thresh_reg(low_thresh);

                sprintf(light_sensor_rsp_msg, "OK");
                ssize_t num_sent_bytes = send(accept_conn_id,
light_sensor_rsp_msg, strlen(light_sensor_rsp_msg), 0);
                if (num_sent_bytes < 0)
                    perror("send failed");
            }
        }
           else if (!strcmp((((struct _socket_req_msg_struct_
*)&recv_buffer)->req_api_msg), "set_interrupt_high_threshold"))
        {
            //if (((struct _socket_req_msg_struct_ *)&recv_buffer)-
>ptr_param_list != NULL)
            {

                uint16_t high_thresh = (uint16_t)(((struct
_socket_req_msg_struct_ *)&recv_buffer)->param);

                write_intr_high_thresh_reg(high_thresh);

                sprintf(light_sensor_rsp_msg, "OK");
                ssize_t num_sent_bytes = send(accept_conn_id,
light_sensor_rsp_msg, strlen(light_sensor_rsp_msg), 0);
                if (num_sent_bytes < 0)
                    perror("send failed");
            }
        }

        else if (!strcmp((((struct _socket_req_msg_struct_
*)&recv_buffer)->req_api_msg), "get_light_sensor_int_thresh_reg"))
        {
#if 0
            uint8_t cmd_thresh_low_low_reg =
I2C_LIGHT_SENSOR_CMD_THRESH_LOW_LOW_REG;
            uint8_t cmd_thresh_low_high_reg =
I2C_LIGHT_SENSOR_CMD_THRESH_LOW_HIGH_REG;
            uint8_t cmd_thresh_high_low_reg =
I2C_LIGHT_SENSOR_CMD_THRESH_HIGH_LOW_REG;
            uint8_t cmd_thresh_high_high_reg =
I2C_LIGHT_SENSOR_CMD_THRESH_HIGH_HIGH_REG;

            int8_t light_sen_thresh_low_low_reg_val =
read_light_sensor_reg(cmd_thresh_low_low_reg);
            printf("thresh low low reg val : %d\n",
light_sen_thresh_low_low_reg_val);

            int8_t light_sen_thresh_low_high_reg_val =
read_light_sensor_reg(cmd_thresh_low_high_reg);
            printf("thresh low high reg val : %d\n",
light_sen_thresh_low_high_reg_val);

            int8_t light_sen_thresh_high_low_reg_val =
read_light_sensor_reg(cmd_thresh_high_low_reg);
```

```
            printf("thresh high low reg val : %d\n",
light_sen_thresh_high_low_reg_val);

            int8_t light_sen_thresh_high_high_reg_val =
read_light_sensor_reg(cmd_thresh_high_high_reg);
            printf("thresh high high reg val : %d\n",
light_sen_thresh_high_high_reg_val);

            memset(light_sensor_rsp_msg, '\0',
sizeof(light_sensor_rsp_msg));

            struct _int_thresh_reg_struct_ int_thresh_reg_struct;
            int_thresh_reg_struct.thresh_low_low =
light_sen_thresh_low_low_reg_val;
            int_thresh_reg_struct.thresh_low_high =
light_sen_thresh_low_high_reg_val;
            int_thresh_reg_struct.thresh_high_low =
light_sen_thresh_high_low_reg_val;
            int_thresh_reg_struct.thresh_high_high =
light_sen_thresh_high_high_reg_val;
#endif
            uint16_t low_thresh, high_thresh;
            read_intr_thresh_reg(&low_thresh, &high_thresh);

            printf("low_thresh: %d, high_thresh: %d\n", low_thresh,
high_thresh);

            struct _int_thresh_reg_struct_ int_thresh_reg_struct;
            int_thresh_reg_struct.low_thresh = low_thresh;
            int_thresh_reg_struct.high_thresh = high_thresh;

            ssize_t num_sent_bytes = send(accept_conn_id,
&int_thresh_reg_struct,
                                          sizeof(struct
_int_thresh_reg_struct_), 0);
            if (num_sent_bytes < 0)
                perror("send failed");
            else
                printf("Sent %d bytes in light task\n", num_sent_bytes);
        }
#if 0
        else if (!strcmp((((struct _socket_req_msg_struct_
*)&recv_buffer)->req_api_msg), "set_light_sensor_int_thresh_reg"))
        {
            if (((struct _socket_req_msg_struct_ *)&recv_buffer)-
>ptr_param_list != NULL)
            {
                struct _int_thresh_reg_struct_ *p_int_thresh_reg_struct =
                    (struct _int_thresh_reg_struct_ *)(((struct
_socket_req_msg_struct_ *)&recv_buffer)->ptr_param_list);
      #if 0
                uint8_t cmd_thresh_low_low_reg =
I2C_LIGHT_SENSOR_CMD_THRESH_LOW_LOW_REG;
                uint8_t cmd_thresh_low_low_reg_val =
(uint8_t)p_int_thresh_reg_struct->thresh_low_low;
                write_light_sensor_reg(cmd_thresh_low_low_reg,
cmd_thresh_low_low_reg_val);

                uint8_t cmd_thresh_low_high_reg =
I2C_LIGHT_SENSOR_CMD_THRESH_LOW_HIGH_REG;
```

```
                uint8_t cmd_thresh_low_high_reg_val =
(uint8_t)p_int_thresh_reg_struct->thresh_low_high;
                write_light_sensor_reg(cmd_thresh_low_high_reg,
cmd_thresh_low_high_reg_val);

                uint8_t cmd_thresh_high_low_reg =
I2C_LIGHT_SENSOR_CMD_THRESH_HIGH_LOW_REG;
                uint8_t cmd_thresh_high_low_reg_val =
(uint8_t)p_int_thresh_reg_struct->thresh_high_low;
                write_light_sensor_reg(cmd_thresh_high_low_reg,
cmd_thresh_high_low_reg_val);

                uint8_t cmd_thresh_high_high_reg =
I2C_LIGHT_SENSOR_CMD_THRESH_HIGH_HIGH_REG;
                uint8_t cmd_thresh_high_high_reg_val =
(uint8_t)p_int_thresh_reg_struct->thresh_high_high;
                write_light_sensor_reg(cmd_thresh_high_high_reg,
cmd_thresh_high_high_reg_val);
      #endif

                uint16_t low_thresh = p_int_thresh_reg_struct-
>low_thresh;
                uint16_t high_thresh = p_int_thresh_reg_struct-
>high_thresh;

                write_intr_thresh_reg(low_thresh, high_thresh);

                sprintf(light_sensor_rsp_msg, "OK");
                ssize_t num_sent_bytes = send(accept_conn_id,
light_sensor_rsp_msg, strlen(light_sensor_rsp_msg), 0);
                if (num_sent_bytes < 0)
                    perror("send failed");
            }
        }
#endif
        else
        {
            printf("Invalid request from socket task\n");
        }
    }

    pthread_exit(NULL);
}

void *sensor_thread_func(void *arg)
{
    while (!g_sig_kill_sensor_thread)
    {
        float sensor_lux_data = get_lux_data();

        printf("Sensor lux data: %3.2f\n", sensor_lux_data);

        log_lux_data(sensor_lux_data);

        sleep(5);
    }

    pthread_exit(NULL);
}
```

```
void init_sock(int *sock_fd, struct sockaddr_in *server_addr_struct,
               int port_num, int listen_qsize)
{
    int serv_addr_len = sizeof(struct sockaddr_in);

    /* Create the socket */
    if ((*sock_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0)
    {
        perror("socket creation failed");
        pthread_exit(NULL); // Change these return values from
pthread_exit
    }

    int option = 1;
    if(setsockopt(*sock_fd, SOL_SOCKET, (SO_REUSEPORT | SO_REUSEADDR),
(void *)&option, sizeof(option)) < 0)
    {
        perror("setsockopt failed");
        pthread_exit(NULL);
    }

    server_addr_struct->sin_family = AF_INET;
    server_addr_struct->sin_addr.s_addr = INADDR_ANY;
    server_addr_struct->sin_port = htons(port_num);

    if (bind(*sock_fd, (struct sockaddr *)server_addr_struct,
                                          sizeof(struct
sockaddr_in))<0)
    {
        perror("bind failed");
        pthread_exit(NULL);
    }

    if (listen(*sock_fd, listen_qsize) < 0)
    {
        perror("listen failed");
        pthread_exit(NULL);
    }

}


void *socket_hb_thread_func(void *arg)
{
    int sock_hb_fd;
    struct sockaddr_in sock_hb_address;
    int sock_hb_addr_len = sizeof(sock_hb_address);

    init_sock(&sock_hb_fd, &sock_hb_address, SOCKET_HB_PORT_NUM,
SOCKET_HB_LISTEN_QUEUE_SIZE);


    int accept_conn_id;
    printf("Waiting for request...\n");
    if ((accept_conn_id = accept(sock_hb_fd, (struct sockaddr
*)&sock_hb_address,
                    (socklen_t*)&sock_hb_addr_len)) < 0)
    {
        perror("accept failed");
        //pthread_exit(NULL);
```

```
    }

    char recv_buffer[MSG_BUFF_MAX_LEN];
    char send_buffer[] = "Alive";

    while (!g_sig_kill_sock_hb_thread)
    {
        memset(recv_buffer, '\0', sizeof(recv_buffer));

        size_t num_read_bytes = read(accept_conn_id, &recv_buffer,
sizeof(recv_buffer));

        if (!strcmp(recv_buffer, "heartbeat"))
        {
                 ssize_t num_sent_bytes = send(accept_conn_id,
send_buffer, strlen(send_buffer), 0);
            if (num_sent_bytes < 0)
                perror("send failed");
        }
        else if (!strcmp(recv_buffer, "startup_check"))
        {
            /* For the sake of start-up check, because we have the
temperature sensor initialized
            ** by the time this thread is spawned. So we perform a
"get_temp_data" call to see if
            ** everything is working fine */
            if (light_sensor_initialized == 1)
                strcpy(send_buffer, "Initialized");
            else
                strcpy(send_buffer, "Uninitialized");

            ssize_t num_sent_bytes = send(accept_conn_id, send_buffer,
strlen(send_buffer), 0);
            if (num_sent_bytes < 0)
                perror("send failed");
        }
    }

    pthread_exit(NULL);
}

float get_lux_data(void)
{
    float sensor_lux_val = 0;

    uint16_t adc_ch0_data, adc_ch1_data;

    get_adc_channel_data(0, &adc_ch0_data);
    get_adc_channel_data(1, &adc_ch1_data);

    sensor_lux_val = calculate_lux_value(adc_ch0_data, adc_ch1_data);

    printf("Sensor lux value: %3.2f\n", sensor_lux_val);

    return sensor_lux_val;
}

void get_adc_channel_data(int channel_num, uint16_t *ch_data)
{
    if (channel_num == 0)
```

```
    {
        uint8_t cmd_data0_low_reg = I2C_LIGHT_SENSOR_CMD_DATA0LOW_REG;
        uint8_t cmd_data0_high_reg = I2C_LIGHT_SENSOR_CMD_DATA0HIGH_REG;

        int8_t ch_data_low = read_light_sensor_reg(cmd_data0_low_reg);
        //printf("data0_low : %d\n", ch_data_low);

        int8_t ch_data_high = read_light_sensor_reg(cmd_data0_high_reg);
        //printf("data0_high : %d\n", ch_data_high);

        *ch_data = ch_data_high << 8 | ch_data_low;
    }
    else if (channel_num == 1)
    {
        uint8_t cmd_data1_low_reg = I2C_LIGHT_SENSOR_CMD_DATA1LOW_REG;
        uint8_t cmd_data1_high_reg = I2C_LIGHT_SENSOR_CMD_DATA1HIGH_REG;

        int8_t ch_data_low = read_light_sensor_reg(cmd_data1_low_reg);
        //printf("data1_low : %d\n", ch_data_low);

        int8_t ch_data_high = read_light_sensor_reg(cmd_data1_high_reg);
        //printf("data1_high : %d\n", ch_data_high);

        *ch_data = ch_data_high << 8 | ch_data_low;
    }
    else
    {
        printf("Channel number %d invalid\n", channel_num);
    }
}

float calculate_lux_value(uint16_t ch0_data, uint16_t ch1_data)
{
    float sensor_lux_val = 0;

    if (ch0_data == 0 || ch1_data == 0)
        return 0;

    /* Mapping between ADC channel data and the sensor lux formula used
    **       CH1/CH0                                  Sensor lux formula
    **
    **  0 < CH1/CH0 ≤ 0.50          Sensor Lux = (0.0304 x CH0) - (0.062
x CH0 x ((CH1/CH0)^1.4))
    **  0.50 < CH1/CH0 ≤ 0.61       Sensor Lux = (0.0224 x CH0) - (0.031
x CH1)
    **  0.61 < CH1/CH0 ≤ 0.80       Sensor Lux = (0.0128 x CH0) -
(0.0153 x CH1)
    **  0.80 < CH1/CH0 ≤ 1.30       Sensor Lux = (0.00146 x CH0) -
(0.00112 x CH1)
    **  CH1/CH0>1.30                Sensor Lux = 0
    **
    */

    float adc_count_ratio = (float)(ch1_data/ch0_data);
    if ( 0 < adc_count_ratio <= 0.5)
    {
        sensor_lux_val = ((0.0304 * ch0_data) - (0.062 * ch0_data *
pow(adc_count_ratio, 1.4)));
    }
    else if (0.5 < adc_count_ratio <= 0.61)
```

```
    {
        sensor_lux_val = ((0.0224 * ch0_data) - (0.031 * ch1_data));
    }
    else if (0.61 < adc_count_ratio <= 0.8)
    {
        sensor_lux_val = ((0.0128 * ch0_data) - (0.0153 * ch1_data));
    }
    else if (0.8 < adc_count_ratio <= 1.3)
    {
        sensor_lux_val = ((0.00146 * ch0_data) - (0.00112 * ch1_data));
    }
    else if (adc_count_ratio > 1.3)
    {
        sensor_lux_val = 0;
    }

    return sensor_lux_val;
}

int write_light_sensor_reg(int cmd_reg_val, int target_reg_val)
{

    /* Write the command register to specify the following two
information
    **       1. Target register address for subsequent write operation
    **       2. If I2C write operation is a word or byte operation
    */
      if (wrapper_write(i2c_light_sensor_fd, &cmd_reg_val, 1) != 1)
      {
             perror("Failed to write to the i2c bus.");
         return -1;
    }

    if(wrapper_write(i2c_light_sensor_fd, &target_reg_val, 1) != 1){
             perror("Failed to write to the i2c bus.");
         return -1;
    }

    return 0;
}

int8_t read_light_sensor_reg(uint8_t read_reg_val)
{
    /* Write the read register to specify the initiate a read operation
*/
      if(wrapper_write(i2c_light_sensor_fd, &read_reg_val, 1) != 1){
            printf("Failed to write to the i2c bus.\n");
         return -1;
    }

    /* Read the value */
    int read_val;
      if (wrapper_read(i2c_light_sensor_fd, &read_val, 1) != 1) {
            perror("adc data read error");
         return -1;
      }
    //printf("**** read val for %d: %d\n", read_reg_val, read_val);

    int8_t ret_val = (int8_t)read_val;
```

```
    return ret_val;
}

void log_lux_data(float lux_data)
{
    int msg_priority;

    /* Set the message queue attributes */
    struct mq_attr logger_mq_attr = { .mq_flags = 0,
                                      .mq_maxmsg =
MSG_QUEUE_MAX_NUM_MSGS,  // Max number of messages on queue
                                      .mq_msgsize =
MSG_QUEUE_MAX_MSG_SIZE  // Max. message size
                                    };

    logger_mq_handle = mq_open(MSG_QUEUE_NAME, O_RDWR, S_IRWXU,
&logger_mq_attr);

    char lux_data_msg[128];
    memset(lux_data_msg, '\0', sizeof(lux_data_msg));

    sprintf(lux_data_msg, "Lux Value: %3.2f", lux_data);

    struct _logger_msg_struct_ logger_msg;
    memset(&logger_msg, '\0', sizeof(logger_msg));
    strcpy(logger_msg.message, lux_data_msg);
    strncpy(logger_msg.logger_msg_src_id, "Light", strlen("Light"));
    logger_msg.logger_msg_src_id[strlen("Light") + 1] = '\0';
    strncpy(logger_msg.logger_msg_level, "Info", strlen("Info"));
    logger_msg.logger_msg_level[strlen("Info") + 1] = '\0';

    msg_priority = 1;
    int num_sent_bytes = mq_send(logger_mq_handle, (char *)&logger_msg,
                            sizeof(logger_msg), msg_priority);
    if (num_sent_bytes < 0)
        perror("mq_send failed");
}

void sig_handler(int sig_num)
{
    char buffer[MSG_BUFF_MAX_LEN];
    memset(buffer, '\0', sizeof(buffer));

    if (sig_num == SIGINT || sig_num == SIGUSR1)
    {
        if (sig_num == SIGINT)
            printf("Caught signal %s in light task\n", "SIGINT");
        else if (sig_num == SIGUSR1)
            printf("Caught signal %s in light task\n", "SIGKILL");

        g_sig_kill_sensor_thread = 1;
        g_sig_kill_sock_thread = 1;
        g_sig_kill_sock_hb_thread = 1;

        //pthread_join(sensor_thread_id, NULL);
        //pthread_join(socket_thread_id, NULL);
        //pthread_join(socket_hb_thread_id, NULL);

        mq_close(logger_mq_handle);
```

```
        if (i2c_light_sensor_fd != -1)
            close(i2c_light_sensor_fd);

        exit(0);
    }
}

void write_cmd_reg(uint8_t cmd_reg_val)
{
    return;
}

uint8_t read_ctrl_reg(void)
{
    uint8_t cmd_ctrl_reg = I2C_LIGHT_SENSOR_CMD_CTRL_REG;

    int8_t light_sen_ctrl_reg_val = read_light_sensor_reg(cmd_ctrl_reg);
    if (light_sen_ctrl_reg_val != -1)
        return (uint8_t)light_sen_ctrl_reg_val;
    else
        return 0xFF; /* Sending 0xFF in case of error */

}

int write_ctrl_reg(uint8_t ctrl_reg_val)
{
    uint8_t cmd_ctrl_reg = I2C_LIGHT_SENSOR_CMD_CTRL_REG;

    if (write_light_sensor_reg(cmd_ctrl_reg, ctrl_reg_val) == 0)
    {
        return 0;
    }
    else
    {
        return -1;
    }
}

uint8_t read_timing_reg(void)
{
    uint8_t cmd_tim_reg = I2C_LIGHT_SENSOR_CMD_TIM_REG;

    int8_t light_sen_tim_reg_val = read_light_sensor_reg(cmd_tim_reg);
    if (light_sen_tim_reg_val != -1)
        return (uint8_t)light_sen_tim_reg_val;
    else
        return 0xFF; /* Sending 0xFF in case of error */

}

int write_timing_reg(uint8_t tim_reg_val, uint8_t field_to_set, uint8_t
field_val)
{
    uint8_t cmd_tim_reg = I2C_LIGHT_SENSOR_CMD_TIM_REG;
    int ret_val = -1;

    if (field_to_set & 0x3 == 0x3)
    {
        /* Setting integration time */
        uint8_t time_reg_val_copy = tim_reg_val;
```

```
        time_reg_val_copy &= 0xFC;
        time_reg_val_copy |= field_val;

        if (write_light_sensor_reg(cmd_tim_reg, time_reg_val_copy) == 0)
        {
            ret_val = 0;
        }
        else
        {
            ret_val = -1;
        }
        return ret_val;
    }
    if (field_to_set & 0x10 == 0x10)
    {
        /* Setting integration gain */
        uint8_t time_reg_val_copy = tim_reg_val;
        time_reg_val_copy &= ~0x10;
        time_reg_val_copy |= (field_val << 4);

        if (write_light_sensor_reg(cmd_tim_reg, time_reg_val_copy) == 0)
        {
            ret_val = 0;
        }
        else
        {
            ret_val = -1;
        }
        return ret_val;
    }
}

int enable_disable_intr_ctrl_reg(uint8_t int_ctrl_reg_val)
{
    uint8_t cmd_intr_ctrl_reg = I2C_LIGHT_SENSOR_CMD_INT_REG;

    if (write_light_sensor_reg(cmd_intr_ctrl_reg, int_ctrl_reg_val) == 0)
    {
        return 0;
    }
    else
    {
        return -1;
    }

}

uint8_t read_id_reg(void)
{
    uint8_t cmd_id_reg = I2C_LIGHT_SENSOR_CMD_ID_REG;

    int8_t light_sen_id_reg_val = read_light_sensor_reg(cmd_id_reg);
    if (light_sen_id_reg_val != -1)
        return (uint8_t)light_sen_id_reg_val;
    else
        return 0xFF; /* Sending 0xFF in case of error */

}

void read_intr_thresh_reg(uint16_t *low_thresh, uint16_t *high_thresh)
```

```
{
    uint8_t cmd_thresh_low_low_reg =
I2C_LIGHT_SENSOR_CMD_THRESH_LOW_LOW_REG;
    uint8_t cmd_thresh_low_high_reg =
I2C_LIGHT_SENSOR_CMD_THRESH_LOW_HIGH_REG;
    uint8_t cmd_thresh_high_low_reg =
I2C_LIGHT_SENSOR_CMD_THRESH_HIGH_LOW_REG;
    uint8_t cmd_thresh_high_high_reg =
I2C_LIGHT_SENSOR_CMD_THRESH_HIGH_HIGH_REG;

    int8_t light_sen_thresh_low_low_reg_val =
read_light_sensor_reg(cmd_thresh_low_low_reg);
    printf("thresh low low reg val : %d\n",
light_sen_thresh_low_low_reg_val);

    int8_t light_sen_thresh_low_high_reg_val =
read_light_sensor_reg(cmd_thresh_low_high_reg);
    printf("thresh low high reg val : %d\n",
light_sen_thresh_low_high_reg_val);

    int8_t light_sen_thresh_high_low_reg_val =
read_light_sensor_reg(cmd_thresh_high_low_reg);
    printf("thresh high low reg val : %d\n",
light_sen_thresh_high_low_reg_val);

    int8_t light_sen_thresh_high_high_reg_val =
read_light_sensor_reg(cmd_thresh_high_high_reg);
    printf("thresh high high reg val : %d\n",
light_sen_thresh_high_high_reg_val);

    *low_thresh = (light_sen_thresh_low_high_reg_val << 8 |
light_sen_thresh_low_low_reg_val);
    *high_thresh = (light_sen_thresh_high_high_reg_val << 8 |
light_sen_thresh_high_low_reg_val);
}

void write_intr_high_thresh_reg(uint16_t high_thresh)
{
      uint8_t cmd_thresh_high_low_reg =
I2C_LIGHT_SENSOR_CMD_THRESH_HIGH_LOW_REG;
    uint8_t cmd_thresh_high_low_reg_val = (uint8_t)high_thresh & 0xFF;
    write_light_sensor_reg(cmd_thresh_high_low_reg,
cmd_thresh_high_low_reg_val);

    uint8_t cmd_thresh_high_high_reg =
I2C_LIGHT_SENSOR_CMD_THRESH_HIGH_HIGH_REG;
    uint8_t cmd_thresh_high_high_reg_val = (uint8_t)((high_thresh >> 8) &
0xFF);
    write_light_sensor_reg(cmd_thresh_high_high_reg,
cmd_thresh_high_high_reg_val);
}

void write_intr_low_thresh_reg(uint16_t low_thresh)
{
      uint8_t cmd_thresh_low_low_reg =
I2C_LIGHT_SENSOR_CMD_THRESH_LOW_LOW_REG;
    uint8_t cmd_thresh_low_low_reg_val = (uint8_t)low_thresh & 0xFF;
    write_light_sensor_reg(cmd_thresh_low_low_reg,
cmd_thresh_low_low_reg_val);
```

```
    uint8_t cmd_thresh_low_high_reg =
I2C_LIGHT_SENSOR_CMD_THRESH_LOW_HIGH_REG;
    uint8_t cmd_thresh_low_high_reg_val = (uint8_t)((low_thresh >> 8) &
0xFF);
    write_light_sensor_reg(cmd_thresh_low_high_reg,
cmd_thresh_low_high_reg_val);
}

#if 0
void write_intr_thresh_reg(uint16_t low_thresh, uint16_t high_thresh)
{
    uint8_t cmd_thresh_low_low_reg =
I2C_LIGHT_SENSOR_CMD_THRESH_LOW_LOW_REG;
    uint8_t cmd_thresh_low_low_reg_val = (uint8_t)low_thresh & 0xFF;
    write_light_sensor_reg(cmd_thresh_low_low_reg,
cmd_thresh_low_low_reg_val);

    uint8_t cmd_thresh_low_high_reg =
I2C_LIGHT_SENSOR_CMD_THRESH_LOW_HIGH_REG;
    uint8_t cmd_thresh_low_high_reg_val = (uint8_t)((low_thresh >> 8) &
0xFF);
    write_light_sensor_reg(cmd_thresh_low_high_reg,
cmd_thresh_low_high_reg_val);

    uint8_t cmd_thresh_high_low_reg =
I2C_LIGHT_SENSOR_CMD_THRESH_HIGH_LOW_REG;
    uint8_t cmd_thresh_high_low_reg_val = (uint8_t)high_thresh & 0xFF;
    write_light_sensor_reg(cmd_thresh_high_low_reg,
cmd_thresh_high_low_reg_val);

    uint8_t cmd_thresh_high_high_reg =
I2C_LIGHT_SENSOR_CMD_THRESH_HIGH_HIGH_REG;
    uint8_t cmd_thresh_high_high_reg_val = (uint8_t)((high_thresh >> 8) &
0xFF);
    write_light_sensor_reg(cmd_thresh_high_high_reg,
cmd_thresh_high_high_reg_val);

}
#endif

void light_sensor_exit(void)
{
    /* Close i2c bus */
    if (i2c_light_sensor_fd != -1)
        close(i2c_light_sensor_fd);
}
/**************************************************************************
******
* Author:       Pavan Dhareshwar & Sridhar Pavithrapu
* Date:         03/07/2018
* File:         wrapper.c
* Description:  Source file describing the functionality and
implementation
*               of wrapper for synchronization of light and temperature
tasks.
**************************************************************************
*****/

/*---------------------------------- INCLUDES ---------------------------
----*/
```

```
#include "wrapper.h"

sem_t *get_named_semaphore_handle(void)
{
    sem_t *sem;
    if ((sem = sem_open("wrapper_sem", O_CREAT, 0644, 1)) == SEM_FAILED)
    {
        perror("sem_open failed");
        return SEM_FAILED;
    }
    else
    {
        //printf("Named semaphore created successfully\n");
        return sem;
    }
}

ssize_t wrapper_write(int fd, void *buf, size_t count){

     ssize_t return_value = 0;

#if 1
    sem_t *wrapper_sem = get_named_semaphore_handle();
    if (wrapper_sem == SEM_FAILED)
    {
        return -1000;
    }

     if(sem_wait(wrapper_sem) == 0)
    {
            return_value = write(fd, buf, count);
    }
     else{
           perror("sem_wait error in wrapper\n");
     }

     if(sem_post(wrapper_sem) != 0){

           perror("sem_post error in wrapper\n");
     }
#else
    return_value = write(fd, buf, count);
#endif
     return return_value;
}

ssize_t wrapper_read(int fd, void *buf, size_t count){

     ssize_t return_value = 0;

#if 1
    sem_t *wrapper_sem = get_named_semaphore_handle();
    if (wrapper_sem == SEM_FAILED)
    {
        return -1000;
    }

    if(sem_wait(wrapper_sem) == 0){

            return_value = read(fd, buf, count);
```

```
    }
      else{
            perror("sem_wait error in wrapper\n");
      }

      if(sem_post(wrapper_sem) != 0){

            perror("sem_post error in wrapper\n");
      }
#else
    return_value = read(fd, buf, count);
#endif

      return return_value;
}
#ifndef _WRAPPER_H_
#define _WRAPPER_H_

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

sem_t *get_named_semaphore_handle(void);

ssize_t wrapper_write(int fd, void *buf, size_t count);
ssize_t wrapper_read(int fd, void *buf, size_t count);

#endif
/*************************************************************************
*
* Author:       Pavan Dhareshwar & Sridhar Pavithrapu
* Date:         03/07/2018
* File:         light_sensor.h
* Description:  Header file containing the macros, structs/enums, globals
                and function prototypes for source file light_sensor.c
*************************************************************************
/

#ifndef _LIGHT_SENSOR_TASK_H_
#define _LIGHT_SENSOR_TASK_H_

/*---------------------------------- INCLUDES ---------------------------
----*/
#include <errno.h>
#include <stdint.h>
#include <string.h>
#include <math.h>

#include <unistd.h>
#include <fcntl.h>
#include <signal.h>
```

```
#include <linux/i2c-dev.h>

#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/msg.h>
#include <sys/ipc.h>

#include <mqueue.h>

#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/socket.h>

#include "wrapper.h"

/*---------------------------------- INCLUDES ----------------------------
----*/

/*----------------------------------- MACROS -----------------------------
----*/

#define I2C_SLAVE_ADDR                0b0111001       // Slave address -
0x39
#define I2C_DEV_NAME                  "/dev/i2c-2"

#define I2C_LIGHT_SENSOR_CMD_CTRL_REG                0x80
#define I2C_LIGHT_SENSOR_CMD_TIM_REG                 0x81

#define I2C_LIGHT_SENSOR_CMD_THRESH_LOW_LOW_REG      0x82
#define I2C_LIGHT_SENSOR_CMD_THRESH_LOW_HIGH_REG     0x83
#define I2C_LIGHT_SENSOR_CMD_THRESH_HIGH_LOW_REG     0x84
#define I2C_LIGHT_SENSOR_CMD_THRESH_HIGH_HIGH_REG    0x85

#define I2C_LIGHT_SENSOR_CMD_INT_REG                 0x86
#define I2C_LIGHT_SENSOR_CMD_ID_REG                  0x8A

#define I2C_LIGHT_SENSOR_CMD_DATA0LOW_REG            0x8C
#define I2C_LIGHT_SENSOR_CMD_DATA0HIGH_REG           0x8D
#define I2C_LIGHT_SENSOR_CMD_DATA1LOW_REG            0x8E
#define I2C_LIGHT_SENSOR_CMD_DATA1HIGH_REG           0x8F

#define I2C_LIGHT_SENSOR_CTRL_REG_VAL                0x3

#define MSG_QUEUE_NAME                               "/logger_task_mq"
#define MSG_QUEUE_MAX_NUM_MSGS                       5
#define MSG_QUEUE_MAX_MSG_SIZE                       1024

#define MSG_MAX_LEN                                  128

#define LIGHT_SENSOR_SERVER_PORT_NUM                 8086
#define LIGHT_SENSOR_LISTEN_QUEUE_SIZE               5

#define MSG_BUFF_MAX_LEN                             1024

#define SOCK_REQ_MSG_API_MSG_LEN                     64

#define SOCKET_HB_PORT_NUM                           8660
#define SOCKET_HB_LISTEN_QUEUE_SIZE                  5
```

```
#define MSG_TYPE_TEMP_DATA                          0
#define MSG_TYPE_LUX_DATA                           1
#define MSG_TYPE_SOCK_DATA                          2
#define MSG_TYPE_MAIN_DATA                          3

#define LOGGER_ATTR_LEN                             32

/*---------------------------------- MACROS ----------------------------
----*/

/*---------------------------------- GLOBALS ----------------------------
----*/
int i2c_light_sensor_fd;
int server_fd, accept_conn_id;
int sensor_thread_id, socket_thread_id, socket_hb_thread_id;

mqd_t logger_mq_handle;

sig_atomic_t g_sig_kill_sensor_thread, g_sig_kill_sock_thread,
g_sig_kill_sock_hb_thread;
int light_sensor_initialized;
/*---------------------------------- GLOBALS ----------------------------
----*/

/*---------------------------- STRUCTURES/ENUMERATIONS ------------------
----*/

struct _logger_msg_struct_
{
    char message[MSG_MAX_LEN];
    char logger_msg_src_id[LOGGER_ATTR_LEN];
    char logger_msg_level[LOGGER_ATTR_LEN];
};

enum _req_recipient_
{
    REQ_RECP_TEMP_TASK,
    REQ_RECP_LIGHT_TASK
};

struct _socket_req_msg_struct_
{
    char req_api_msg[SOCK_REQ_MSG_API_MSG_LEN];
    enum _req_recipient_ req_recipient;
    int param;
};

#if 0
struct _int_thresh_reg_struct_
{
    uint8_t thresh_low_low;
    uint8_t thresh_low_high;
    uint8_t thresh_high_low;
    uint8_t thresh_high_high;
};
#endif
struct _int_thresh_reg_struct_
{
    uint16_t low_thresh;
```

```
    uint16_t high_thresh;
};

struct _light_sensor_tim_params
{
    uint8_t tim_reg_val;
    uint8_t tim_reg_field_to_set;
    uint8_t tim_reg_field_val;

};
/*---------------------------- STRUCTURES/ENUMERATIONS ------------------
----*/

/*---------------------------- FUNCTION PROTOTYPES ----------------------
----*/
/**
 *  @brief Initialize the light sensor
 *
 *  This function will open the i2c bus for read and write operation and
 *  initialize the communication with the peripheral.
 *
 *  @param void
 *
 *  @return 0  : if sensor initialization is a success
            -1 : if sensor initialization fails
*/
int light_sensor_init();

/**
 *  @brief Power on the light sensor
 *
 *  This function will configure the control register to power on the
light
 *  sensor.
 *
 *  @param void
 *
 *  @return void
 *
 */
void power_on_light_sensor(void);

/**
 *  @brief Create sensor,socket and hearbeat socket threads for light
task
 *
 *  The light task is made multi-threaded with
 *     1. sensor thread responsible for communicating via I2C interface
 *        with the light sensor to get light data and a socket
 *        thread.
 *     2. socket thread responsible for communicating with socket thread
and
 *        serve request from external application forwarded via socket
task.
  *      3. socket heartbeat responsible for communicating with main
task,
 *           to log heartbeat every time its requested by main task.
 *
 *  @param void
 *
```

```
 *  @return 0  : thread creation success
 *          -1 : thread creation failed
 *
 */
int create_threads(void);

/**
 *  @brief Initialize light task socket
 *
 *  This function will create, bind and make the socket listen for
incoming
 *  connections.
 *
 *  @param sock_addr_struct : pointer to sockaddr_in structure
 *
 *  @return void
 *
 */
void init_light_socket(struct sockaddr_in *sock_addr_struct);

/**
 *  @brief Entry point and executing entity for sensor thread
 *
 *  The sensor thread starts execution by invoking this
function(start_routine)
 *
 *  @param arg : argument to start_routine
 *
 *  @return void
 *
 */
void *sensor_thread_func(void *arg);

/**
 *  @brief Entry point and executing entity for socket thread
 *
 *  The socket thread starts execution by invoking this
function(start_routine)
 *
 *  @param arg : argument to start_routine
 *
 *  @return void
 *
 */
void *socket_thread_func(void *arg);

/**
 *  @brief Entry point and executing entity for socket thread
 *
 *  The socket thread for heartbeat starts execution by invoking this
function(start_routine)
 *
 *  @param arg : argument to start_routine
 *
 *  @return void
 *
 */
void *socket_hb_thread_func(void *arg);

/**
```

```
 *  @brief Get lux data from light sensor
 *
 *  This function will get the illuminance (ambient light level) in lux
and
 *  return this value.
 *
 *  @param void
 *
 *  @return float lux data
*/
float get_lux_data();

/**
 *  @brief Write light sensor register
 *
 *  This function will write to light sensor data specifed by @param(
 *  cmd_reg_val) with a value specified by @param(target_reg_val)
 *
 *  @param cmd_reg_val     : command register value
 *  @param target_reg_val  : value to be written to target register
 *
 *  @return 0   : if register write is successful
 *          -1  : if register write fails
*/
int write_light_sensor_reg(int cmd_reg_val, int target_reg_val);

/**
 *  @brief Read light sensor register
 *
 *  This function will read light sensor data specifed by @param(
 *  read_reg_val)
 *
 *  @param read_reg_val     : register to be read
 *
 *  @return reg_val   : if register read is successful
 *          -1        : if register read fails
*/
int8_t read_light_sensor_reg(uint8_t read_reg_val);

/**
 *  @brief Get the ADC channel data
 *
 *  This function will read the ADC data for channel specified by @param(
 *  channel_num) and populate them @param(ch_data_low) and
@param(ch_data_high)
 *
 *  @param channel_num     : ADC channel number to be read
 *  @param ch_data         : pointer to ADC data
 *
 *  @return void
*/
void get_adc_channel_data(int channel_num, uint16_t *ch_data);

/**
 *  @brief Calculate the lux value
 *
 *  This function calculates the illuminance value
 *
 *  @param ch0_data     : ADC channel 0 data
 *  @param ch1_data     : ADC channel 1 data
```

```
 *
 *  @return lux_val
*/
float calculate_lux_value(uint16_t ch0_data, uint16_t ch1_data);

/**
 *  @brief Log the lux value
 *
 *  This function writes the lux value calculated to logger message queue
 *
 *  @param lux_data     : lux_data
 *
 *  @return void
*/
void log_lux_data(float lux_data);

/**
 *  @brief Cleanup of the light sensor
 *
 *  This function will close the i2c bus for read and write operation and
 *  perform any cleanup required
 *
 *  @param void
 *
 *  @return void
*/
void light_sensor_exit(void);

/**
 *  @brief Create the socket and initialize
 *
 *  This function create the socket for the given socket id.
 *
 *  @param sock_fd                      : socket file descriptor
 *             server_addr_struct  : server address of the socket
 *             port_num                 : port number in which the socket
is communicating
 *             listen_qsize        : number of connections the socket is
accepting
 *
 *  @return void
*/
void init_sock(int *sock_fd, struct sockaddr_in *server_addr_struct,
               int port_num, int listen_qsize);

/**
 *  @brief Signal handler for temperature task
 *
 *  This function handles the reception of SIGKILL and SIGINT signal to
the
 *  temperature task and terminates all the threads, closes the I2C file
descriptor
 *  and logger message queue handle and exits.
 *
 *  @param sig_num              : signal number
 *
 *  @return void
*/

void sig_handler(int sig_num);
```

```
/**
 *  @brief Write command register of light sensor
 *
 *  This function will write to command register of light sensor
 *
 *  @param cmd_reg_val    : value to be written
 *
 *  @return 0   : success
 *          -1  : failure
 */
void write_cmd_reg(uint8_t cmd_reg_val);

/**
 *  @brief Read control register of light sensor
 *
 *  This function will read the control register of light sensor
 *
 *  @param void
 *
 *  @return ctrl_reg_val
*/
uint8_t read_ctrl_reg(void);

/**
 *  @brief Write control register of light sensor
 *
 *  This function will write to control register of light sensor
 *
 *  @param ctrl_reg_val    : value to be written
 *
 *  @return 0   : success
 *          -1  : failure
 */
int write_ctrl_reg(uint8_t ctrl_reg_val);

/**
 *  @brief Read timing register of light sensor
 *
 *  This function will read the timing register of light sensor
 *
 *  @param void
 *
 *  @return tim_reg_val
*/
uint8_t read_timing_reg(void);

/**
 *  @brief Write timing register of light sensor
 *
 *  This function will write to timing register of light sensor
 *
 *  @param tim_reg_val     : value to be written
 *  @param field_to_set    : timing register field to be set
 *  @param field_val       : field value
 *
 *  @return 0   : success
 *          -1  : failure
 */
```

```
int write_timing_reg(uint8_t tim_reg_val, uint8_t field_to_set, uint8_t
field_val);

/**
 *  @brief Enable or disable interrupt register of light sensor
 *
 *  This function will enable or diable the interrupt control register of
 *  light sensor
 *
 *  @param int_ctrl_reg_val     : value to be written
 *
 *  @return 0   : success
 *          -1  : failure
*/
int enable_disable_intr_ctrl_reg(uint8_t int_ctrl_reg_val);

/**
 *  @brief Read identification register of light sensor
 *
 *  This function will read the identification register of light sensor
 *
 *  @param void
 *
 *  @return tim_reg_val
*/
uint8_t read_id_reg(void);

/**
 *  @brief Read interrupt threshold register of light sensor
 *
 *  This function will read the interrupt threshold register of light
sensor
 *
 *  @param low_thresh            : pointer to low threshold value
 *  @param high_thresh           : pointer to high threshold value
 *
 *  @return void
*/
void read_intr_thresh_reg(uint16_t *low_thresh, uint16_t *high_thresh);

/**
 *  @brief Write interrupt threshold register of light sensor
 *
 *  This function will write the interrupt threshold register of light
sensor
 *
 *  @param low_thresh            : low threshold value to be written
 *  @param high_thresh           : high threshold value to be written
 *
 *  @return void
*/
void write_intr_thresh_reg(uint16_t low_thresh, uint16_t high_thresh);

void write_intr_high_thresh_reg(uint16_t high_thresh);
void write_intr_low_thresh_reg(uint16_t low_thresh);
/*---------------------------- FUNCTION PROTOTYPES ----------------------
----*/

#endif
/**
```

```
 * @file                   Temperature_alert.c
 * @author                 Sridhar Pavithrapu
 * @date                   15 March 2018
 * @original author        Derek Molloy
 * @original date          19 April 2015
 * @brief  A kernel module for controlling a GPIO LED/button pair. The
device mounts devices via
 * sysfs /sys/class/gpio/gpio115 and gpio49. Therefore, this test LKM
circuit assumes that an LED
 * is attached to GPIO 49 which is on P9_23 and the button is attached to
GPIO 115 on P9_27. There
 * is no requirement for a custom overlay, as the pins are in their
default mux mode states.
 * @see http://www.derekmolloy.ie/

 * Credit/Note: This code was originally developed by Derek Molloy. We
have used it as reference and
 *              modified it to meet our requirements. Most of the code
remains the same as original,
 *              and to demonstrate code reference, we haven't modified it
to look any different.
*/

#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/gpio.h>                  // Required for the GPIO
functions
#include <linux/interrupt.h>             // Required for the IRQ code

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Derek Molloy");
MODULE_DESCRIPTION("A Button/LED test driver for the BBB");
MODULE_VERSION("0.1");

static unsigned int gpioLED = 49;        ///< hard coding the LED gpio for
this example to P9_23 (GPIO49)
static unsigned int gpioButton = 115;    ///< hard coding the button gpio
for this example to P9_27 (GPIO115)
static unsigned int irqNumber;           ///< Used to share the IRQ number
within this file
static unsigned int numberPresses = 0;  ///< For information, store the
number of button presses
static bool    ledOn = 0;          ///< Is the LED on or off? Used to
invert its state (off by default)

/// Function prototype for the custom IRQ handler function -- see below
for the implementation
static irq_handler_t  ebbgpio_irq_handler(unsigned int irq, void *dev_id,
struct pt_regs *regs);

/** @brief The LKM initialization function
 *  The static keyword restricts the visibility of the function to within
this C file. The __init
 *  macro means that for a built-in driver (not a LKM) the function is
only used at initialization
 *  time and that it can be discarded and its memory freed up after that
point. In this example this
 *  function sets up the GPIOs and the IRQ
 *  @return returns 0 if successful
```

```
 */
static int __init ebbgpio_init(void){
   int result = 0;
   printk(KERN_INFO "GPIO_TEST: Initializing the GPIO_TEST LKM\n");
   // Is the GPIO a valid GPIO number (e.g., the BBB has 4x32 but not all
available)
   if (!gpio_is_valid(gpioLED)){
      printk(KERN_INFO "GPIO_TEST: invalid LED GPIO\n");
      return -ENODEV;
   }
   // Going to set up the LED. It is a GPIO in output mode and will be on
by default
   ledOn = true;
   gpio_request(gpioLED, "sysfs");          // gpioLED is hardcoded to
49, request it
   gpio_direction_output(gpioLED, ledOn);   // Set the gpio to be in
output mode and on
// gpio_set_value(gpioLED, ledOn);          // Not required as set by
line above (here for reference)
   gpio_export(gpioLED, false);             // Causes gpio49 to appear in
/sys/class/gpio
                                       // the bool argument prevents the
direction from being changed
   gpio_request(gpioButton, "sysfs");       // Set up the gpioButton
   gpio_direction_input(gpioButton);        // Set the button GPIO to be
an input
   gpio_export(gpioButton, false);          // Causes gpio115 to appear
in /sys/class/gpio
                                       // the bool argument prevents the
direction from being changed
   // Perform a quick test to see that the button is working as expected
on LKM load
   printk(KERN_INFO "GPIO_TEST: The button state is currently: %d\n",
gpio_get_value(gpioButton));

   // GPIO numbers and IRQ numbers are not the same! This function
performs the mapping for us
   irqNumber = gpio_to_irq(gpioButton);
   printk(KERN_INFO "GPIO_TEST: The button is mapped to IRQ: %d\n",
irqNumber);

   // This next call requests an interrupt line
   result = request_irq(irqNumber,             // The interrupt number
requested
                        (irq_handler_t) ebbgpio_irq_handler, // The
pointer to the handler function below
                        IRQF_TRIGGER_HIGH,   // Interrupt on rising edge
(button press, not release)
                        "ebb_gpio_handler",    // Used in
/proc/interrupts to identify the owner
                        NULL);                 // The *dev_id for shared
interrupt lines, NULL is okay

   printk(KERN_INFO "GPIO_TEST: The interrupt request result is: %d\n",
result);
   return result;
}

/** @brief The LKM cleanup function
```

```
 *  Similar to the initialization function, it is static. The __exit
macro notifies that if this
 *  code is used for a built-in driver (not a LKM) that this function is
not required. Used to release the
 *  GPIOs and display cleanup messages.
 */
static void __exit ebbgpio_exit(void){
   printk(KERN_INFO "GPIO_TEST: The button state is currently: %d\n",
gpio_get_value(gpioButton));
   printk(KERN_INFO "GPIO_TEST: The button was pressed %d times\n",
numberPresses);
   gpio_set_value(gpioLED, 0);              // Turn the LED off, makes it
clear the device was unloaded
   gpio_unexport(gpioLED);                  // Unexport the LED GPIO
   free_irq(irqNumber, NULL);               // Free the IRQ number, no
*dev_id required in this case
   gpio_unexport(gpioButton);               // Unexport the Button GPIO
   gpio_free(gpioLED);                      // Free the LED GPIO
   gpio_free(gpioButton);                   // Free the Button GPIO
   printk(KERN_INFO "GPIO_TEST: Goodbye from the LKM!\n");
}

/** @brief The GPIO IRQ Handler function
 *  This function is a custom interrupt handler that is attached to the
GPIO above. The same interrupt
 *  handler cannot be invoked concurrently as the interrupt line is
masked out until the function is complete.
 *  This function is static as it should not be invoked directly from
outside of this file.
 *  @param irq    the IRQ number that is associated with the GPIO --
useful for logging.
 *  @param dev_id the *dev_id that is provided -- can be used to identify
which device caused the interrupt
 *  Not used in this example as NULL is passed.
 *  @param regs   h/w specific register values -- only really ever used
for debugging.
 *  return returns IRQ_HANDLED if successful -- should return IRQ_NONE
otherwise.
 */
static irq_handler_t ebbgpio_irq_handler(unsigned int irq, void *dev_id,
struct pt_regs *regs){
   ledOn = !ledOn;                          // Invert the LED state on
each button press
   gpio_set_value(gpioLED, ledOn);          // Set the physical LED
accordingly
   printk(KERN_INFO "GPIO_TEST: Interrupt! (button state is %d)\n",
gpio_get_value(gpioButton));
   numberPresses++;                         // Global counter, will be
outputted when the module is unloaded
   return (irq_handler_t) IRQ_HANDLED;      // Announce that the IRQ has
been handled correctly
}

/// This next calls are  mandatory -- they identify the initialization
function
/// and the cleanup function (as above).
module_init(ebbgpio_init);
module_exit(ebbgpio_exit);
/****************************************************************
***
```

```
* Author:       Pavan Dhareshwar & Sridhar Pavithrapu
* Date:         03/11/2018
* File:         main_task.c
* Description:  Source file containing the functionality and
implementation
*               of the main task
**************************************************************************
**/

#include "main_task.h"

int main(void)
{
      char buffer[BUFF_SIZE];

    create_sub_processes();

    /* Open semaphore used for synchronization */
    sem_t *shared_sem;

    if ((shared_sem = sem_open("wrapper_sem", O_CREAT | O_EXCL, 0644, 1))
== SEM_FAILED)
    {
        perror("sem_open failed");
    }
    else
    {
        printf("Named semaphore created successfully\n");
        sem_unlink("wrapper_sem");
    }

    if (signal(SIGINT, sig_handler) == SIG_ERR)
        printf("SigHandler setup for SIGINT failed\n");

    if (signal(SIGUSR1, sig_handler) == SIG_ERR)
        printf("SigHandler setup for SIGKILL failed\n");

    sleep(3);

    /* Create and initialize temperature task socket */
    initialize_sub_task_socket(&temp_task_sockfd, &temp_task_sock_addr,
TEMP_TASK_PORT_NUM);

    if (connect(temp_task_sockfd, (struct sockaddr
*)&temp_task_sock_addr, sizeof(temp_task_sock_addr)) < 0)
    {
        printf("\nConnection Failed for temp task \n");
        return -1;
    }

    /* Create and initialize light task socket */
    initialize_sub_task_socket(&light_task_sockfd, &light_task_sock_addr,
LIGHT_TASK_PORT_NUM);

    if (connect(light_task_sockfd, (struct sockaddr
*)&light_task_sock_addr, sizeof(light_task_sock_addr)) < 0)
    {
        printf("\nConnection Failed for light task \n");
        return -1;
    }
```

```
    /* Create and initialize socket task socket */
    initialize_sub_task_socket(&socket_task_sockfd,
&socket_task_sock_addr, SOCKET_TASK_PORT_NUM);

    if (connect(socket_task_sockfd, (struct sockaddr
*)&socket_task_sock_addr, sizeof(socket_task_sock_addr)) < 0)
    {
        printf("\nConnection Failed for socket task \n");
        return -1;
    }

    /* Create and initialize logger task socket */
    initialize_sub_task_socket(&logger_task_sockfd,
&logger_task_sock_addr, LOGGER_TASK_PORT_NUM);

    if (connect(logger_task_sockfd, (struct sockaddr
*)&logger_task_sock_addr, sizeof(logger_task_sock_addr)) < 0)
    {
        printf("\nConnection Failed for logger task \n");
        return -1;
    }

    sleep(2);

    printf("Performing system start-up test\n");
    perform_startup_test();

    while(!g_kill_main_task)
    {
        check_status_of_sub_tasks();

        sleep(10);
    }

     return 0;
}

void create_sub_processes(void)
{
    char sub_process_name[32];

    FILE *fp_pid_file = fopen("pid_info_file.txt", "r");
    if (fp_pid_file)
    {
        fclose(fp_pid_file);
        remove("pid_info_file.txt");
    }

    /* Creating logger task */
    memset(sub_process_name, '\0', sizeof(sub_process_name));
    strcpy(sub_process_name, "logger");
    create_sub_process(sub_process_name);

    /* Creating temperature sensor task */
    memset(sub_process_name, '\0', sizeof(sub_process_name));
    strcpy(sub_process_name, "temperature");
    create_sub_process(sub_process_name);

    /* Creating light sensor task */
```

```
    memset(sub_process_name, '\0', sizeof(sub_process_name));
    strcpy(sub_process_name, "light");
    create_sub_process(sub_process_name);

    sleep(2);

    /* Creating socket task */
    memset(sub_process_name, '\0', sizeof(sub_process_name));
    strcpy(sub_process_name, "socket");
    create_sub_process(sub_process_name);

}

void create_sub_process(char *process_name)
{
    pid_t child_pid;

    child_pid = fork();

    if (child_pid == 0)
    {
        /* Child Process */
        if (!strcmp(process_name, "temperature"))
        {
            write_pid_to_file(process_name, getpid());
            printf("Creating temperature task\n");
            //char *args[]={LOGGER_TASK_EXEC_NAME, NULL};
            char *args[]={"./temp_task", "&", NULL};
            execvp(args[0],args);
        }
        else if (!strcmp(process_name, "light"))
        {
            write_pid_to_file(process_name, getpid());
            printf("Creating light task\n");
            char *args[]={"./light_task", "&", NULL};
            execvp(args[0],args);
        }
        else if (!strcmp(process_name, "socket"))
        {
            write_pid_to_file(process_name, getpid());
            printf("Creating socket task\n");
            char *args[]={"./socket_task", "&", NULL};
            execvp(args[0],args);
        }
        else if (!strcmp(process_name, "logger"))
        {
            write_pid_to_file(process_name, getpid());
            printf("Creating logger task\n");
            char *args[]={"./logger_task", "&", NULL};
            execvp(args[0],args);
        }
    }
    else if (child_pid > 0)
    {
        /* Parent Process */
        /* We are just returning back to main function in the parent
process */
    }
    else
    {
```

```
        printf("fork failed while creating child process for %s task\n",
process_name);
        perror("fork failed");
    }

    return;

}

void write_pid_to_file(char *proc_name, pid_t child_pid)
{
    FILE *fp_pid_file = fopen("pid_info_file.txt", "r");
    char pid_info_str[64];
    memset(pid_info_str, '\0', sizeof(pid_info_str));

    if (fp_pid_file == NULL)
    {
        fp_pid_file = fopen("pid_info_file.txt", "w");

        sprintf(pid_info_str, "%s task: %d\n", proc_name,
(int)child_pid);
        fwrite(pid_info_str, strlen(pid_info_str), sizeof(char),
fp_pid_file);

        fclose(fp_pid_file);
    }
    else
    {
        fclose(fp_pid_file);
        fp_pid_file = fopen("pid_info_file.txt", "a");

        sprintf(pid_info_str, "%s task: %d\n", proc_name,
(int)child_pid);
        fwrite(pid_info_str, strlen(pid_info_str), sizeof(char),
fp_pid_file);

        fclose(fp_pid_file);
    }
}

void initialize_sub_task_socket(int *sock_fd, struct sockaddr_in
*sock_addr_struct, int port_num)
{
      memset(sock_addr_struct, '0', sizeof(struct sockaddr_in));
      sock_addr_struct->sin_family = AF_INET;
      sock_addr_struct->sin_port = htons(port_num);

    if ((*sock_fd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    {
        perror("socket creation failed");
        exit(EXIT_FAILURE);
    }

    int option = 1;
    if(setsockopt(*sock_fd, SOL_SOCKET, (SO_REUSEPORT | SO_REUSEADDR),
                (char*)&option,sizeof(option)) < 0)
    {
        perror("setsockopt for socket reusability failed");
        close(*sock_fd);
        exit(EXIT_FAILURE);
```

```
    }

    struct timeval rcv_timeout;
    rcv_timeout.tv_sec = 5;
    rcv_timeout.tv_usec = 0;
    if (setsockopt(*sock_fd, SOL_SOCKET, SO_SNDTIMEO, (struct timeval
*)&rcv_timeout,sizeof(struct timeval)) < 0)
    {
        perror("setsockopt for send timeout set failed");
        close(*sock_fd);
        exit(EXIT_FAILURE);
    }

    if (setsockopt(*sock_fd, SOL_SOCKET, SO_RCVTIMEO, (struct timeval
*)&rcv_timeout,sizeof(struct timeval)) < 0)
    {
        perror("setsockopt for recv timeout set failed");
        close(*sock_fd);
        exit(EXIT_FAILURE);
    }

    // Convert IPv4 and IPv6 addresses from text to binary form
    if(inet_pton(AF_INET, SENSOR_TASK_SOCK_IP_ADDR, &(sock_addr_struct-
>sin_addr))<=0)
    {
        perror("inet_pton failed");
        printf("\nInvalid address/ Address not supported for temperature
task\n");
        exit(EXIT_FAILURE);
    }
}

void check_status_of_sub_tasks(void)
{
    /* Check if temperature task is alive */
    check_subtask_status(temp_task_sockfd, "Temperature");

    /* Check if light task is alive */
    check_subtask_status(light_task_sockfd, "Light");

    /* Check if socket task is alive */
    check_subtask_status(socket_task_sockfd, "Socket");

    /* Check if logger task is alive */
    check_subtask_status(logger_task_sockfd, "Logger");
}

void check_subtask_status(int sock_fd, char *task_name)
{
    char recv_buffer[BUFF_SIZE];
    char send_buffer[] = "heartbeat";

    memset(recv_buffer, '\0', sizeof(recv_buffer));

    ssize_t num_sent_bytes = send(sock_fd, send_buffer,
sizeof(send_buffer), 0);
    if (num_sent_bytes < 0)
        perror("send failed");
```

```
    ssize_t num_recv_bytes = recv(sock_fd, recv_buffer,
sizeof(recv_buffer), 0);
    if (num_recv_bytes < 0)
        perror("recv failed");

    if (!strcmp(recv_buffer, "Alive"))
        printf("%s task alive\n", task_name);
    else
    {
        if (!strcmp(task_name, "Temperature"))
        {
            temp_task_unalive_count++;

            if (temp_task_unalive_count >=
TEMP_TASK_UNALIVE_CNT_LOG_LIMIT)
                log_task_unalive_msg_to_log_file(task_name);
        }
        else if (!strcmp(task_name, "Light"))
        {
            light_task_unalive_count++;

            if (light_task_unalive_count >=
LIGHT_TASK_UNALIVE_CNT_LOG_LIMIT)
                log_task_unalive_msg_to_log_file(task_name);
        }
        else if (!strcmp(task_name, "Logger"))
        {
            logger_task_unalive_count++;

            if (logger_task_unalive_count >=
LOGGER_TASK_UNALIVE_CNT_LOG_LIMIT)
                log_task_unalive_msg_to_log_file(task_name);
        }
        else if (!strcmp(task_name, "Socket"))
        {
            socket_task_unalive_count++;

            if (socket_task_unalive_count >=
SOCK_TASK_UNALIVE_CNT_LOG_LIMIT)
                log_task_unalive_msg_to_log_file(task_name);
        }
    }
}

void perform_startup_test(void)
{
    /* The startup test will validate whether the hardware and software
is in
    ** working order.
    **
    ** Specifically, it checks the following things:
    **
    ** 1. Communication with the temperature sensor to confirm that I2C
interface
    **    works and the hardware is working.
    ** 2. Communication with the light sensor to confirm that I2C
interface
    **    works and the hardware is working.
    ** 3. Communication to the sub processes to make sure they have all
started
```

```
    **    and are up and running
    */

    /* Check the temperature sensor task, hardware and I2C */
    int temp_task_st_status =
perform_sub_task_startup_test(temp_task_sockfd, "temp");
    if (temp_task_st_status != 0)
        stop_entire_system();

    /* Check the light sensor task, hardware and I2C */
    int light_task_st_status =
perform_sub_task_startup_test(light_task_sockfd, "light");
    if (light_task_st_status != 0)
        stop_entire_system();

    /* Check the logger task */
    int logger_task_st_status =
perform_sub_task_startup_test(logger_task_sockfd, "logger");
    if (logger_task_st_status != 0)
        stop_entire_system();

    /* Check the socket task */
    int socket_task_st_status =
perform_sub_task_startup_test(socket_task_sockfd, "socket");
    if (socket_task_st_status != 0)
        stop_entire_system();
}

int perform_sub_task_startup_test(int sock_fd, char *proc_name)
{
    char recv_buffer[BUFF_SIZE];
    char send_buffer[] = "startup_check";

    memset(recv_buffer, '\0', sizeof(recv_buffer));

    ssize_t num_sent_bytes = send(sock_fd, send_buffer,
sizeof(send_buffer), 0);
    if (num_sent_bytes < 0)
        perror("send failed");

    ssize_t num_recv_bytes = recv(sock_fd, recv_buffer,
sizeof(recv_buffer), 0);
    if (num_recv_bytes < 0)
        perror("recv failed");

    if (!strcmp(recv_buffer, "Initialized"))
    {
        printf("%s sensor is initialized\n", proc_name);
        return 0;
    }
    else if (!strcmp(recv_buffer, "Uninitialized"))
    {
        printf("%s sensor isn't initalized\n", proc_name);
        return -1;
    }
    else
    {
        printf("Message received on socket : %s unknown\n", recv_buffer);
        return -1;
    }
```

```
}

void stop_entire_system(void)
{
    kill_already_created_processes();

    /* Turn on USR led to indicate that a failure has occurred */
    turn_on_usr_led();

    exit(1);
}

void kill_already_created_processes(void)
{
    printf("Killing already created processes\n");

    FILE *fp_pid_info_file = fopen("./pid_info_file.txt", "r");
    if (fp_pid_info_file == NULL)
    {
        perror("file open failed");
        printf("File %s open failed\n", "pid_info_file.txt");
        return;
    }

    char *buffer;
    size_t num_bytes = 120;
    char colon_delimiter[] = ":";
    ssize_t bytes_read;

    buffer = (char *)malloc(num_bytes*sizeof(char));

    while ((bytes_read = getline(&buffer, &num_bytes, fp_pid_info_file))
!= -1)
    {
        char *token = strtok(buffer, colon_delimiter);

        if (!strcmp(token, "temperature task"))
        {
            token = strtok(NULL, colon_delimiter);
            printf("Killing temperature task\n");
            int pid_to_kill = atoi(token);

            /* We wanted to kill the temperature process here by sending
a SIGKILL,
            ** but since we could not setup a signal handler for SIGKILL,
we are
            ** sending a SIGSTOP instead and trying to handle SIGUSR1 in
the
            ** temperature process */
            kill(pid_to_kill, SIGUSR1);

            int status;
            pid_t end_id = waitpid(pid_to_kill, &status, 0);
            if (end_id == pid_to_kill)
            {
                if (WIFEXITED(status))
                    printf("Temperature task successfully killed\n");
            }
            else
            {
```

```
            perror("Temperature task: waitpid error\n");
        }
    }
    else if (!strcmp(token, "light task"))
    {
        token = strtok(NULL, colon_delimiter);
        printf("Killing light task\n");
        int pid_to_kill = atoi(token);

        kill(pid_to_kill, SIGUSR1);

        int status;
        pid_t end_id = waitpid(pid_to_kill, &status, 0);
        if (end_id == pid_to_kill)
        {
            if (WIFEXITED(status))
                printf("Light task successfully killed\n");
        }
        else
        {
            perror("Light task: waitpid error\n");
        }

    }
    else if (!strcmp(token, "logger task"))
    {
        token = strtok(NULL, colon_delimiter);
        printf("Killing logger task\n");
        int pid_to_kill = atoi(token);

        kill(pid_to_kill, SIGUSR1);

        int status;
        pid_t end_id = waitpid(pid_to_kill, &status, 0);
        if (end_id == pid_to_kill)
        {
            if (WIFEXITED(status))
                printf("Logger task successfully killed\n");
        }
        else
        {
            perror("Logger task: waitpid error\n");
        }
    }
    else if (!strcmp(token, "socket task"))
    {
        token = strtok(NULL, colon_delimiter);
        printf("Killing socket task\n");
        int pid_to_kill = atoi(token);

        kill(pid_to_kill, SIGUSR1);

        int status;
        pid_t end_id = waitpid(pid_to_kill, &status, 0);
        if (end_id == pid_to_kill)
        {
            if (WIFEXITED(status))
                printf("Socket task successfully killed\n");
        }
        else
```

```
            {
                perror("Socket task: waitpid error\n");
            }
        }
    }

    if (buffer)
        free(buffer);

    if (fp_pid_info_file)
        fclose(fp_pid_info_file);
}

void turn_on_usr_led(void)
{
    printf("Turning on USR led\n");

#if 0
    FILE *fp_brightness_file =
fopen("/sys/class/leds/beaglebone:green:usr2/brightness", "w");
    if (fp_brightness_file == NULL)
    {
        perror("fopen failed");
        printf("Failed to open brightness file\n");
        return;
    }

    int on_value = 1;

    fwrite(&on_value, 1, sizeof(int), fp_brightness_file);

    fclose(fp_brightness_file);
#endif

    char led_turn_on_cmd[128];
    memset(led_turn_on_cmd, '\0', sizeof(led_turn_on_cmd));

    sprintf(led_turn_on_cmd, "sudo sh -c 'echo 1 >
/sys/class/leds/beaglebone:green:usr3/brightness'");
    system(led_turn_on_cmd);
}

void log_task_unalive_msg_to_log_file(char *task_name)
{
    int msg_priority;

    /* Set the message queue attributes */
    struct mq_attr logger_mq_attr = { .mq_flags = 0,
                                      .mq_maxmsg =
MSG_QUEUE_MAX_NUM_MSGS,  // Max number of messages on queue
                                      .mq_msgsize =
MSG_QUEUE_MAX_MSG_SIZE  // Max. message size
                                    };

    logger_mq_handle = mq_open(MSG_QUEUE_NAME, O_RDWR, S_IRWXU,
&logger_mq_attr);

    char main_task_data_msg[MSG_MAX_LEN];
    memset(main_task_data_msg, '\0', sizeof(main_task_data_msg));
```

```
    sprintf(main_task_data_msg, "%s task not alive", task_name);

    struct _logger_msg_struct_ logger_msg;
    memset(&logger_msg, '\0', sizeof(logger_msg));
    strcpy(logger_msg.message, main_task_data_msg);
    strncpy(logger_msg.logger_msg_src_id, "Main", strlen("Main"));
    logger_msg.logger_msg_src_id[strlen("Main")] = '\0';
    strncpy(logger_msg.logger_msg_level, "Error", strlen("Error"));
    logger_msg.logger_msg_level[strlen("Error")] = '\0';

    msg_priority = 1;
    int num_sent_bytes = mq_send(logger_mq_handle, (char *)&logger_msg,
                            sizeof(logger_msg), msg_priority);
    if (num_sent_bytes < 0)
        perror("mq_send failed");
}

void sig_handler(int sig_num)
{
    char buffer[MSG_BUFF_MAX_LEN];
    memset(buffer, '\0', sizeof(buffer));

    if (sig_num == SIGINT || sig_num == SIGUSR1)
    {
        if (sig_num == SIGINT)
            printf("Caught signal %s in temperature task\n", "SIGINT");
        else if (sig_num == SIGUSR1)
            printf("Caught signal %s in temperature task\n", "SIGKILL");

        kill_already_created_processes();

        mq_close(logger_mq_handle);

        g_kill_main_task = 1;
    }
}

/*****************************************************************************
*
* Author:       Pavan Dhareshwar & Sridhar Pavithrapu
* Date:         03/11/2018
* File:         main_task.h
* Description:  Header file containing the macros, structs/enums, globals
                and function prototypes for source file main_task.c
*****************************************************************************
/

#ifndef _MAIN_TASK_H_
#define _MAIN_TASK_H_

/*---------------------------------- INCLUDES ---------------------------
----*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <unistd.h>
#include <signal.h>

#include <sys/socket.h>
```

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#include <netinet/in.h>
#include <arpa/inet.h>

#include <mqueue.h>

#include <semaphore.h>

/*---------------------------------- INCLUDES ----------------------------
----*/

/*----------------------------------- MACROS -----------------------------
----*/
#define TEMP_TASK_PORT_NUM                    8650
#define TEMP_TASK_QUEUE_SIZE              10

#define LIGHT_TASK_PORT_NUM               8660
#define LIGHT_TASK_QUEUE_SIZE                  10

#define SOCKET_TASK_PORT_NUM                   8670
#define SOCKET_TASK_QUEUE_SIZE                 10

#define LOGGER_TASK_PORT_NUM                   8680
#define LOGGER_TASK_QUEUE_SIZE                 10

#define SENSOR_TASK_SOCK_IP_ADDR           "127.0.0.1"

#define BUFF_SIZE                          1024
#define MSG_MAX_LEN                        128
#define MSG_BUFF_MAX_LEN                   1024

#define MSG_QUEUE_NAME                     "/logger_task_mq"
#define MSG_QUEUE_MAX_NUM_MSGS             5
#define MSG_QUEUE_MAX_MSG_SIZE             1024

#define TEMP_TASK_UNALIVE_CNT_LOG_LIMIT    5
#define LIGHT_TASK_UNALIVE_CNT_LOG_LIMIT   5
#define LOGGER_TASK_UNALIVE_CNT_LOG_LIMIT  5
#define SOCK_TASK_UNALIVE_CNT_LOG_LIMIT    5

#define TEMP_SENSOR_TASK_EXEC_NAME         "./temp_task &"
#define LIGHT_SENSOR_TASK_EXEC_NAME        "./light_task &"
#define SOCKET_TASK_EXEC_NAME              "./socket_task &"
#define LOGGER_TASK_EXEC_NAME              "./logger_task &"

#define LOGGER_ATTR_LEN                    32

/*----------------------------------- MACROS -----------------------------
----*/

/*---------------------------------- GLOBALS -----------------------------
----*/
int temp_task_sockfd, light_task_sockfd;
int socket_task_sockfd, logger_task_sockfd;

struct sockaddr_in temp_task_sock_addr, light_task_sock_addr;
struct sockaddr_in socket_task_sock_addr, logger_task_sock_addr;
```

```
int temp_task_unalive_count, light_task_unalive_count;
int logger_task_unalive_count, socket_task_unalive_count;

mqd_t logger_mq_handle;

sig_atomic_t g_kill_main_task;

/*---------------------------------- GLOBALS ----------------------------
----*/

/*---------------------------- STRUCTURES/ENUMERATIONS ------------------
----*/
struct _logger_msg_struct_
{
    char message[MSG_MAX_LEN];
    char logger_msg_src_id[LOGGER_ATTR_LEN];
    char logger_msg_level[LOGGER_ATTR_LEN];
};

/*---------------------------- STRUCTURES/ENUMERATIONS ------------------
----*/

/*---------------------------- FUNCTION PROTOTYPES ----------------------
----*/

/**
 *  @brief Initialize sub tasks interface socket
 *
 *  For the main task to check the status of each of the remaining tasks,
it
 *  sends a heartbeat message to each of these tasks and when it receives
a
 *  reply, it knows that the task is alive. For the main task to check
the
 *  status, it uses socket as an IPC mechanism.
 *
 *  This function creates a socket between main task and the sensor task
for
 *  communication.
 *
 *  @param sock_fd                 : pointer socket file descriptor
 *  @param sock_addr_struct        : sockaddr_in structure pointer
 *  @param port_num                : port number associated with the
socket
 *
 *  @return void
 *
 */
void initialize_sub_task_socket(int *sock_fd, struct sockaddr_in
*sock_addr_struct,
                                int port_num);

/**
 *  @brief Check status of a specified sub task
 *
 *  This function checks the status of the specified subtask to see if it
 *  is alive
 *
 *  @param sock_fd                 : socket file descriptor for the task
```

```
 *  @param task_name               : name of the subtask
 *
 *  @return void
 *
 */
void check_subtask_status(int sock_fd, char *task_name);

/**
 *  @brief Check status of sub tasks
 *
 *  This function checks the status of each of the subtasks to see if
they
 *  are alive
 *
 *  @param void
 *
 *  @return void
 *
 */
void check_status_of_sub_tasks(void);

/**
 *  @brief Log unalive message to logger task message queue
 *
 *  This function logs a message to the logger task message queue if a
certain
 *  process isn't alive when checked for a predefined number of times
 *
 *  @param task_name               : name of the subtask
 *
 *  @return void
 *
 */
void log_task_unalive_msg_to_log_file(char *task_name);

/**
 *  @brief Create sub processes
 *
 *  This function creates the temperature, light, logger and socket sub-
processess
 *
 *  @param void
 *
 *  @return void
 *
 */
void create_sub_processes(void);

/**
 *  @brief Create a specific sub process
 *
 *  This function creates a new task as per the name specified by @param
task_name
 *
 *  @param task_name                 : name of the subtask
 *
 *  @return void
 *
 */
void create_sub_process(char *process_name);
```

```
/**
 *  @brief Perform start-up tests
 *
 *  This function performs the start-up tests to ensure that the
hardware, processes
 *  and communication primitivies are working. If any of the start-up
test fails, the
 *  already existing processes and threads are killed and some cleanup is
done
 *
 *  @param void
 *
 *  @return void
 *
 */
void perform_startup_test(void);

/**
 *  @brief Perform start-up test for a sub task
 *
 *  This function performs the start-up tests to ensure that the
hardware, threads
 *  and communication primitives of the specified sub task are working.
 *
 *  @param sock_fd                    : socket file descriptor
 *  @param proc_name                  : process name
 *
 *  @return void
 *
 */
int perform_sub_task_startup_test(int sock_fd, char *proc_name);

/**
 *  @brief Stop entire system
 *
 *  This function is called when a certain start-up test fails and
performs some
 *  clean-up and exits.
 *
 *  @param void
 *
 *  @return void
 *
 */
void stop_entire_system(void);

/**
 *  @brief Kill already created processes
 *
 *  This function kills all the created processes by the main task before
the start-up
 *  test is triggered, as part of clean-up and exit of the entire system.
 *
 *  @param void
 *
 *  @return void
 *
 */
void kill_already_created_processes(void);
```

```
/**
 *  @brief Turn on user led
 *
 *  This function turns on a user led on the beagle bone green to
indicate of the
 *  system failure to start-up
 *
 *  @param void
 *
 *  @return void
 *
 */
void turn_on_usr_led(void);

/**
 *  @brief Write pid of created processes to a file
 *
 *  This function writes the pid of the sub processes created by main
task to a file
 *
 *  @param proc_name                  : name of the child process
 *  @param child_pid                  : pid of the child process
 *
 *  @return void
 *
 */
void write_pid_to_file(char *proc_name, pid_t child_pid);

/**
 *  @brief Signal handler for main task
 *
 *  This function handles the reception of SIGKILL and SIGINT signal to
the
 *  temperature task and terminates all the threads, closes the I2C file
descriptor
 *  and logger message queue handle and exits.
 *
 *  @param sig_num              : signal number
 *
 *  @return void
*/

void sig_handler(int sig_num);
/*---------------------------- FUNCTION PROTOTYPES ----------------------
----*/

#endif // _MAIN_TASK_H_
/*****************************************************************************
*
* Author:       Pavan Dhareshwar & Sridhar Pavithrapu
* Date:         03/08/2018
* File:         logger_task.h
* Description:  Header file containing the macros, structs/enums, globals
                and function prototypes for source file logger_task.c
*****************************************************************************
/

#ifndef _LOGGER_TASK_H_
#define _LOGGER_TASK_H_
```

```
/*---------------------------------- INCLUDES ----------------------------
----*/
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <stdint.h>
#include <string.h>
#include <time.h>

#include <unistd.h>
#include <fcntl.h>
#include <pthread.h>
#include <signal.h>

#include <sys/types.h>
#include <sys/stat.h>
#include <sys/ipc.h>
#include <sys/types.h>
#include <sys/msg.h>

#include <mqueue.h>

#include <netinet/in.h>
#include <arpa/inet.h>
/*---------------------------------- INCLUDES ----------------------------
----*/

/*---------------------------------- MACROS ----------------------------
----*/
// Message queue attribute macros
#define MSG_QUEUE_MAX_NUM_MSGS              5
#define MSG_QUEUE_MAX_MSG_SIZE              1024
#define MSG_QUEUE_NAME                      "/logger_task_mq"

#define LOGGER_FILE_PATH                    "./"
#define LOGGER_FILE_NAME                    "logger_file.txt"

#define LOG_MSG_PAYLOAD_SIZE                256
#define MSG_MAX_LEN                         128

#define MSG_BUFF_MAX_LEN                    1024

#define LOGGER_FILE_PATH_LEN                256
#define LOGGER_FILE_NAME_LEN                64

#define SOCKET_HB_PORT_NUM                  8680
#define SOCKET_HB_LISTEN_QUEUE_SIZE         10

#define MSG_TYPE_TEMP_DATA                  0
#define MSG_TYPE_LUX_DATA                   1
#define MSG_TYPE_SOCK_DATA                  2
#define MSG_TYPE_MAIN_DATA                  3

#define LOGGER_ATTR_LEN                     32

/*---------------------------------- MACROS ----------------------------
----*/
```

```
/*---------------------------------- GLOBALS ----------------------------
----*/
mqd_t logger_mq_handle;
int logger_fd;
pthread_t logger_thread_id, socket_hb_thread_id;

sig_atomic_t g_sig_kill_logger_thread, g_sig_kill_sock_hb_thread;

int logger_task_initialized = 0;

/*---------------------------------- GLOBALS ----------------------------
----*/

/*---------------------------- STRUCTURES/ENUMERATIONS -----------------
----*/
struct _logger_msg_struct_
{
    char message[MSG_MAX_LEN];
    char logger_msg_src_id[LOGGER_ATTR_LEN];
    char logger_msg_level[LOGGER_ATTR_LEN];
};

/*---------------------------- STRUCTURES/ENUMERATIONS -----------------
----*/

/*---------------------------- FUNCTION PROTOTYPES ---------------------
----*/
/**
 *  @brief Initialize the logger task
 *
 *  This funcition will create the message queue for logger task and
 *  open a file handle of logger file for writing. (If the logger file
 *  already exists, it is deleted and a fresh one is created).
 *
 *  @param void
 *
 *  @return 0  : if sensor initialization is a success
            -1 : if sensor initialization fails
*/
int logger_task_init();

/**
 *  @brief Read from configuration file for the logger task
 *
 *  This function reads the configuration parameters for the logger task
file
 *  and sets-up the logger file as per this configuration
 *
 *  @param file     : name of the config file
 *
 *  @return void
 *
 */
int read_logger_conf_file(char *file);

/**
 *  @brief Create logger and hearbeat socket threads for logger task
 *
 *  The logger task is made multi-threaded with
```

```
 *     1. logger thread responsible for reading messages from its message
queue
 *        and logging it to a file.
 *     2. socket heartbeat responsible for communicating with main task,
 *        to log heartbeat every time its requested by main task.
 *
 *  @param void
 *
 *  @return 0  : thread creation success
 *          -1 : thread creation failed
 *
 */
int create_threads(void);

/**
 *  @brief Entry point and executing entity for logger thread
 *
 *  The logger thread starts execution by invoking this
function(start_routine)
 *
 *  @param arg : argument to start_routine
 *
 *  @return void
 *
 */
void *logger_thread_func(void *arg);

/**
 *  @brief Entry point and executing entity for socket thread
 *
 *  The socket thread for heartbeat starts execution by invoking this
function(start_routine)
 *
 *  @param arg : argument to start_routine
 *
 *  @return void
 *
 */
void *socket_hb_thread_func(void *arg);

/**
 *  @brief Create the socket and initialize
 *
 *  This function create the socket for the given socket id.
 *
 *  @param sock_fd              : socket file descriptor
 *         server_addr_struct   : server address of the socket
 *         port_num             : port number in which the socket is
communicating
 *         listen_qsize         : number of connections the socket is
accepting
 *
 *  @return void
*/
void init_sock(int *sock_fd, struct sockaddr_in *server_addr_struct,
               int port_num, int listen_qsize);

#if 0
void write_test_msg_to_logger();
#endif
```

```
/**
 *  @brief Read message from logger message queue
 *
 *  This function will read messages from its message queue and log it to
a file
 *
 *  @param void
 *
 *  @return void
*/
void read_from_logger_msg_queue(void);

/**
 *  @brief Cleanup of the logger sensor
 *
 *  This function will close the message queue and the logger file handle
 *
 *  @param void
 *
 *  @return void
*/
void logger_task_exit(void);

/**
 *  @brief Signal handler for temperature task
 *
 *  This function handles the reception of SIGKILL and SIGINT signal to
the
 *  temperature task and terminates all the threads, closes the I2C file
descriptor
 *  and logger message queue handle and exits.
 *
 *  @param sig_num              : signal number
 *
 *  @return void
*/

void sig_handler(int sig_num);
#endif // _LOGGER_TASK_H_
/*************************************************************************
*
* Author:       Pavan Dhareshwar & Sridhar Pavithrapu
* Date:         03/08/2018
* File:         logger_task.c
* Description:  Source file describing the functionality and
implementation
*               of logger task.
*************************************************************************
/

#include "logger_task.h"

int main(void)
{

    logger_task_initialized = 0;

    int init_status = logger_task_init();
    if (init_status == -1)
```

```
    {
        printf("logger task initialization failed\n");
        exit(1);
    }

    //write_test_msg_to_logger();

    int thread_create_status = create_threads();
    if (thread_create_status)
    {
        printf("Thread creation failed\n");
    }
    else
    {
        printf("Thread creation success\n");
    }

    if (signal(SIGINT, sig_handler) == SIG_ERR)
        printf("SigHandler setup for SIGINT failed\n");

    if (signal(SIGUSR1, sig_handler) == SIG_ERR)
        printf("SigHandler setup for SIGKILL failed\n");

    g_sig_kill_logger_thread = 0;
    g_sig_kill_sock_hb_thread = 0;

    pthread_join(logger_thread_id, NULL);
    pthread_join(socket_hb_thread_id, NULL);

    logger_task_exit();

    return 0;
}

int logger_task_init()
{
    /* In the logger task init function, we create the message queue */

    /* Set the message queue attributes */
    struct mq_attr logger_mq_attr = { .mq_flags = 0,
                                      .mq_maxmsg =
MSG_QUEUE_MAX_NUM_MSGS,  // Max number of messages on queue
                                      .mq_msgsize =
MSG_QUEUE_MAX_MSG_SIZE  // Max. message size
                                    };


    logger_mq_handle = mq_open(MSG_QUEUE_NAME, O_CREAT | O_RDWR, S_IRWXU,
&logger_mq_attr);
    if (logger_mq_handle < 0)
    {
        perror("Logger message queue create failed");
        return -1;
    }

    printf("Logger message queue successfully created\n");

    char filename[100];
    memset(filename, '\0', sizeof(filename));
    int conf_file_read_status = read_logger_conf_file(filename);
```

```
    if (conf_file_read_status != 0)
    {
        printf("Logger task config file read failed. Using default log
file path and name\n");
        sprintf(filename, "%s%s", LOGGER_FILE_PATH, LOGGER_FILE_NAME);
    }

    if (open(filename, O_RDONLY) != -1)
    {
        printf("Logger file exists. Deleting existing file.\n");
        remove(filename);
        sync();
    }

    printf("Trying to create file %s\n", filename);
    logger_fd = creat(filename ,(S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH));
    if (logger_fd == -1)
    {
        perror("Logger file open failed");
        return -1;
    }
    else
    {
        printf("Logger file open success\n");
    }

    logger_task_initialized = 1;

    return 0;
}

int read_logger_conf_file(char *file)
{
    FILE *fp_conf_file = fopen("./logger_task_conf_file.txt", "r");
    if (fp_conf_file == NULL)
    {
        perror("file open failed");
        printf("File %s open failed\n", "logger_task_conf_file.txt");
        return -1;
    }

    char logger_file_path[LOGGER_FILE_PATH_LEN];
    char logger_file_name[LOGGER_FILE_NAME_LEN];
    char *buffer;
    size_t num_bytes = 120;
    char equal_delimiter[] = "=";
    ssize_t bytes_read;

    memset(logger_file_path, '\0', sizeof(logger_file_path));
    memset(logger_file_name, '\0', sizeof(logger_file_name));

    buffer = (char *)malloc(num_bytes*sizeof(char));

    while ((bytes_read = getline(&buffer, &num_bytes, fp_conf_file)) != -
1)
    {
        char *token = strtok(buffer, equal_delimiter);

        if (!strcmp(token, "LOGGER_FILE_PATH"))
        {
```

```
            token = strtok(NULL, equal_delimiter);
            strcpy(logger_file_path, token);
            int len = strlen(logger_file_path);
            if (logger_file_path[len-1] == '\n')
                logger_file_path[len-1] = '\0';
        }
        else if (!strcmp(token, "LOGGER_FILE_NAME"))
        {
            token = strtok(NULL, equal_delimiter);
            strcpy(logger_file_name, token);
            int len = strlen(logger_file_name);
            if (logger_file_name[len-1] == '\n')
                logger_file_name[len-1] = '\0';
        }
    }

    strcpy(file, logger_file_path);
    strcat(file, logger_file_name);

    if (buffer)
        free(buffer);

    if (fp_conf_file)
        fclose(fp_conf_file);

    return 0;
}

int create_threads(void)
{
    int logger_t_creat_ret_val = pthread_create(&logger_thread_id, NULL,
&logger_thread_func, NULL);
    if (logger_t_creat_ret_val)
    {
        perror("Sensor thread creation failed");
        return -1;
    }

    int sock_hb_t_creat_ret_val = pthread_create(&socket_hb_thread_id,
NULL, &socket_hb_thread_func, NULL);
    if (sock_hb_t_creat_ret_val)
    {
        perror("Socket heartbeat thread creation failed");
        return -1;
    }

    return 0;
}

void *logger_thread_func(void *arg)
{
    while(!g_sig_kill_logger_thread)
    {
        /* This function will continously read from the logger task
message
        ** queue and write it to logger file */
        read_from_logger_msg_queue();
    }

    pthread_exit(NULL);
```

```
}

void *socket_hb_thread_func(void *arg)
{
    int sock_hb_fd;
    struct sockaddr_in sock_hb_address;
    int sock_hb_addr_len = sizeof(sock_hb_address);

    init_sock(&sock_hb_fd, &sock_hb_address, SOCKET_HB_PORT_NUM,
SOCKET_HB_LISTEN_QUEUE_SIZE);


    int accept_conn_id;
    printf("Waiting for request...\n");
    if ((accept_conn_id = accept(sock_hb_fd, (struct sockaddr
*)&sock_hb_address,
                    (socklen_t*)&sock_hb_addr_len)) < 0)
    {
        perror("accept failed");
        //pthread_exit(NULL);
    }

    char recv_buffer[MSG_BUFF_MAX_LEN];
    char send_buffer[] = "Alive";

    while (!g_sig_kill_sock_hb_thread)
    {
        memset(recv_buffer, '\0', sizeof(recv_buffer));

        size_t num_read_bytes = read(accept_conn_id, &recv_buffer,
sizeof(recv_buffer));

        if (!strcmp(recv_buffer, "heartbeat"))
        {
                 ssize_t num_sent_bytes = send(accept_conn_id,
send_buffer, strlen(send_buffer), 0);
            if (num_sent_bytes < 0)
                perror("send failed");
        }
        else if (!strcmp(recv_buffer, "startup_check"))
        {
            /* For the sake of start-up check, because we have the
temperature sensor initialized
            ** by the time this thread is spawned. So we perform a
"get_temp_data" call to see if
            ** everything is working fine */
            if (logger_task_initialized == 1)
                strcpy(send_buffer, "Initialized");
            else
                strcpy(send_buffer, "Uninitialized");

            ssize_t num_sent_bytes = send(accept_conn_id, send_buffer,
strlen(send_buffer), 0);
            if (num_sent_bytes < 0)
                perror("send failed");
        }
    }
}

#if 0
```

```
void write_test_msg_to_logger()
{
    struct _logger_msg_struct_ logger_msg = {0};

    const char test_msg[] = "Testing if msg queue comm works";
    strcpy(logger_msg.message, test_msg);
    logger_msg.msg_len = strlen(test_msg);

    logger_msg.logger_msg_type = MSG_TYPE_TEMP_DATA;

    int msg_priority = 1;
    int num_sent_bytes = mq_send(logger_mq_handle, (char *)&logger_msg,
                                 sizeof(logger_msg), msg_priority);

    if (num_sent_bytes < 0)
        perror("mq_send failed");
}
#endif

void init_sock(int *sock_fd, struct sockaddr_in *server_addr_struct,
               int port_num, int listen_qsize)
{
    int serv_addr_len = sizeof(struct sockaddr_in);

    /* Create the socket */
    if ((*sock_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0)
    {
        perror("socket creation failed");
        pthread_exit(NULL); // Change these return values from
pthread_exit
    }

    int option = 1;
    if(setsockopt(*sock_fd, SOL_SOCKET, (SO_REUSEPORT | SO_REUSEADDR),
(void *)&option, sizeof(option)) < 0)
    {
        perror("setsockopt failed");
        pthread_exit(NULL);
    }

    server_addr_struct->sin_family = AF_INET;
    server_addr_struct->sin_addr.s_addr = INADDR_ANY;
    server_addr_struct->sin_port = htons(port_num);

    if (bind(*sock_fd, (struct sockaddr *)server_addr_struct,
                                          sizeof(struct
sockaddr_in))<0)
    {
        perror("bind failed");
        pthread_exit(NULL);
    }

    if (listen(*sock_fd, listen_qsize) < 0)
    {
        perror("listen failed");
        pthread_exit(NULL);
    }

}
```

```
void read_from_logger_msg_queue(void)
{
    char recv_buffer[MSG_MAX_LEN];
    memset(recv_buffer, '\0', sizeof(recv_buffer));

    int msg_priority;

    int num_recv_bytes;
    while ((num_recv_bytes = mq_receive(logger_mq_handle, (char
*)&recv_buffer,
                                        MSG_QUEUE_MAX_MSG_SIZE,
&msg_priority)) != -1)
    {
        if (num_recv_bytes < 0)
        {
            perror("mq_receive failed");
            return;
        }

#if 0
        printf("Message received: %s, msg_src: %s, message level: %s\n",
            (((struct _logger_msg_struct_ *)&recv_buffer)->message),
            (((struct _logger_msg_struct_ *)&recv_buffer)-
>logger_msg_src_id),
            (((struct _logger_msg_struct_ *)&recv_buffer)-
>logger_msg_level));
#endif

        time_t tval = time(NULL);
        struct tm *cur_time = localtime(&tval);

        char timestamp_str[32];
        memset(timestamp_str, '\0', sizeof(timestamp_str));

        sprintf(timestamp_str, "%02d:%02d:%02d", cur_time->tm_hour,
cur_time->tm_min, cur_time->tm_sec);

        char msg_to_write[LOG_MSG_PAYLOAD_SIZE];
        memset(msg_to_write, '\0', sizeof(msg_to_write));

        sprintf(msg_to_write, "Timestamp: %s | Message_Src: %s |
Message_Type: %s | Message: %s\n",
            timestamp_str, (((struct _logger_msg_struct_ *)&recv_buffer)-
>logger_msg_src_id),
            (((struct _logger_msg_struct_ *)&recv_buffer)-
>logger_msg_level),
            (((struct _logger_msg_struct_ *)&recv_buffer)->message));

        printf("Message to write: %s\n", msg_to_write);
        int num_written_bytes = write(logger_fd, msg_to_write,
strlen(msg_to_write));
    }

}

void sig_handler(int sig_num)
{
    char buffer[MSG_BUFF_MAX_LEN];
    memset(buffer, '\0', sizeof(buffer));
```

```
    if (sig_num == SIGINT || sig_num == SIGUSR1)
    {
        if (sig_num == SIGINT)
            printf("Caught signal %s in logger task\n", "SIGINT");
        else if (sig_num == SIGUSR1)
            printf("Caught signal %s in logger task\n", "SIGKILL");

        g_sig_kill_logger_thread = 1;
        g_sig_kill_sock_hb_thread = 1;

        //pthread_join(sensor_thread_id, NULL);
        //pthread_join(socket_thread_id, NULL);
        //pthread_join(socket_hb_thread_id, NULL);

        mq_close(logger_mq_handle);

        exit(0);
    }
}

void logger_task_exit(void)
{
    int mq_close_status = mq_close(logger_mq_handle);
    if (mq_close_status == -1)
        perror("Logger message queue close failed");

    if (logger_fd)
        close(logger_fd);

}
/*************************************************************************
*
* Author:       Pavan Dhareshwar & Sridhar Pavithrapu
* Date:         03/07/2018
* File:         temperature_sensor.h
* Description:  Header file containing the macros, structs/enums, globals
                and function prototypes for source file
temperature_sensor.c
*************************************************************************
/

#ifndef _TEMPERATURE_SENSOR_TASK_H_
#define _TEMPERATURE_SENSOR_TASK_H_


/*---------------------------------- INCLUDES ----------------------------
----*/

#include <errno.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <linux/i2c-dev.h>
#include <fcntl.h>

#include <unistd.h>
#include <signal.h>

#include <netinet/in.h>
```

```
#include <arpa/inet.h>

#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/msg.h>
#include <sys/ipc.h>
#include <sys/socket.h>

#include <mqueue.h>
#include "wrapper.h"

/*---------------------------------- GLOBALS ----------------------------
----*/
char i2c_name[10];
int sensor_thread_id, socket_thread_id, socket_hb_thread_id;
int file_descriptor;
int default_config_byte_one = 0XA0;
int default_config_byte_two = 0X60;

int temp_sensor_initialized;

sig_atomic_t g_sig_kill_sensor_thread, g_sig_kill_sock_thread,
g_sig_kill_sock_hb_thread;
mqd_t logger_mq_handle;

/*---------------------------------- MACROS ----------------------------
----*/

#define I2C_SLAVE_ADDR                    0b01001000
#define I2C_SLAVE_DEV_NAME              "/dev/i2c-2"


#define I2C_TEMP_SENSOR_TEMP_DATA_REG  0b00000000 // Temperature data
register (read-only)
#define I2C_TEMP_SENSOR_CONFIG_REG         0b00000001   // command
register
#define I2C_TEMP_SENSOR_TLOW_REG      0b00000010   // T_low register
#define I2C_TEMP_SENSOR_THIGH_REG     0b00000011   // T_high register

#define SERVER_PORT_NUM                 8081
#define SERVER_LISTEN_QUEUE_SIZE        5

#define MSG_BUFF_MAX_LEN                1024
#define MSG_MAX_LEN                     128

#define MSG_QUEUE_NAME                  "/logger_task_mq"
#define MSG_QUEUE_MAX_NUM_MSGS          5
#define MSG_QUEUE_MAX_MSG_SIZE          1024

#define SOCK_REQ_MSG_API_MSG_LEN        64

#define SOCKET_HB_PORT_NUM              8650
#define SOCKET_HB_LISTEN_QUEUE_SIZE     5

#define LOGGER_ATTR_LEN                 32

/*--------------------------- STRUCTURES/ENUMERATIONS -----------------
----*/
```

```
typedef enum{

      TEMP_CELSIUS = 0,
      TEMP_KELVIN = 1,
      TEMP_FARENHEIT = 2

}tempformat_e;

struct _logger_msg_struct_
{
    char message[MSG_MAX_LEN];
    char logger_msg_src_id[LOGGER_ATTR_LEN];
    char logger_msg_level[LOGGER_ATTR_LEN];
};

enum _req_recipient_
{
    REQ_RECP_TEMP_TASK,
    REQ_RECP_LIGHT_TASK
};

struct _socket_req_msg_struct_
{
    char req_api_msg[SOCK_REQ_MSG_API_MSG_LEN];
    enum _req_recipient_ req_recipient;
    int params;
};

/*---------------------------- FUNCTION PROTOTYPES ----------------------
----*/
/**
 *  @brief Write pointer register of temperature sensor
 *
 *  This function will open the i2c bus write operation of pointer
register
 *  of Temperature sensor.
 *
 *  @param value : value to be written into pointer register
 *
 *  @return void
*/
void write_pointer_register(uint8_t value);

/**
 *  @brief Write temperature high and low register of temperature sensor
 *
 *  This function will open the i2c bus write operation of temperature
high and
 *  low register of Temperature sensor.
 *
 *  @param sensor_register  : register address of either temperature high
or low register
 *            data                    : value to be written into
register
 *
 *  @return void
*/
void write_temp_high_low_register(int sensor_register, int16_t data );

/**
```

```
 *  @brief Write config register of temperature sensor
 *
 *  This function will open the i2c bus write operation of config
register of Temperature sensor.
 *
 *  @param data                  : value to be written into register
 *
 *  @return void
*/
void write_config_register_on_off(uint8_t data );

/**
 *  @brief Write config register of temperature sensor
 *
 *  This function will open the i2c bus write operation of config
register for em bits of Temperature sensor.
 *
 *  @param data                  : value to be written for em bits of
config register
 *
 *  @return void
*/
void write_config_register_em(uint8_t data );

/**
 *  @brief Write config register of temperature sensor
 *
 *  This function will open the i2c bus write operation of config
register for conversion rate of Temperature sensor.
 *
 *  @param data                  : value to be written for conversion
rate of config register
 *
 *  @return void
*/
void write_config_register_conversion_rate(uint8_t data );

/**
 *  @brief Write config register of temperature sensor
 *
 *  This function will open the i2c bus write operation of default values
into config register of Temperature sensor.
 *
 *  @param data                  : void
 *
 *  @return void
*/
void write_config_register_default( );

/**
 *  @brief Read temperature high and low register of temperature sensor
 *
 *  This function will open the i2c bus for read of temperature high and
 *  low register of Temperature sensor.
 *
 *  @param sensor_register  : register address of either temperature high
or low register
 *            data                    : value to be read from register
 *
 *  @return reg_val   : if register read is successful
```

```
 *          -1        : if register read fails
*/
int16_t read_temp_high_low_register(int sensor_register);

/**
 *  @brief Read temperature config of temperature sensor
 *
 *  This function will open the i2c bus for read config
 *  register of Temperature sensor.
 *
 *  @param void
 *
 *  @return reg_val   : if register read is successful
 *          -1        : if register read fails
*/
uint16_t read_temp_config_register();

/**
 *  @brief Read temperature data of temperature sensor
 *
 *  This function will open the i2c bus for read temperature data
 *  register of Temperature sensor.
 *
 *  @param void
 *
 *  @return temp_value  : if register read is successful
            -1 : if sensor initialization fails
*/
float read_temperature_data_register(int format);

/**
 *  @brief Initialize the temperature sensor
 *
 *  This function will open the i2c bus for read and write operation and
 *  initialize the communication with the peripheral.
 *
 *  @param void
 *
 *  @return 0  : if sensor initialization is a success
            -1 : if sensor initialization fails
*/
int temp_sensor_init();

/**
 *  @brief Log the temperature value
 *
 *  This function writes the temperature value calculated to logger
message queue
 *
 *  @param temp_data     : temperature data to be logged
 *
 *  @return void
*/
void log_temp_data(float temp_data);

/**
 *  @brief Entry point and executing entity for sensor thread
 *
 *  The sensor thread starts execution by invoking this
function(start_routine)
```

```
 *
 *  @param arg : argument to start_routine
 *
 *  @return void
 *
 */
void *sensor_thread_func(void *arg);

/**
 *  @brief Entry point and executing entity for socket thread
 *
 *  The socket thread starts execution by invoking this
function(start_routine)
 *
 *  @param arg : argument to start_routine
 *
 *  @return void
 *
 */
void *socket_thread_func(void *arg);

/**
 *  @brief Entry point and executing entity for socket thread
 *
 *  The socket thread for heartbeat starts execution by invoking this
function(start_routine)
 *
 *  @param arg : argument to start_routine
 *
 *  @return void
 *
 */
void *socket_hb_thread_func(void *arg);

/**
 *  @brief Create sensor,socket and heartbeat threads for temperature
task
 *
 *  The temperature task is made multi-threaded with
 *     1. sensor thread responsible for communicating via I2C interface
 *        with the temperature sensor to get temperature data and a
socket
 *        thread.
 *     2. socket thread responsible for communicating with socket thread
and
 *        serve request from external application forwarded via socket
task.
 *      3. socket heartbeat responsible for communicating with main
task,
 *           to log heartbeat every time its requested by main task.
 *
 *  @param void
 *
 *  @return 0  : thread creation success
 *          -1 : thread creation failed
 *
 */
int create_threads(void);

/**
```

```
 *  @brief Create the socket and initialize
 *
 *  This function create the socket for the given socket id.
 *
 *  @param sock_fd                      : socket file descriptor
 *            server_addr_struct  : server address of the socket
 *            port_num                  : port number in which the socket
is communicating
 *            listen_qsize        : number of connections the socket is
accepting
 *
 *  @return void
*/
void init_sock(int *sock_fd, struct sockaddr_in *server_addr_struct,
               int port_num, int listen_qsize);

/**
 *  @brief Signal handler for temperature task
 *
 *  This function handles the reception of SIGKILL and SIGINT signal to
the
 *  temperature task and terminates all the threads, closes the I2C file
descriptor
 *  and logger message queue handle and exits.
 *
 *  @param sig_num                      : signal number
 *
 *  @return void
*/

void sig_handler(int sig_num);

#endif // #ifndef _TEMPERATURE_SENSOR_TASK_H_
/*************************************************************************
*
* Author:       Pavan Dhareshwar & Sridhar Pavithrapu
* Date:         03/07/2018
* File:         temperature_sensor.c
* Description:  Source file describing the functionality and
implementation
*               of temperature sensor task.
*************************************************************************
/

#include "temperature_sensor.h"
#include "wrapper.h"

void write_pointer_register(uint8_t value){

      if (wrapper_write(file_descriptor, &value, 1) != 1) {
            perror("wrapper_write pointer register error\n");
      }
}

void write_temp_high_low_register(int sensor_register, int16_t data ){

      /* Writing to the pointer register for reading T_High/T_low
register */
      write_pointer_register(sensor_register);
```

```
	/* Writing the T_High/T_low register value */
	if (wrapper_write(file_descriptor, &data, 2) != 2) {
		perror("T-low register wrapper_write error");
	}
}


void write_config_register_on_off(uint8_t data ){

	/* Writing to the pointer register for configuration register */
	write_pointer_register(I2C_TEMP_SENSOR_CONFIG_REG);
	if((data == 0) || (data == 1)){
		default_config_byte_one |= data;

		/* Writing data to the configuration register */
		if (wrapper_write(file_descriptor, &default_config_byte_one,
1) != 1) {
			perror("Configuration register wrapper_write error for
first byte");
		}

		if (wrapper_write(file_descriptor, &default_config_byte_two,
1) != 1) {
			perror("Configuration register wrapper_write error for
second byte");
		}
	}
}

void write_config_register_em(uint8_t data ){

	/* Writing to the pointer register for configuration register */
	write_pointer_register(I2C_TEMP_SENSOR_CONFIG_REG);
	if((data == 0) || (data == 1)){

	  uint16_t config_reg_data;
	  config_reg_data = read_temp_config_register();
	  printf("CONFIG_REG_DATA: %d\n", config_reg_data);

	  config_reg_data = config_reg_data & (uint16_t)(~0x10);

	  config_reg_data |= (uint16_t)(data << 4);

	  uint8_t config_high_data = (uint8_t)(config_reg_data >> 8);
	  uint8_t config_low_data = (uint8_t)(config_reg_data & 0XFF);

		//default_config_byte_two |= (data << 4);

		/* Writing data to the configuration register */
		if (wrapper_write(file_descriptor, &config_high_data, 1) != 1)
{
			perror("Configuration register wrapper_write error for
first byte");
		}

		if (wrapper_write(file_descriptor, &config_low_data, 1) != 1)
{
			perror("Configuration register wrapper_write error for
second byte");
		}
```

```
      }
}

void write_config_register_conversion_rate(uint8_t data ){

      /* Writing to the pointer register for configuration register */
      write_pointer_register(I2C_TEMP_SENSOR_CONFIG_REG);
      if((data >= 0) || (data <= 3)){
        uint16_t config_reg_data;
        config_reg_data = read_temp_config_register();
        config_reg_data = config_reg_data & (uint16_t)(~0xC0);

        config_reg_data |= (uint16_t)(data << 6);

        uint8_t config_high_data = (uint8_t)(config_reg_data >> 8);
        uint8_t config_low_data = (uint8_t)(config_reg_data & 0XFF);

           /* Writing data to the configuration register */
           if (wrapper_write(file_descriptor, &config_high_data, 1) != 1)
{
                perror("Configuration register wrapper_write error for
first byte");
           }

           if (wrapper_write(file_descriptor, &config_low_data, 1) != 1)
{
                perror("Configuration register wrapper_write error for
second byte");
           }
      }
}

void write_config_register_fault_bits(uint8_t data ){

      /* Writing to the pointer register for configuration register */
      write_pointer_register(I2C_TEMP_SENSOR_CONFIG_REG);
      if((data >= 0) || (data <= 3)){
        uint16_t config_reg_data;
        config_reg_data = read_temp_config_register();
        config_reg_data = config_reg_data & (uint16_t)(~0x1800);

        config_reg_data |= (uint16_t)(data << 11);

        uint8_t config_high_data = (uint8_t)(config_reg_data >> 8);
        uint8_t config_low_data = (uint8_t)(config_reg_data & 0XFF);

           /* Writing data to the configuration register */
           if (wrapper_write(file_descriptor, &config_high_data, 1) != 1)
{
                perror("Configuration register wrapper_write error for
first byte");
           }

           if (wrapper_write(file_descriptor, &config_low_data, 1) != 1)
{
                perror("Configuration register wrapper_write error for
second byte");
           }
      }
}
```

```
uint8_t read_config_register_fault_bits(){

    /* Reading fault bits of temperature config register */
    uint16_t config_value = read_temp_config_register();
    uint8_t return_value = (uint8_t)((config_value & 0x1800) >> 11);
    return return_value;

}

uint8_t read_config_register_em(){

    /* Reading em-bit of temperature config register */
    uint16_t config_value = read_temp_config_register();
#define TEMP_CONF_REG_EM_BM       0x10

    uint8_t return_value = (config_value & TEMP_CONF_REG_EM_BM) >> 4;
    return return_value;

}

uint8_t read_config_register_conversion_rate(){

    /* Reading conversion rate of temperature config register */
    uint16_t config_value = read_temp_config_register();
    uint8_t return_value = (uint8_t)((config_value & 0x00C0) >> 6);
    return return_value;
}

void write_config_register_default( ){

      /* Writing to the pointer register for configuration register */
      write_pointer_register(I2C_TEMP_SENSOR_CONFIG_REG);

      /* Writing data to the configuration register */
      if (wrapper_write(file_descriptor, &default_config_byte_one, 1) !=
1) {
            perror("Configuration register wrapper_write error for first
byte");
      }

      if (wrapper_write(file_descriptor, &default_config_byte_two, 1) !=
1) {
            perror("Configuration register wrapper_write error for second
byte");
      }
}

int16_t read_temp_high_low_register(int sensor_register){

      int16_t tlow_output_value;
     int8_t *ptr_tlow_val = (int8_t *)&tlow_output_value;
      int8_t data[2]={0};

      /* Writing to the pointer register for reading Tlow register */
      write_pointer_register(sensor_register);

      /* Reading the Tlow register value */
      if (wrapper_read(file_descriptor, data, 1) != 1) {
            perror("T-low register wrapper_read error");
```

```
      }

    printf("data[0]: %d, data[1]:%d\n", data[0], data[1]);

      tlow_output_value = ((int16_t)data[0] | ((int16_t)((data[1] & 0XF)
<< 8)));
      printf("T-low register value is: %d \n", tlow_output_value);

      return tlow_output_value;

}

uint16_t read_temp_config_register(){

      uint16_t temp_config_value;
      uint8_t data[2]={0};

      /* Writing to the pointer register for reading THigh register */
      write_pointer_register(I2C_TEMP_SENSOR_CONFIG_REG);

      /* Reading the THigh register value */
      if (wrapper_read(file_descriptor, data, 2) != 2) {
            perror("Temperature configuration register wrapper_read
error");
      }

    printf("data[0]: %d, data[1]:%d\n", data[0], data[1]);

      temp_config_value = (((int16_t)data[0])<<8 | ((int16_t)data[1]));
      printf("Temperature configuration register value is: %d \n",
temp_config_value);
      return temp_config_value;

}

float read_temperature_data_register(int format){

      float temperature_value;
      uint8_t data[3]={0};

      /* Writing to the pointer register for reading temperature data
register */
      write_pointer_register(I2C_TEMP_SENSOR_TEMP_DATA_REG);

      /* Reading the temperature data register value */
      if (wrapper_read(file_descriptor, data, 2) != 2) {
            perror("Temperature data register wrapper_read error");
      }

      if(format == TEMP_CELSIUS){
            temperature_value = (data[0]<<4 | (data[1] >> 4 & 0XF)) *
0.0625;
            printf("Temperature value is: %3.2f degree Celsius \n",
temperature_value);
      }
      else if(format == TEMP_KELVIN){
            temperature_value = (data[0]<<4 | (data[1] >> 4 & 0XF)) *
0.0625;
            temperature_value += 273.15;
```

```
            printf("Temperature value is: %3.2f degree Kelvin \n",
temperature_value);
      }
      else if(format == TEMP_FARENHEIT){
            temperature_value = (data[0]<<4 | (data[1] >> 4 & 0XF)) *
0.0625;
            temperature_value = ((temperature_value * 9)/5 + 32);
            printf("Temperature value is: %3.2f degree Fahrenheit \n",
temperature_value);

      }
      else{
            printf("Invalid format\n");
      }
      return temperature_value;

}

int temp_sensor_init()
{
      if ((file_descriptor = open(I2C_SLAVE_DEV_NAME, O_RDWR)) < 0) {
        perror("Failed to open the bus.");
        /* ERROR HANDLING; you can check errno to see what went wrong */
            return -1;
      }

      if (ioctl(file_descriptor,I2C_SLAVE,I2C_SLAVE_ADDR) < 0) {
            perror("Failed to acquire bus access and/or talk to slave");
            /* ERROR HANDLING; you can check errno to see what went wrong
*/
            return -1;
      }

    if (temp_sensor_initialized == 0)
        temp_sensor_initialized = 1;

    return 0;
}

void *sensor_thread_func(void *arg)
{

      write_config_register_default();
      float temp_value;

    while (!g_sig_kill_sensor_thread)
    {
        temp_value = read_temperature_data_register(TEMP_CELSIUS);

        log_temp_data(temp_value);

        sleep(10);
    }

    pthread_exit(NULL);

    return NULL;
}

void log_temp_data(float temp_data)
```

```
{
    int msg_priority;

    /* Set the message queue attributes */
    struct mq_attr logger_mq_attr = { .mq_flags = 0,
                                      .mq_maxmsg =
MSG_QUEUE_MAX_NUM_MSGS,  // Max number of messages on queue
                                      .mq_msgsize =
MSG_QUEUE_MAX_MSG_SIZE  // Max. message size
    };

    logger_mq_handle = mq_open(MSG_QUEUE_NAME, O_RDWR, S_IRWXU,
&logger_mq_attr);

    char temp_data_msg[MSG_MAX_LEN];
    memset(temp_data_msg, '\0', sizeof(temp_data_msg));

    sprintf(temp_data_msg, "Temp Value: %3.2f", temp_data);

    struct _logger_msg_struct_ logger_msg;
    memset(&logger_msg, '\0', sizeof(logger_msg));
    strcpy(logger_msg.message, temp_data_msg);
    strncpy(logger_msg.logger_msg_src_id, "Temp", strlen("Temp"));
    logger_msg.logger_msg_src_id[strlen("Temp")] = '\0';
    strncpy(logger_msg.logger_msg_level, "Info", strlen("Info"));
    logger_msg.logger_msg_level[strlen("Info")] = '\0';

    msg_priority = 2;
    int num_sent_bytes = mq_send(logger_mq_handle, (char *)&logger_msg,
            sizeof(logger_msg), msg_priority);
    if (num_sent_bytes < 0)
        perror("mq_send failed");
}

void init_sock(int *sock_fd, struct sockaddr_in *server_addr_struct,
               int port_num, int listen_qsize)
{
    int serv_addr_len = sizeof(struct sockaddr_in);

    /* Create the socket */
    if ((*sock_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0)
    {
        perror("socket creation failed");
        pthread_exit(NULL); // Change these return values from
pthread_exit
    }

    int option = 1;
    if(setsockopt(*sock_fd, SOL_SOCKET, (SO_REUSEPORT | SO_REUSEADDR),
(void *)&option, sizeof(option)) < 0)
    {
        perror("setsockopt failed");
        pthread_exit(NULL);
    }

    server_addr_struct->sin_family = AF_INET;
    server_addr_struct->sin_addr.s_addr = INADDR_ANY;
    server_addr_struct->sin_port = htons(port_num);

    if (bind(*sock_fd, (struct sockaddr *)server_addr_struct,
```

```
                                                sizeof(struct
sockaddr_in))<0)
    {
        perror("bind failed");
        pthread_exit(NULL);
    }

    if (listen(*sock_fd, listen_qsize) < 0)
    {
        perror("listen failed");
        pthread_exit(NULL);
    }

}

void *socket_thread_func(void *arg)
{
    int server_fd;
    struct sockaddr_in server_address;
    int serv_addr_len = sizeof(server_address);

    init_sock(&server_fd, &server_address, SERVER_PORT_NUM,
SERVER_LISTEN_QUEUE_SIZE);

    int accept_conn_id;
    printf("Waiting for request...\n");
    if ((accept_conn_id = accept(server_fd, (struct sockaddr
*)&server_address,
                    (socklen_t*)&serv_addr_len)) < 0)
    {
        perror("accept failed");
        //pthread_exit(NULL);
    }

    char recv_buffer[MSG_BUFF_MAX_LEN];

    while (!g_sig_kill_sock_thread)
    {
        memset(recv_buffer, '\0', sizeof(recv_buffer));

        size_t num_read_bytes = read(accept_conn_id, &recv_buffer,
sizeof(recv_buffer));

        printf("[Temp_Task] Message req api: %s, req recp: %s, req api
params: %d\n",
                (((struct _socket_req_msg_struct_ *)&recv_buffer)-
>req_api_msg),
                ((((struct _socket_req_msg_struct_ *)&recv_buffer)-
>req_recipient)
                 == REQ_RECP_TEMP_TASK ? "Temp Task" : "Light Task"),
                (int)(((struct _socket_req_msg_struct_ *)&recv_buffer)-
>params));


        if (!strcmp((((struct _socket_req_msg_struct_ *)&recv_buffer)-
>req_api_msg), "get_temp_data"))
        {
            float temp_data =
read_temperature_data_register(TEMP_CELSIUS);
            char temp_data_msg[64];
```

```
            memset(temp_data_msg, '\0', sizeof(temp_data_msg));

            sprintf(temp_data_msg, "Temp Data: %3.2f", temp_data);

            ssize_t num_sent_bytes = send(accept_conn_id, temp_data_msg,
strlen(temp_data_msg), 0);
            if (num_sent_bytes < 0)
                perror("send failed");
        }
           else if (!strcmp((((struct _socket_req_msg_struct_
*)&recv_buffer)->req_api_msg), "get_temp_low_data"))
        {
            int16_t temp_data =
read_temp_high_low_register(I2C_TEMP_SENSOR_TLOW_REG);
            char temp_data_msg[64];
            memset(temp_data_msg, '\0', sizeof(temp_data_msg));

            sprintf(temp_data_msg, "Tlow Data: %d", temp_data);

            ssize_t num_sent_bytes = send(accept_conn_id, temp_data_msg,
strlen(temp_data_msg), 0);
            if (num_sent_bytes < 0)
                perror("send failed");
        }
           else if (!strcmp((((struct _socket_req_msg_struct_
*)&recv_buffer)->req_api_msg), "get_temp_high_data"))
        {
            int16_t temp_data =
read_temp_high_low_register(I2C_TEMP_SENSOR_THIGH_REG);
            char temp_data_msg[64];
            memset(temp_data_msg, '\0', sizeof(temp_data_msg));

            sprintf(temp_data_msg, "T_High Data: %d", temp_data);

            ssize_t num_sent_bytes = send(accept_conn_id, temp_data_msg,
strlen(temp_data_msg), 0);
            if (num_sent_bytes < 0)
                perror("send failed");
        }
           else if (!strcmp((((struct _socket_req_msg_struct_
*)&recv_buffer)->req_api_msg), "get_temp_em"))
        {
            uint8_t temp_data = read_config_register_em();
            char temp_data_msg[64];
            memset(temp_data_msg, '\0', sizeof(temp_data_msg));

            sprintf(temp_data_msg, "Temp EM data: %d", temp_data);

            ssize_t num_sent_bytes = send(accept_conn_id, temp_data_msg,
strlen(temp_data_msg), 0);
            if (num_sent_bytes < 0)
                perror("send failed");
        }
           else if (!strcmp((((struct _socket_req_msg_struct_
*)&recv_buffer)->req_api_msg), "get_temp_conversion_rate"))
        {
            uint8_t temp_data = read_config_register_conversion_rate();
            char temp_data_msg[64];
            memset(temp_data_msg, '\0', sizeof(temp_data_msg));
```

```
            sprintf(temp_data_msg, "T_High Data: %d", temp_data);

            ssize_t num_sent_bytes = send(accept_conn_id, temp_data_msg,
strlen(temp_data_msg), 0);
            if (num_sent_bytes < 0)
                perror("send failed");
        }
           else if (!strcmp((((struct _socket_req_msg_struct_
*)&recv_buffer)->req_api_msg), "get_temp_conf_data"))
        {
            uint16_t temp_data = read_temp_config_register();
            char temp_data_msg[64];
            memset(temp_data_msg, '\0', sizeof(temp_data_msg));

            sprintf(temp_data_msg, "Conf Data: %d", temp_data);

            ssize_t num_sent_bytes = send(accept_conn_id, temp_data_msg,
strlen(temp_data_msg), 0);
            if (num_sent_bytes < 0)
                perror("send failed");
        }
           else if (!strcmp((((struct _socket_req_msg_struct_
*)&recv_buffer)->req_api_msg), "set_temp_on_off"))
        {
            uint8_t data = (uint8_t)(((struct _socket_req_msg_struct_
*)&recv_buffer)->params);
            write_config_register_on_off(data);
            char temp_data_msg[64];
            memset(temp_data_msg, '\0', sizeof(temp_data_msg));

            sprintf(temp_data_msg, "%s", "Set success");

            ssize_t num_sent_bytes = send(accept_conn_id, temp_data_msg,
strlen(temp_data_msg), 0);
            if (num_sent_bytes < 0)
                perror("send failed");
        }
           else if (!strcmp((((struct _socket_req_msg_struct_
*)&recv_buffer)->req_api_msg), "set_temp_em"))
        {
            uint8_t data = (uint8_t)(((struct _socket_req_msg_struct_
*)&recv_buffer)->params);
            write_config_register_em(data);
            char temp_data_msg[64];
            memset(temp_data_msg, '\0', sizeof(temp_data_msg));

            sprintf(temp_data_msg, "%s", "Set success");

            ssize_t num_sent_bytes = send(accept_conn_id, temp_data_msg,
strlen(temp_data_msg), 0);
            if (num_sent_bytes < 0)
                perror("send failed");
        }
           else if (!strcmp((((struct _socket_req_msg_struct_
*)&recv_buffer)->req_api_msg), "set_temp_conversion_rate"))
        {
            uint8_t data = (uint8_t)(((struct _socket_req_msg_struct_
*)&recv_buffer)->params);
            write_config_register_conversion_rate(data);
            char temp_data_msg[64];
```

```
            memset(temp_data_msg, '\0', sizeof(temp_data_msg));

            sprintf(temp_data_msg, "%s", "Set success");

            ssize_t num_sent_bytes = send(accept_conn_id, temp_data_msg,
strlen(temp_data_msg), 0);
            if (num_sent_bytes < 0)
                perror("send failed");
        }
           else if (!strcmp((((struct _socket_req_msg_struct_
*)&recv_buffer)->req_api_msg), "set_temp_high_data"))
        {
            int16_t data = (int16_t)(((struct _socket_req_msg_struct_
*)&recv_buffer)->params);
            write_temp_high_low_register(I2C_TEMP_SENSOR_THIGH_REG,data);
            char temp_data_msg[64];
            memset(temp_data_msg, '\0', sizeof(temp_data_msg));

            sprintf(temp_data_msg, "%s", "Set success");

            ssize_t num_sent_bytes = send(accept_conn_id, temp_data_msg,
strlen(temp_data_msg), 0);
            if (num_sent_bytes < 0)
                perror("send failed");
        }
           else if (!strcmp((((struct _socket_req_msg_struct_
*)&recv_buffer)->req_api_msg), "set_temp_low_data"))
        {
            int16_t data = (int16_t)(((struct _socket_req_msg_struct_
*)&recv_buffer)->params);
            write_temp_high_low_register(I2C_TEMP_SENSOR_TLOW_REG,data);
            char temp_data_msg[64];
            memset(temp_data_msg, '\0', sizeof(temp_data_msg));

            sprintf(temp_data_msg, "%s", "Set success");

            ssize_t num_sent_bytes = send(accept_conn_id, temp_data_msg,
strlen(temp_data_msg), 0);
            if (num_sent_bytes < 0)
                perror("send failed");
        }
        else if (!strcmp((((struct _socket_req_msg_struct_
*)&recv_buffer)->req_api_msg), "set_temp_fault_bits"))
        {
            uint8_t data = (uint8_t)(((struct _socket_req_msg_struct_
*)&recv_buffer)->params);
            write_config_register_fault_bits(data);
            char temp_data_msg[64];
            memset(temp_data_msg, '\0', sizeof(temp_data_msg));

            sprintf(temp_data_msg, "%s", "Set success");

            ssize_t num_sent_bytes = send(accept_conn_id, temp_data_msg,
strlen(temp_data_msg), 0);
            if (num_sent_bytes < 0)
                perror("send failed");
        }
           else if (!strcmp((((struct _socket_req_msg_struct_
*)&recv_buffer)->req_api_msg), "get_temp_fault_bits"))
        {
```

```
            uint8_t temp_data = read_config_register_fault_bits();
            char temp_data_msg[64];
            memset(temp_data_msg, '\0', sizeof(temp_data_msg));

            printf("Fault Bits: %d", temp_data);
            sprintf(temp_data_msg, "Fault Bits: %d", temp_data);

            ssize_t num_sent_bytes = send(accept_conn_id, temp_data_msg,
strlen(temp_data_msg), 0);
            if (num_sent_bytes < 0)
                perror("send failed");
        }
    }

    printf("Calling pthread_exit in sock thread\n");
    pthread_exit(NULL);

    return NULL;
}

void *socket_hb_thread_func(void *arg)
{
    int sock_hb_fd;
    struct sockaddr_in sock_hb_address;
    int sock_hb_addr_len = sizeof(sock_hb_address);

    init_sock(&sock_hb_fd, &sock_hb_address, SOCKET_HB_PORT_NUM,
SOCKET_HB_LISTEN_QUEUE_SIZE);

    int accept_conn_id;
    printf("Waiting for request...\n");
    if ((accept_conn_id = accept(sock_hb_fd, (struct sockaddr
*)&sock_hb_address,
                    (socklen_t*)&sock_hb_addr_len)) < 0)
    {
        perror("accept failed");
        //pthread_exit(NULL);
    }

    char recv_buffer[MSG_BUFF_MAX_LEN];
    char send_buffer[20];
    memset(send_buffer, '\0', sizeof(send_buffer));

    while (!g_sig_kill_sock_hb_thread)
    {
        memset(recv_buffer, '\0', sizeof(recv_buffer));

        size_t num_read_bytes = read(accept_conn_id, &recv_buffer,
sizeof(recv_buffer));

        if (!strcmp(recv_buffer, "heartbeat"))
        {
            strcpy(send_buffer, "Alive");
                 ssize_t num_sent_bytes = send(accept_conn_id,
send_buffer, strlen(send_buffer), 0);
            if (num_sent_bytes < 0)
                perror("send failed");
        }
        else if (!strcmp(recv_buffer, "startup_check"))
        {
```

```
            /* For the sake of start-up check, because we have the
temperature sensor initialized
            ** by the time this thread is spawned. So we perform a
"get_temp_data" call to see if
            ** everything is working fine */
            if (temp_sensor_initialized == 1)
                strcpy(send_buffer, "Initialized");
            else
                strcpy(send_buffer, "Uninitialized");

                 ssize_t num_sent_bytes = send(accept_conn_id,
send_buffer, strlen(send_buffer), 0);
            if (num_sent_bytes < 0)
                perror("send failed");
        }
    }

    printf("Calling pthread_exit in sock hb thread\n");
    pthread_exit(NULL);

    return NULL;
}

void sig_handler(int sig_num)
{
      char buffer[MSG_BUFF_MAX_LEN];
      memset(buffer, '\0', sizeof(buffer));

      if (sig_num == SIGINT || sig_num == SIGUSR1)
      {
        if (sig_num == SIGINT)
            printf("Caught signal %s in temperature task\n", "SIGINT");
        else if (sig_num == SIGUSR1)
            printf("Caught signal %s in temperature task\n", "SIGKILL");

        g_sig_kill_sensor_thread = 1;
        g_sig_kill_sock_thread = 1;
        g_sig_kill_sock_hb_thread = 1;

        //pthread_join(sensor_thread_id, NULL);
        //pthread_join(socket_thread_id, NULL);
        //pthread_join(socket_hb_thread_id, NULL);

        mq_close(logger_mq_handle);

        close(file_descriptor);

        exit(0);
    }
}

int create_threads()
{
    int sens_t_creat_ret_val = pthread_create(&sensor_thread_id, NULL,
&sensor_thread_func, NULL);
    if (sens_t_creat_ret_val)
    {
        perror("Sensor thread creation failed");
        return -1;
    }
```

```
    int sock_t_creat_ret_val = pthread_create(&socket_thread_id, NULL,
&socket_thread_func, NULL);
    if (sock_t_creat_ret_val)
    {
        perror("Socket thread creation failed");
        return -1;
    }

    int sock_hb_t_creat_ret_val = pthread_create(&socket_hb_thread_id,
NULL, &socket_hb_thread_func, NULL);
    if (sock_hb_t_creat_ret_val)
    {
        perror("Socket heartbeat thread creation failed");
        return -1;
    }

    return 0;
}

int main()
{
    temp_sensor_initialized = 0;

    int temp_sensor_init_status = temp_sensor_init();
    if (temp_sensor_init_status == -1)
    {
        printf("Temperature sensor init failed\n");
        exit(1);
    }
    else
    {
        printf("Temperature sensor init success\n");
    }

    int thread_create_status = create_threads();
    if (thread_create_status)
    {
        printf("Thread creation failed\n");
    }
    else
    {
        printf("Thread creation success\n");
    }

    if (signal(SIGINT, sig_handler) == SIG_ERR)
        printf("SigHandler setup for SIGINT failed\n");

    if (signal(SIGUSR1, sig_handler) == SIG_ERR)
        printf("SigHandler setup for SIGKILL failed\n");

    g_sig_kill_sensor_thread = 0;
    g_sig_kill_sock_thread = 0;
    g_sig_kill_sock_hb_thread = 0;

    pthread_join(sensor_thread_id, NULL);
    pthread_join(socket_thread_id, NULL);
    pthread_join(socket_hb_thread_id, NULL);

    close(file_descriptor);
```

```
    return 0;
}
/*************************************************************************
******
* Author:       Pavan Dhareshwar & Sridhar Pavithrapu
* Date:         03/07/2018
* File:         wrapper.c
* Description:  Source file describing the functionality and
implementation
*               of wrapper for synchronization of light and temperature
tasks.
*************************************************************************
*****/

/*---------------------------------- INCLUDES ----------------------------
----*/
#include "wrapper.h"

sem_t *get_named_semaphore_handle(void)
{
    sem_t *sem;
    if ((sem = sem_open("wrapper_sem", O_CREAT, 0644, 1)) == SEM_FAILED)
    {
        perror("sem_open failed");
        return SEM_FAILED;
    }
    else
    {
        //printf("Named semaphore created successfully\n");
        return sem;
    }
}

ssize_t wrapper_write(int fd, void *buf, size_t count){

     ssize_t return_value = 0;

#if 1
    sem_t *wrapper_sem = get_named_semaphore_handle();
    if (wrapper_sem == SEM_FAILED)
    {
        return -1000;
    }

     if(sem_wait(wrapper_sem) == 0)
    {
            return_value = write(fd, buf, count);
    }
     else{
           perror("sem_wait error in wrapper\n");
     }

     if(sem_post(wrapper_sem) != 0){

           perror("sem_post error in wrapper\n");
     }
#else
    return_value = write(fd, buf, count);
#endif
```

```
      return return_value;
}

ssize_t wrapper_read(int fd, void *buf, size_t count){

      ssize_t return_value = 0;

#if 1
    sem_t *wrapper_sem = get_named_semaphore_handle();
    if (wrapper_sem == SEM_FAILED)
    {
        return -1000;
    }

    if(sem_wait(wrapper_sem) == 0){

             return_value = read(fd, buf, count);
    }
      else{
            perror("sem_wait error in wrapper\n");
      }

      if(sem_post(wrapper_sem) != 0){

            perror("sem_post error in wrapper\n");
      }
#else
    return_value = read(fd, buf, count);
#endif

      return return_value;
}
#ifndef _WRAPPER_H_
#define _WRAPPER_H_

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

sem_t *get_named_semaphore_handle(void);

ssize_t wrapper_write(int fd, void *buf, size_t count);
ssize_t wrapper_read(int fd, void *buf, size_t count);

#endif
```