ECEN5013 Project1 - Roberto Baquerizo

Generated by Doxygen 1.8.13

Contents

Chapter 1

Smart Environment Monitoring Device

Environment monitoring device making use of the BeagleBone Green development board and two offboard sensors: 1) Texas Instruments Temperature Sensor: TMP102

http://www.ti.com/lit/ds/symlink/tmp102.pdf 2) Broadcom Light Sensor: APDS-9301

https://www.broadcom.com/products/optical-sensors/ambient-light-photo-sensors/apds-9301

Overview:

In this project, we will implement a “smart” environment monitoring device using a BeagleBone Green (BBG) and two offboard sensors: a temperature sensor and a light sensor. The sensors will connect to the BBG using the same I2C bus via grove connectors. The application will periodically monitor the sensors and, in that period, log data to a single file on the system. The application will also be capable of detecting exceptional conditions which occur outside of the monitoring period and log the occurrence to the same log file. Exceptional conditions will also be reported by flashing one of the three LEDs of the BBG.

The application shall consist of three threads, in addition to a main (master) process, which will spawn the three children threads: one thread will interface with the temperature sensor; another, with the light sensor; and, a third to take care of any logging of the system, and sub-systems, states. The main thread will also be responsible for maintaining a watchdog task to check for the health of the three children.

Chapter 2

Data Structure Index

2.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 3

File Index

3.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 4

Data Structure Documentation

4.1 conv_res_t Struct Reference

Data Fields

- uint16_t **res_0**
- uint16_t **res_1**

The documentation for this struct was generated from the following file:

- /home/baquerrj/boulder/ecen5013/project_1/inc/temperature.h

4.2 file_t Struct Reference

Data Fields

- char * **name**
- FILE * **fid**

The documentation for this struct was generated from the following file:

- /home/baquerrj/boulder/ecen5013/project_1/inc/common.h

4.3 i2c_handle_t Struct Reference

Data Fields

- mraa_i2c_context **context**
- pthread_mutex_t **mutex**

The documentation for this struct was generated from the following file:

- /home/baquerrj/boulder/ecen5013/project_1/inc/i2c.h

4.4 msg_t Struct Reference

Collaboration diagram for msg_t:

Data Fields

- request_e **id**
- mqd_t **src**
- char **info** [GEN_BUFFER_SIZE]
- sensor_data_t **data**

The documentation for this struct was generated from the following file:

- /home/baquerrj/boulder/ecen5013/project_1/inc/common.h

4.5 sensor_data_t Struct Reference

Data Fields

- float **data**
- int night

4.5.1 Field Documentation

4.5.1.1 night

```
int sensor_data_t::night
```

Can be temperature in Celsius, Fahrenheit, or Kelvin OR lux output from light sensor

The documentation for this struct was generated from the following file:

- /home/baquerrj/boulder/ecen5013/project_1/inc/common.h

4.6 shared_data_t Struct Reference

Data Fields

- char **buffer** [SHM_BUFFER_SIZE]
- char header [SHM_BUFFER_SIZE]
- sem_t w_sem
- sem_t **r_sem**

4.6.1 Field Documentation

4.6.1.1 header

```
char shared_data_t::header[SHM_BUFFER_SIZE]
```

Buffer for message from thread

4.6.1.2 w_sem

```
sem_t shared_data_t::w_sem
```

Buffer for header identifying the thread who wrote to shm

The documentation for this struct was generated from the following file:

- /home/baquerrj/boulder/ecen5013/project_1/inc/common.h

4.7 thread_id_s Struct Reference

Data Fields

- pthread_t **temp_thread**
- pthread_t **light_thread**
- pthread_t **logger_thread**
- pthread_t **socket_thread**
- pthread_t **watchdog_thread**

The documentation for this struct was generated from the following file:

- /home/baquerrj/boulder/ecen5013/project_1/inc/common.h

4.8 tmp102_config_t Struct Reference

Collaboration diagram for tmp102_config_t:

Data Fields

- tmp102_mode_t **mode**
- uint16_t **polarity**
- uint16_t **fault_queue**
- conv_res_t **resolution**
- uint16_t **one_shot**
- uint16_t **operation**
- uint16_t **alert**
- uint16_t **conv_rate**

The documentation for this struct was generated from the following file:

- /home/baquerrj/boulder/ecen5013/project_1/inc/temperature.h

4.9 tmp102_mode_t Struct Reference

Data Fields

- uint16_t **shutdown**
- uint16_t **thermostat**

The documentation for this struct was generated from the following file:

- /home/baquerrj/boulder/ecen5013/project_1/inc/temperature.h

Chapter 5

File Documentation

5.1 /home/baquerrj/boulder/ecen5013/project_1/inc/common.h File Reference

Defines types and functions common between the threads of the application.

```
#include <signal.h>
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#include <semaphore.h>
#include <mqueue.h>
```

Include dependency graph for common.h:

5.2 /home/baquerrj/boulder/ecen5013/project_1/inc/i2c.h File Reference

Interface to I2C Bus of BeagleBone Green using libmraa https://iotdk.intel.com/docs/master/mraa/.

```
#include <pthread.h>
#include "mraa/i2c.h"
```

Include dependency graph for i2c.h: This graph shows which files directly or indirectly include this file:

Data Structures

- struct i2c_handle_t

Functions

- int **i2c_set** (int slave, int addr)
- int i2c_write_byte (int slave, int reg, uint8_t data)

  Writes byte to register address.
- int i2c_write (int slave, int reg, uint16_t data)

  Writes data to register address.
- int i2c_read (int slave, int reg, uint8_t *data, size_t len)

  Reads data from register address.
- int i2c_init (i2c_handle_t *i2c)

  Initialize singleton master i2c context.
- int i2c_stop (i2c_handle_t *i2c)

  Stops i2c instance.

5.2.1 Detailed Description

Interface to I2C Bus of BeagleBone Green using libmraa https://iotdk.intel.com/docs/master/mraa/.

==========

Author

Roberto Baquerizo (baquerrj), roba8460@colorado.edu

5.2.2 Function Documentation

5.2.2.1 i2c_init()

```
int i2c_init (
              i2c_handle_t * i2c )
```

Initialize singleton master i2c context.

========== Function↩
: i2c_init

Parameters

*i2c	- pointer to handle to be master **EXIT_CLEAN on success, otherwise EXIT_INIT**

```
00203 {
00204    if( NULL != my_i2c )
00205    {
00206       i2c = my_i2c;
00207       return EXIT_CLEAN;
00208    }
00209
00210    if( NULL != i2c )
00211    {
00212       i2c->context = mraa_i2c_init_raw( 2 );
00213
00214       if( NULL == i2c->context )
00215       {
00216          int errnum = errno;
00217          fprintf( stderr, "Failed to initialize I2C master instance: (%s)\n",
00218                   strerror( errnum ) );
00219          my_i2c = NULL;
00220          return EXIT_INIT;
00221       }
00222
00223       int retVal = pthread_mutex_init( &i2c->mutex, NULL );
00224       if( 0 > retVal )
00225       {
00226          int errnum = errno;
00227          fprintf( stderr, "Failed to initialize mutex for I2C master instance: (%s)\n",
00228                   strerror( errnum ) );
00229          my_i2c = NULL;
00230          retVal = mraa_i2c_stop( i2c->context );
00231          if( 0 > retVal )
00232          {
00233             mraa_result_print( retVal );
```

```
00234           }
00235           return EXIT_INIT;
00236       }
00237       my_i2c = i2c;
00238    }
00239    return EXIT_CLEAN;
00240 }
```

5.2.2.2 i2c_read()

```
int i2c_read (
              int slave,
              int reg,
              uint8_t * data,
              size_t len )
```

Reads data from register address.

========== Function↩
: i2c_read

Parameters

slave	- address of i2c slave
reg	- address to read from
∗*data*	- pointer to location to store read data
len	- size of memory to read in bytes **EXIT_CLEAN on success, otherwise one of exit_e**

```
00151 {
00152    if( NULL == my_i2c )
00153    {
00154       fprintf( stderr, "I2C master has not been initialized!\n" );
00155       return EXIT_INIT;
00156    }
00157
00158    pthread_mutex_lock( &my_i2c->mutex );
00159
00160    mraa_result_t retVal = mraa_i2c_address( my_i2c->context, slave );
00161    if( 0 != retVal )
00162    {
00163       mraa_result_print( retVal );
00164       pthread_mutex_unlock( &my_i2c->mutex );
00165       return EXIT_ERROR;
00166    }
00167
00168    if( len )
00169    {
00170       retVal = mraa_i2c_read_bytes_data( my_i2c->context, reg, data, len );
00171       pthread_mutex_unlock( &my_i2c->mutex );
00172       if( len != retVal )
00173       {
00174          fprintf( stderr, "Could not read all data from register!\n" );
00175          return EXIT_ERROR;
00176       }
00177    }
00178    else
00179    {
00180       /* only read one byte */
00181       retVal = mraa_i2c_read_byte_data( my_i2c->context, reg );
```

```
00182       pthread_mutex_unlock( &my_i2c->mutex );
00183       if( -1 != retVal )
00184       {
00185          *data  = retVal;
00186       }
00187    }
00188
00189    return EXIT_CLEAN;
00190 }
```

Here is the caller graph for this function:

5.2.2.3 i2c_stop()

```
int i2c_stop (
              i2c_handle_t * i2c )
```

Stops i2c instance.

========== Function↩
: i2c_stop

Parameters

*i2c	- pointer to i2c context handle **EXIT_CLEAN on success, otherwise EXIT_ERROR**

```
00253 {
00254    if( NULL == my_i2c )
00255    {
00256       return EXIT_CLEAN;
00257    }
00258    else if( NULL == i2c )
00259    {
00260       return EXIT_CLEAN;
00261    }
00262
00263    if( my_i2c != i2c )
00264    {
00265       return EXIT_ERROR;
00266    }
00267
00268    while( EBUSY  == pthread_mutex_destroy( &i2c->mutex ) );
00269
00270    mraa_result_t retVal = mraa_i2c_stop( i2c->context );
00271    if( 0 > retVal )
00272    {
00273       mraa_result_print( retVal );
00274       return EXIT_ERROR;
00275    }
00276
00277    my_i2c = NULL;
00278    return EXIT_CLEAN;
00279 }
```

5.2.2.4 i2c_write()

```
int i2c_write (
              int slave,
```

```
            int reg,
            uint16_t data )
```

Writes data to register address.

========== Function↩
: i2c_write

Parameters

slave	- address of i2c slave
reg	- address of register to write to
data	- data to write **EXIT_CLEAN on success, otherwise one of exit_e**

```
00114 {
00115    if( NULL == my_i2c )
00116    {
00117       fprintf( stderr, "I2C master has not been initialized!\n" );
00118       return EXIT_INIT;
00119    }
00120
00121    /* take hardware mutex */
00122    pthread_mutex_lock( &my_i2c->mutex );
00123
00124    mraa_result_t retVal = mraa_i2c_address( my_i2c->context, slave );
00125    if( 0 != retVal )
00126    {
00127       mraa_result_print( retVal );
00128       pthread_mutex_unlock( &my_i2c->mutex );
00129       return EXIT_ERROR;
00130    }
00131
00132    retVal = mraa_i2c_write_word_data( my_i2c->context, data, reg );
00133    pthread_mutex_unlock( &my_i2c->mutex );
00134
00135    return EXIT_CLEAN;
00136 }
```

Here is the caller graph for this function:

5.2.2.5 i2c_write_byte()

```
int i2c_write_byte (
            int slave,
            int reg,
            uint8_t data )
```

Writes byte to register address.

========== Function↩
: i2c_write_byte

Parameters

slave	- address of i2c slave
reg	- address of register to write to
data	- data to write
	EXIT_CLEAN on success, otherwise one of exit_e

```
00078 {
00079    if( NULL == my_i2c )
00080    {
00081       fprintf( stderr, "I2C master has not been initialized!\n" );
00082       return EXIT_INIT;
00083    }
00084
00085    /* take hardware mutex */
00086    pthread_mutex_lock( &my_i2c->mutex );
00087
00088    mraa_result_t retVal = mraa_i2c_address( my_i2c->context, slave );
00089    if( 0 != retVal )
00090    {
00091       mraa_result_print( retVal );
00092       pthread_mutex_unlock( &my_i2c->mutex );
00093       return EXIT_ERROR;
00094    }
00095
00096    retVal = mraa_i2c_write_byte_data( my_i2c->context, data, reg );
00097    pthread_mutex_unlock( &my_i2c->mutex );
00098
00099    return EXIT_CLEAN;
00100 }
```

Here is the caller graph for this function:

5.3 /home/baquerrj/boulder/ecen5013/project_1/inc/led.h File Reference

Interface to USR LEDs of BeagleBone Green http://derekmolloy.ie/beaglebone-controlling-the-on-board-

`#include <stdio.h>`
Include dependency graph for led.h: This graph shows which files directly or indirectly include this file:

Macros

- #define **LED0_PATH** "/sys/class/leds/beaglebone:green:heartbeat"
- #define **LED1_PATH** "/sys/class/leds/beaglebone:green:mmc0"
- #define **LED2_PATH** "/sys/class/leds/beaglebone:green:usr2"
- #define **LED3_PATH** "/sys/class/leds/beaglebone:green:usr3"
- #define **LED_BRIGHTNESS**(LED_PATH) (LED_PATH"/brightness")
- #define **LED_TRIGGER**(LED_PATH) (LED_PATH"/trigger")
- #define **LED_DELAYON**(LED_PATH) (LED_PATH"/delay_on")
- #define **LED_DELAYOFF**(LED_PATH) (LED_PATH"/delay_off")
- #define **LED0_BRIGHTNESS** LED_BRIGHTNESS(LED0_PATH)
- #define **LED1_BRIGHTNESS** LED_BRIGHTNESS(LED1_PATH)
- #define **LED2_BRIGHTNESS** LED_BRIGHTNESS(LED2_PATH)
- #define **LED3_BRIGHTNESS** LED_BRIGHTNESS(LED3_PATH)
- #define **LED0_TRIGGER** LED_TRIGGER(LED0_PATH)
- #define **LED1_TRIGGER** LED_TRIGGER(LED1_PATH)
- #define **LED2_TRIGGER** LED_TRIGGER(LED2_PATH)
- #define **LED3_TRIGGER** LED_TRIGGER(LED3_PATH)
- #define **LED0_DELAYON** LED_DELAYON(LED0_PATH)
- #define **LED1_DELAYON** LED_DELAYON(LED1_PATH)
- #define **LED2_DELAYON** LED_DELAYON(LED2_PATH)
- #define **LED3_DELAYON** LED_DELAYON(LED3_PATH)
- #define **LED0_DELAYOFF** LED_DELAYOFF(LED0_PATH)
- #define **LED1_DELAYOFF** LED_DELAYOFF(LED1_PATH)
- #define **LED2_DELAYOFF** LED_DELAYOFF(LED2_PATH)
- #define **LED3_DELAYOFF** LED_DELAYOFF(LED3_PATH)

Functions

- void get_status (const char *led)
- int set_trigger (const char *led, char *trigger)
- int set_delay (const char *led_path, int delay)
- void led_on (const char *led)
- void led_off (const char *led)
- void led_toggle (const char *led)

5.3.1 Detailed Description

Interface to USR LEDs of BeagleBone Green http://derekmolloy.ie/beaglebone-controlling-the-on-board-

==========

Define macros for interacting with user LEDs of BeagleBone Green.

Author

Roberto Baquerizo (baquerrj), roba8460@colorado.edu

5.3.2 Function Documentation

5.3.2.1 get_status()

```
void get_status (
            const char * led )
```

========== Function↩
: get_status

Parameters

<+NAME+>	<+DESCRIPTION+>

Returns

<+DESCRIPTION+>

<+DETAILED+>

```
00041 {
00042     return;
00043 }
```

5.3.2.2 led_off()

```
void led_off (
            const char * led )
```

========== Function↩
: led_off

Parameters

<+NAME+>	<+DESCRIPTION+>

Returns

<+DESCRIPTION+>

<+DETAILED+>

```
00126 {
00127    FILE *fp;
00128    fp = fopen( led, "w+" );
00129    fprintf( fp, "0" );
00130    fclose( fp );
00131    return;
00132 }
```

Here is the caller graph for this function:

5.3.2.3 led_on()

```
void led_on (
            const char * led )
```

========== Function↩
: led_on

Parameters

<+NAME+>	<+DESCRIPTION+>

Returns

<+DESCRIPTION+>

<+DETAILED+>

```
00106 {
00107    FILE *fp;
00108    fp = fopen( led, "w+" );
00109    fprintf( fp, "1" );
00110    fclose( fp );
00111    return;
00112 }
```

Here is the caller graph for this function:

5.3.2.4 led_toggle()

```
void led_toggle (
            const char * led )
```

========== Function↩
: led_toggle

Parameters

<+NAME+>	<+DESCRIPTION+>

Returns

<+DESCRIPTION+>

<+DETAILED+>

```
00145 {
00146    FILE *fp;
00147    fp = fopen( led, "rt" );
00148    fseek( fp, 0, SEEK_END );
00149    long size = ftell( fp );
00150    rewind( fp );
00151
00152    char *value = (char*) malloc( sizeof(char) * size );
00153    fread( value, 1, size, fp );
00154    fclose( fp );
00155    switch( *value )
00156    {
00157       case '0':
00158          led_on( led );
00159          break;
00160       case '1':
00161          led_off( led );
00162          break;
00163       default:
00164          break;
00165    }
00166    return;
00167 }
```

Here is the caller graph for this function:

5.3.2.5 set_delay()

```
int set_delay (
            const char * led,
            int delay )
```

========== Function↩
: set_delay

Parameters

<+NAME+>	<+DESCRIPTION+>

Returns

<+DESCRIPTION+>

<+DETAILED+>

```
00081 {
00082    FILE *fp = fopen( led, "w+" );
00083    if( NULL == fp )
00084    {
00085       int errnum = errno;
00086       fprintf( stderr, "Encuntered error trying to set delay for %s (%s)\nAre you sure LED is in correct
      configuration?\n",
00087                led, strerror ( errnum ) );
00088       return -1;
00089    }
00090    fprintf( fp, "%u", delay );
00091    fclose( fp );
00092    return delay;
00093 }
```

5.3.2.6 set_trigger()

```
int set_trigger (
             const char * led,
             char * trigger )
```

========== Function↩
: get_trigger

Parameters

<+NAME+>	<+DESCRIPTION+>

Returns

<+DESCRIPTION+>

<+DETAILED+>

```
00056 {
00057    FILE *fp = fopen( led, "w+" );
00058    if( NULL == fp )
00059    {
00060       int errnum = errno;
00061       fprintf( stderr, "Encountered error trying to set trigger %s for %s (%s)\n",
00062                trigger, led, strerror ( errnum ) );
00063       return -1;
00064    }
00065    fprintf( fp, "%s", trigger );
00066    fclose( fp );
00067    return 0;
00068 }
```

5.4 /home/baquerrj/boulder/ecen5013/project_1/inc/light.h File Reference

Interface to APDS9301 Light Sensor.

```
#include "common.h"
#include "i2c.h"
```

Include dependency graph for light.h: This graph shows which files directly or indirectly include this file:

Macros

- #define **LIGHT_QUEUE_NAME** "/light-queue"
- #define **APDS9301_ADDRESS** (0x39)
- #define APDS9301_REG_CMD (0x80)
- #define **APDS9301_REG_CNTRL** (0x80)
- #define **APDS9301_REG_TIME** (0x81)
- #define **APDS9301_REG_TH_LL** (0x82)
- #define **APDS9301_REG_TH_LH** (0x83)
- #define **APDS9301_REG_TH_HL** (0x84)
- #define **APDS9301_REG_TH_HH** (0x85)
- #define **APDS9301_REG_INT_CNTRL** (0x86)
- #define **APDS9301_REG_ID** (0x8A)
- #define **APDS9301_REG_DLOW_0** (0x8C)
- #define **APDS9301_REG_DHIGH_0** (0x8D)
- #define **APDS9301_REG_DLOW_1** (0x8E)
- #define **APDS9301_REG_DHIGH_1** (0x8F)
- #define POWER_ON (0x03)
- #define **POWER_OFF** (0x00)
- #define **CMD_CLEAR_INTR** (1<<5)
- #define **CMD_WORD_ENBL** (1<<6)
- #define DEFAULT_GAIN (0x00) /** low gain */
- #define **DEFAULT_INTEGRATION_TIME** (0x02) /** 402ms integration time */
- #define **DEFAULT_INTERRUPT** (0x00) /** No interrupts */
- #define **DARK_THRESHOLD** (50)

Functions

- float get_lux (void)

  Returns last lux reading.
- int is_dark (void)

  Returns int speciyfing if it is night or day.
- int apds9301_set_config (void)

  Set configuration of light sensor. For the APDS9301, the configuration is spread out across the: Timing Register, Interrupt Control Register, and Control Register. So, I have to write to all of these to set the config.
- int apds9301_set_integration (uint8_t val)

  Sets the integration time for APDS9301 by writing a value to bits INTEG of the Timing Register.
- int apds9301_clear_interrupt (void)

  Clears any pending interrupt for APDS9301 by writing a 1 to the CLEAR bit of the Command Register.
- int apds9301_set_interrupt (uint8_t enable)

  Enables or disables interrupts for APDS9301 by setting or clearing the INTR bits of the Interrupt Control Register.
- int apds9301_set_gain (uint8_t gain)

  Sets gain for APDS9301 by setting or clearing the GAIN bit of the Timing Register.
- int apds9301_read_control (uint8_t *data)

  Read contents of Control Register.
- int apds9301_write_threshold_low (uint16_t threshold)

  Write value to low threshold register.
- int apds9301_read_threshold_low (uint16_t *threshold)

  Read value from low threshold register.
- int apds9301_write_threshold_high (uint16_t threshold)

  Write value to high threshold register.
- int apds9301_read_threshold_high (uint16_t *threshold)

Read value from high threshold register.

- int apds9301_read_id (uint8_t *id)

  Read APDS9301 Identification Register.

- int apds9301_get_lux (float *lux)

  Read ADC Registers and calculate lux in lumen.

- int apds9301_read_data0 (uint16_t *data)

  Read ADC register for channel 0.

- int apds9301_read_data1 (uint16_t *data)

  Read ADC register for channel 1.

- int apds9301_power (uint16_t on)

  power on (or off) APDS9301 as set by paramater

- mqd_t get_light_queue (void)

  Get file descriptor for light sensor thread. Called by watchdog thread in order to be able to send heartbeat check via queue.

- int light_queue_init (void)

  Initialize message queue for light sensor thread.

- void * light_fn (void *thread_args)

  Entry point for light sensor processing thread.

5.4.1 Detailed Description

Interface to APDS9301 Light Sensor.

==========

Author

Roberto Baquerizo (baquerrj), roba8460@colorado.edu

5.4.2 Macro Definition Documentation

5.4.2.1 APDS9301_REG_CMD

```
#define APDS9301_REG_CMD (0x80)
```

Register adddresses

5.4.2.2 DEFAULT_GAIN

```
#define DEFAULT_GAIN (0x00) /** low gain */
```

Defaults

5.4.2.3 POWER_ON

```
#define POWER_ON (0x03)
```

Helpful constants

5.4.3 Function Documentation

5.4.3.1 apds9301_clear_interrupt()

```
int apds9301_clear_interrupt (
            void  )
```

Clears any pending interrupt for APDS9301 by writing a 1 to the CLEAR bit of the Command Register.

========== Function↩
: apds9301_clear_interrupt

Parameters

void	**see i2c_set()**

```
00236 {
00237    uint8_t clear = APDS9301_REG_CMD | CMD_CLEAR_INTR;
00238
00239    int retVal = i2c_set( APDS9301_ADDRESS, clear );
00240
00241    return retVal;
00242 }
```

5.4.3.2 apds9301_get_lux()

```
int apds9301_get_lux (
            float * lux )
```

Read ADC Registers and calculate lux in lumen.

========== Function↩
: apds9301_get_lux

Parameters

**lux*	- pointer to location to write decoded lux to **EXIT_CLEAN if successful, otherwise EXIT_ERROR**

Read ADC Registers and calculate lux in lumen.

========== Function↩
: apds9301_get_lux

Parameters

**lux*	- pointer to location to write decoded lux to **EXIT_CLEAN if successful, otherwise EXIT_ERROR**

```
00413 {
00414    float ratio = 0;
00415    uint16_t data0 = 0;
00416    uint16_t data1 = 0;
00417
00418    int retVal = apds9301_read_data0( &data0 );
00419    if( EXIT_CLEAN != retVal )
00420    {
00421       return EXIT_ERROR;
00422    }
00423
00424    retVal = apds9301_read_data1( &data1 );
00425    if( EXIT_CLEAN != retVal )
00426    {
00427       return EXIT_ERROR;
00428    }
00429
00430    if( 0 == data0 )
00431    {
00432       ratio = 0.0;
00433    }
00434    else
00435    {
00436       ratio = (float)data1 / (float)data0;
00437    }
00438
00439    if( (0 < ratio) && (0.50 >= ratio) )
00440    {
00441       *lux = 0.0304*data0 - 0.062*data0*(pow(ratio, 1.4));
00442     }
00443     else if( (0.50 < ratio) && (0.61 >= ratio) )
00444     {
00445         *lux = 0.0224*data0 - 0.031*data1;
00446     }
00447     else if( (0.61 < ratio) && (0.80 >= ratio) )
00448     {
00449         *lux = 0.0128*data0 - 0.0153*data1;
00450     }
00451     else if( (0.80 < ratio) && (1.30 >= ratio) )
00452     {
00453         *lux = 0.00146*data0 - 0.00112*data1;
00454     }
00455     else if( 1.30 < ratio )
00456     {
00457         *lux = 0;
00458    }
00459
00460    return EXIT_CLEAN;
00461 }
```

5.4.3.3 apds9301_power()

```
int apds9301_power (
            uint16_t on )
```

power on (or off) APDS9301 as set by paramater

========== Function↩
: apds9301_power

Parameters

on	- specifies if sensor is to be powered on or off see i2c_write_byte()

```
00538 {
00539    int retVal = 0;
00540    if( on )
00541    {
00542       /* power on */
00543       retVal = i2c_write_byte( APDS9301_ADDRESS, APDS9301_REG_CNTRL,
      POWER_ON );
00544    }
00545    else
00546    {
00547       /* power off */
00548       retVal = i2c_write_byte( APDS9301_ADDRESS, APDS9301_REG_CNTRL, POWER_OFF );
00549    }
00550
00551    return retVal;
00552 }
```

5.4.3.4 apds9301_read_control()

```
int apds9301_read_control (
            uint8_t * data )
```

Read contents of Control Register.

========== Function↩
: apds9301_read_control

Parameters

**data*	- where to store contents see i2c_read()

```
00322 {
00323    int retVal = i2c_read( APDS9301_ADDRESS, APDS9301_REG_CNTRL, data, sizeof( *data ) );
00324    return retVal;
00325 }
```

5.4.3.5 apds9301_read_data0()

```
int apds9301_read_data0 (
            uint16_t * data )
```

Read ADC register for channel 0.

========== function↩
: apds9301_read_data0

Parameters

*data	- pointer to location to write decoded value to **EXIT_CLEAN if successful, otherwise exit_error**

```
00473 {
00474    uint8_t low  = 0;
00475    uint8_t high = 0;
00476    int retVal = i2c_read( APDS9301_ADDRESS, APDS9301_REG_DLOW_0, &low, 0 );
00477
00478    if( EXIT_CLEAN != retVal )
00479    {
00480      return EXIT_ERROR;
00481    }
00482
00483    retVal = i2c_read( APDS9301_ADDRESS, APDS9301_REG_DHIGH_0, &high, 0 );
00484
00485    if( EXIT_CLEAN == retVal )
00486    {
00487       *data = ( low | (high << 8 ) );
00488    }
00489    else
00490    {
00491       return EXIT_ERROR;
00492    }
00493    return EXIT_CLEAN;
00494 }
```

Here is the caller graph for this function:

5.4.3.6 apds9301_read_data1()

```
int apds9301_read_data1 (
            uint16_t * data )
```

Read ADC register for channel 1.

========== function↩
: apds9301_read_data1

Parameters

∗data	- pointer to location to write decoded value to **EXIT_CLEAN if successful, otherwise exit_error**

```
00505 {
00506    uint8_t low  = 0;
00507    uint8_t high = 0;
00508    int retVal = i2c_read( APDS9301_ADDRESS, APDS9301_REG_DLOW_1, &low, 0 );
00509
00510    if( EXIT_CLEAN != retVal )
00511    {
00512      return EXIT_ERROR;
00513    }
00514
00515    retVal = i2c_read( APDS9301_ADDRESS, APDS9301_REG_DHIGH_1, &high, 0 );
00516
00517    if( EXIT_CLEAN == retVal )
00518    {
00519       *data = ( low | (high << 8 ) );
00520    }
00521    else
00522    {
00523       return EXIT_ERROR;
00524    }
00525    return EXIT_CLEAN;
00526 }
```

5.4.3.7 apds9301_read_id()

```
int apds9301_read_id (
              uint8_t * id )
```

Read APDS9301 Identification Register.

========== Function↩
: apds9301_read_id

Parameters

∗id	- where to write ID from register **EXIT_CLEAN if successful, EXIT_ERROR otherwise**

========== Function↩
: apds9301_read_id

Parameters

∗id	- where to write ID from register **see i2c_read()**

```
00397 {
00398     int retVal = i2c_read( APDS9301_ADDRESS, APDS9301_REG_ID, id, sizeof( *id ) );
00399     return retVal;
00400 }
```

5.4.3.8 apds9301_read_threshold_high()

```
int apds9301_read_threshold_high (
            uint16_t * threshold )
```

Read value from high threshold register.

========== Function↩
: apds9301_write_threshold_high

Parameters

**threshold*	- where to write value read **see i2c_write()**

```
00382 {
00383     int retVal = i2c_read( APDS9301_ADDRESS, APDS9301_REG_TH_HL, (uint8_t*)threshold, sizeof( *
      threshold ) );
00384     return retVal;
00385 }
```

5.4.3.9 apds9301_read_threshold_low()

```
int apds9301_read_threshold_low (
            uint16_t * threshold )
```

Read value from low threshold register.

========== Function↩
: apds9301_write_threshold_low

Parameters

**threshold*	- where to write value read **see i2c_write()**

```
00352 {
00353     int retVal = i2c_read( APDS9301_ADDRESS, APDS9301_REG_TH_LL, (uint8_t*)threshold, sizeof( *
```

```
      threshold ) );
00354    return retVal;
00355 }
```

5.4.3.10 apds9301_set_config()

```
int apds9301_set_config (
              void  )
```

Set configuration of light sensor. For the APDS9301, the configuration is spread out across the: Timing Register, Interrupt Control Register, and Control Register. So, I have to write to all of these to set the config.

========== Function↩
: apds9301_set_config

Parameters

void	**EXIT_CLEAN if successful, otherwise see i2c_write()**

```
00166 {
00167    int retVal = apds9301_set_gain( DEFAULT_GAIN );
00168    if( retVal )
00169    {
00170       return retVal;
00171    }
00172    else
00173    {
00174       retVal = apds9301_set_interrupt( DEFAULT_INTERRUPT );
00175       if( retVal )
00176       {
00177          return retVal;
00178       }
00179       else
00180       {
00181          retVal = apds9301_set_integration( DEFAULT_INTEGRATION_TIME );
00182          if( retVal )
00183          {
00184             return retVal;
00185          }
00186       }
00187    }
00188    return EXIT_CLEAN;
00189 }
```

5.4.3.11 apds9301_set_gain()

```
int apds9301_set_gain (
              uint8_t gain )
```

Sets gain for APDS9301 by setting or clearing the GAIN bit of the Timing Register.

========== Function↩
: apds9301_set_gain

Parameters

gain	- set if we want high gain see i2c_write_byte()

```
00289 {
00290    uint8_t data;
00291    int retVal = i2c_read( APDS9301_ADDRESS, APDS9301_REG_TIME, &data, sizeof( data ) );
00292    if( retVal )
00293    {
00294       return EXIT_ERROR;
00295    }
00296
00297    /* if gain != 0, high gain */
00298    if( gain )
00299    {
00300       data |= (1<<4);
00301    }
00302    else
00303    {
00304       data &= ~(1<<4);
00305    }
00306
00307    retVal = i2c_write_byte( APDS9301_ADDRESS, APDS9301_REG_TIME, data );
00308
00309    return retVal;
00310 }
```

Here is the caller graph for this function:

5.4.3.12 apds9301_set_integration()

```
int apds9301_set_integration (
             uint8_t val )
```

Sets the integration time for APDS9301 by writing a value to bits INTEG of the Timing Register.

========== Function↩
: apds9301_set_integration

Parameters

val	- value to write to timing register see i2c_write_byte() - if val is not an allowed value, EXIT_ERROR

```
00202 {
00203    if( 3 < val )
00204    {
00205       /* invalid value */
00206       return EXIT_ERROR;
00207    }
00208    uint8_t data;
00209    int retVal = i2c_read( APDS9301_ADDRESS, APDS9301_REG_TIME, &data, sizeof( data ) );
00210
00211    if( retVal )
00212    {
00213       return EXIT_ERROR;
00214    }
00215
```

```
00216     data &= ~(0b11);  /* clears lower 2 bits of TIMING REG */
00217     data |= val;
00218
00219     retVal = i2c_write_byte( APDS9301_ADDRESS, APDS9301_REG_TIME, data );
00220
00221     return retVal;
00222 }
```

5.4.3.13 apds9301_set_interrupt()

```
int apds9301_set_interrupt (
              uint8_t enable )
```

Enables or disables interrupts for APDS9301 by setting or clearing the INTR bits of the Interrupt Control Register.

========== Function↩
: apds9301_set_interrupt

Parameters

enable	- set if we want to enable interrupts see i2c_write_byte()

```
00256 {
00257     uint8_t data;
00258     int retVal = i2c_read( APDS9301_ADDRESS, APDS9301_REG_INT_CNTRL, &data, sizeof( data ) );
00259     if( retVal )
00260     {
00261        return EXIT_ERROR;
00262     }
00263
00264     if( enable )
00265     {
00266        data |= (1<<4);
00267     }
00268     else
00269     {
00270        data &= ~(1<<4);
00271     }
00272
00273     retVal = i2c_write_byte( APDS9301_ADDRESS, APDS9301_REG_INT_CNTRL, data );
00274
00275     return retVal;
00276 }
```

Here is the caller graph for this function:

5.4.3.14 apds9301_write_threshold_high()

```
int apds9301_write_threshold_high (
              uint16_t threshold )
```

Write value to high threshold register.

========== Function↩
: apds9301_write_threshold_high

Parameters

threshold	- value to write see i2c_write()

```
00367 {
00368     int retVal = i2c_write( APDS9301_ADDRESS, APDS9301_REG_TH_HL, threshold );
00369     return retVal;
00370 }
```

5.4.3.15 apds9301_write_threshold_low()

```
int apds9301_write_threshold_low (
              uint16_t threshold )
```

Write value to low threshold register.

========== Function↩
: apds9301_write_threshold_low

Parameters

threshold	- value to write see i2c_write()

```
00337 {
00338     int retVal = i2c_write( APDS9301_ADDRESS, APDS9301_REG_TH_LL, threshold );
00339     return retVal;
00340 }
```

5.4.3.16 get_light_queue()

```
mqd_t get_light_queue (
              void  )
```

Get file descriptor for light sensor thread. Called by watchdog thread in order to be able to send heartbeat check via queue.

========== Function↩
: get_light_queue

Parameters

void	**temp_queue - file descriptor for light sensor thread message queue**

```
00615 {
00616     return light_queue;
00617 }
```

5.4.3.17 get_lux()

```
float get_lux (
            void  )
```

Returns last lux reading.

========== Function↩
: get_lux

Parameters

void	**last_lux_value - last lux reading we have**

```
00130 {
00131     return last_lux_value;
00132 }
```

5.4.3.18 is_dark()

```
int is_dark (
            void  )
```

Returns int speciyfing if it is night or day.

========== Function↩
: is_dark

Parameters

void	**night - 0 if it is day, 1 if night, i.e. below DARK_THRESHOLD**

```
00145 {
00146    int dark = 0;
00147    if( DARK_THRESHOLD > last_lux_value )
00148    {
00149       dark = 1;
00150    }
00151    return dark;
00152 }
```

5.4.3.19 light_fn()

```
void* light_fn (
             void * thread_args )
```

Entry point for light sensor processing thread.

========== Function↩
: light_fn

Parameters

thread_args	- void ptr to arguments used to initialize thread

Returns

NULL - We don't really exit from this function,

since the exit point is thread_exit()

```
00663 {
00664    /* Get time that thread was spawned */
00665    struct timespec time;
00666    clock_gettime(CLOCK_REALTIME, &time);
00667    shm = get_shared_memory();
00668
00669    /* Write initial state to shared memory */
00670    sem_wait(&shm->w_sem);
00671    print_header(shm->header);
00672    sprintf( shm->buffer, "Hello World! Start Time: %ld.%ld secs\n",
00673             time.tv_sec, time.tv_nsec );
00674    /* Signal to logger that shared memory has been updated */
00675    sem_post(&shm->r_sem);
00676
00677    signal(SIGUSR1, sig_handler);
00678    signal(SIGUSR2, sig_handler);
00679
00680    light_queue = light_queue_init();
00681    if( 0 > light_queue )
00682    {
00683       thread_exit( EXIT_INIT );
00684    }
00685
00686    int retVal = i2c_init( &i2c_apds9301 );
00687    if( EXIT_INIT == retVal )
00688    {
00689       sem_wait(&shm->w_sem);
00690       print_header(shm->header);
00691       sprintf( shm->buffer, "ERROR: Failed to initialize I2C for light sensor!\n" );
00692       sem_post(&shm->r_sem);
00693       thread_exit( EXIT_INIT );
00694    }
00695    retVal = apds9301_power( POWER_ON );
00696    if( retVal )
00697    {
00698       sem_wait(&shm->w_sem);
00699       print_header(shm->header);
```

```
00700       sprintf( shm->buffer, "ERROR: Failed to power on light sensor!\n" );
00701       sem_post(&shm->r_sem);
00702       thread_exit( EXIT_INIT );
00703    }
00704
00705    timer_setup( &timerid, &timer_handler );
00706
00707    timer_start( &timerid, 5000000 );
00708    cycle();
00709
00710    thread_exit( 0 );
00711    return NULL;
00712 }
```

5.4.3.20 light_queue_init()

```
int light_queue_init (
             void  )
```

Initialize message queue for light sensor thread.

========== Function↩
: light_queue_init

Parameters

void	**msg_q - file descriptor for initialized message queue**

```
00629 {
00630    /* unlink first in case we hadn't shut down cleanly last time */
00631    mq_unlink( LIGHT_QUEUE_NAME );
00632
00633    struct mq_attr attr;
00634    attr.mq_flags = 0;
00635    attr.mq_maxmsg = MAX_MESSAGES;
00636    attr.mq_msgsize = sizeof( msg_t );
00637    attr.mq_curmsgs = 0;
00638
00639    int msg_q = mq_open( LIGHT_QUEUE_NAME, O_CREAT | O_RDWR, 0666, &attr );
00640    if( 0 > msg_q )
00641    {
00642       int errnum = errno;
00643       sem_wait(&shm->w_sem);
00644       print_header(shm->header);
00645       sprintf( shm->buffer, "Encountered error creating message queue %s: (%s)\n",
00646                LIGHT_QUEUE_NAME, strerror( errnum ) );
00647       sem_post(&shm->r_sem);
00648    }
00649    return msg_q;
00650 }
```

5.5 /home/baquerrj/boulder/ecen5013/project_1/inc/logger.h File Reference

<+DETAILED+>

```
#include "common.h"
```

Include dependency graph for logger.h: This graph shows which files directly or indirectly include this file:

Functions

- void * logger_fn (void *thread_args)

  Entry point for logger thread.

5.5.1 Detailed Description

<+DETAILED+>

==========

Author

Roberto Baquerizo (baquerrj), roba8460@colorado.edu

5.5.2 Function Documentation

5.5.2.1 logger_fn()

```
void* logger_fn (
            void * arg )
```

Entry point for logger thread.

========== Function↩
: logger_fn

Parameters

thread_args	- void ptr to arguments used to initialize thread

Returns

NULL - We don't really exit from this function,

since the exit point is thread_exit()

```
00079 {
00080    struct timespec time;
00081    clock_gettime(CLOCK_REALTIME, &time);
00082    static int failure = 1;
00083
00084    signal(SIGUSR1, sig_handler);
00085    signal(SIGUSR2, sig_handler);
00086
00087    /* Initialize thread */
00088    if( NULL == arg )
00089    {
00090       fprintf( stderr, "Thread requires name of log file!\n" );
00091       pthread_exit(&failure);
00092    }
00093
```

```
00094    log = (FILE *)arg;
00095    if( NULL == log )
00096    {
00097       perror( "Encountered error opening log file" );
00098       pthread_exit(&failure);
00099    }
00100
00101    shm = get_shared_memory();
00102    if( NULL == shm )
00103    {
00104       int errnum = errno;
00105       fprintf( stderr, "Encountered error memory mapping shared memory: %s\n",
00106                strerror( errnum ) );
00107    }
00108
00109
00110    shared_data_t *buf = malloc( sizeof( shared_data_t ) );
00111    if( NULL == buf )
00112    {
00113       int errnum = errno;
00114       fprintf( stderr, "Encountered error allocating memory for local buffer %s\n",
00115                strerror( errnum ) );
00116    }
00117
00118    while( 1 )
00119    {
00120       sem_wait(&shm->r_sem);
00121       memcpy( buf, shm, sizeof(*shm) );
00122
00123       fprintf( log, "%s\n%s", buf->header, buf->buffer );
00124       fflush( log );
00125
00126       led_toggle( LED3_BRIGHTNESS );
00127       sem_post(&shm->w_sem);
00128    }
00129
00130    return NULL;
00131 }
```

5.6 /home/baquerrj/boulder/ecen5013/project_1/inc/socket.h File Reference

Remote Socket task capable of requesting sensor readings from temperature and light sensor threads.

`#include "common.h"`
Include dependency graph for socket.h: This graph shows which files directly or indirectly include this file:

Functions

- msg_t process_request (msg_t *request)

  Process a request from remote client.
- int socket_init (void)

  Cycle function for remote socket task. Spins in this infinite while-loop checking for new connections to make. When it receives a new connection, it starts processing requests from the client.
- void * socket_fn (void *thread_arg)

  Entry point for remote socket thread.

5.6.1 Detailed Description

Remote Socket task capable of requesting sensor readings from temperature and light sensor threads.

==========

Author

Roberto Baquerizo (baquerrj), roba8460@colorado.edu

5.6.2 Function Documentation

5.6.2.1 process_request()

```
msg_t process_request (
            msg_t * request )
```

Process a request from remote client.

========== Function↩
: process_request

Parameters

∗request	- request from client **response - our response**

```
00050 {
00051    msg_t response = {0};
00052    switch( request->id )
00053    {
00054       case REQUEST_LUX:
00055          response.id = request->id;
00056          response.data.data = get_lux();
00057          sem_wait(&shm->w_sem);
00058          print_header(shm->header);
00059          sprintf( shm->buffer, "Request Lux: %.5f\n",
00060                   response.data.data );
00061          sem_post(&shm->r_sem);
00062          break;
00063       case REQUEST_DARK:
00064          response.id = request->id;
00065          response.data.night = is_dark();
00066          sem_wait(&shm->w_sem);
00067          print_header(shm->header);
00068          sprintf( shm->buffer, "Request Day or Night: %s\n",
00069                   (response.data.night == 0) ? "day" : "night");
00070          sem_post(&shm->r_sem);
00071          break;
00072       case REQUEST_TEMP:
00073          response.id = request->id;
00074          response.data.data = get_temperature();
00075          sem_wait(&shm->w_sem);
00076          print_header(shm->header);
00077          sprintf( shm->buffer, "Request Temperature: %.5f C\n",
00078                   response.data.data );
00079          sem_post(&shm->r_sem);
00080          break;
00081       case REQUEST_TEMP_K:
00082          response.id = request->id;
00083          response.data.data = get_temperature() + 273.15;
00084          sem_wait(&shm->w_sem);
00085          print_header(shm->header);
00086          sprintf( shm->buffer, "Request Temperature: %.5f K\n",
00087                   response.data.data );
00088          sem_post(&shm->r_sem);
00089          break;
00090        case REQUEST_TEMP_F:
00091          response.id = request->id;
00092          response.data.data = (get_temperature() *1.80) + 32.0;
00093          sem_wait(&shm->w_sem);
00094          print_header(shm->header);
00095          sprintf( shm->buffer, "Request Temperature: %.5f F\n",
00096                   response.data.data );
00097          sem_post(&shm->r_sem);
00098          break;
```

```
00099       case REQUEST_CLOSE:
00100          response.id = request->id;
00101          sem_wait(&shm->w_sem);
00102          print_header(shm->header);
00103          sprintf( shm->buffer, "Request Close Connection\n" );
00104          sem_post(&shm->r_sem);
00105          break;
00106       case REQUEST_KILL:
00107          response.id = request->id;
00108          sem_wait(&shm->w_sem);
00109          print_header(shm->header);
00110          sprintf( shm->buffer, "Request Kill Application\n" );
00111          sem_post(&shm->r_sem);
00112          break;
00113       default:
00114          sem_wait(&shm->w_sem);
00115          print_header(shm->header);
00116          sprintf( shm->buffer, "Invalid Request\n" );
00117          sem_post(&shm->r_sem);
00118          break;
00119    }
00120
00121    return response;
00122 }
```

Here is the caller graph for this function:

5.6.2.2 socket_fn()

```
void* socket_fn (
             void * thread_args )
```

Entry point for remote socket thread.

========== Function↩
: socket_fn

Parameters

∗thread_args	- thread arguments (if any)

Returns

NULL - We don't really exit from this function,

since the exit point is thread_exit()

```
00306 {
00307    /* Get time that thread was spawned */
00308    struct timespec time;
00309    clock_gettime(CLOCK_REALTIME, &time);
00310
00311    /* Get pointer to shared memory struct */
00312    shm = get_shared_memory();
00313
00314    int server = socket_init();
00315    if( -1 == server )
00316    {
00317       fprintf( stderr, "Failed to set up server!\n" );
00318       thread_exit( EXIT_INIT );
00319    }
00320
00321    /* Write initial state to shared memory */
00322    sem_wait(&shm->w_sem);
00323    print_header(shm->header);
00324    sprintf( shm->buffer, "Hello World! Start Time: %ld.%ld secs\n",
00325             time.tv_sec, time.tv_nsec );
00326    /* Signal to logger that shared memory has been updated */
```

```
00327     sem_post(&shm->r_sem);
00328
00329     cycle( server );
00330
00331     thread_exit( EXIT_CLEAN );
00332     return NULL;
00333 }
```

5.6.2.3 socket_init()

```
int socket_init (
             void  )
```

Cycle function for remote socket task. Spins in this infinite while-loop checking for new connections to make. When it receives a new connection, it starts processing requests from the client.

========== Function↩
: cycle

Parameters

server	- server socket file descriptor **void**

Cycle function for remote socket task. Spins in this infinite while-loop checking for new connections to make. When it receives a new connection, it starts processing requests from the client.

========== Function↩
: socket_init

Parameters

void	**server - file descriptor for newly created socket for server**

```
00230 {
00231     int retVal = 0;
00232     int opt = 1;
00233     struct sockaddr_in addr;
00234
00235     int server = socket( AF_INET, SOCK_STREAM, 0 );
00236     if( 0 == server )
00237     {
00238        int errnum = errno;
00239        fprintf( stderr, "Encountered error creating new socket (%s)\n",
00240                 strerror( errnum ) );
00241        return -1;
00242     }
00243
00244     retVal = setsockopt( server, SOL_SOCKET, SO_REUSEPORT | SO_REUSEADDR, &(opt), sizeof(opt) );
00245     if( 0 != retVal )
00246     {
00247        int errnum = errno;
00248        sem_wait(&shm->w_sem);
00249        print_header(shm->header);
```

```
00250       sprintf( shm->buffer, "Encountered error setting socket options (%s)\n",
00251                strerror( errnum ) );
00252       sem_post(&shm->r_sem);
00253       return -1;
00254    }
00255
00256    addr.sin_family = AF_INET;
00257    addr.sin_addr.s_addr = INADDR_ANY;
00258    addr.sin_port = htons( PORT );
00259
00260    /* Attempt to bind socket to address */
00261    retVal = bind( server, (struct sockaddr*)&addr, sizeof( addr ) );
00262    if( 0 > retVal )
00263    {
00264       int errnum = errno;
00265       sem_wait(&shm->w_sem);
00266       print_header(shm->header);
00267       sprintf( shm->buffer, "Encountered error binding the new socket (%s)\n",
00268                strerror( errnum ) );
00269       sem_post(&shm->r_sem);
00270       return -1;
00271    }
00272
00273    /* Try to listen */
00274    retVal = listen( server, 10 );
00275    if( 0 > retVal )
00276    {
00277       int errnum = errno;
00278       sem_wait(&shm->w_sem);
00279       print_header(shm->header);
00280       sprintf( shm->buffer, "Encountered error listening with new socket (%s)\n",
00281                strerror( errnum ) );
00282       sem_post(&shm->r_sem);
00283       return -1;
00284    }
00285
00286    sem_wait(&shm->w_sem);
00287    print_header(shm->header);
00288    sprintf( shm->buffer, "Created new socket [%d]!\n", server );
00289    sem_post(&shm->r_sem);
00290
00291    return server;
00292 }
```

Here is the caller graph for this function:

5.7 /home/baquerrj/boulder/ecen5013/project_1/inc/temperature.h File Reference

Header for temperature sensor thread.

```
#include "common.h"
#include "i2c.h"
#include <mqueue.h>
```

Include dependency graph for temperature.h: This graph shows which files directly or indirectly include this file:

Data Structures

- struct conv_res_t
- struct tmp102_mode_t
- struct tmp102_config_t

Macros

- #define **TEMP_QUEUE_NAME** "/temperature-queue"
- #define TMP102_SLAVE (0x48)
- #define TMP102_REG_TEMP (0x00)
- #define **TMP102_REG_CONFIG** (0x01)
- #define **TMP102_TLOW** (0x02)
- #define **TMP102_THIGH** (0x03)
- #define TMP102_SHUTDOWN_MODE (1)
- #define **TMP102_THERMOSTAT_MODE** (1)
- #define **TMP102_POLARITY** (1)
- #define **TMP102_FAULT_QUEUE** (1)
- #define **TMP102_RESOLUTION_0** (2)
- #define **TMP102_RESOLUTION_1** (4)
- #define **TMP102_EXTENDED_MODE** (0)
- #define **TMP102_CONVERSION_RATE** (2)

Functions

- float get_temperature (void)

  Returns last temperature reading we have.
- int tmp102_write_config (tmp102_config_t *config_reg)

  Write configuration register of TMP102 sensor.
- int tmp102_get_temp (float *temperature)

  Read temperature registers fo TMP102 sensor and decode temperature value.
- int tmp102_write_thigh (float thigh)

  Write value thigh (in celsius) to Thigh register for TMP102 sensor.
- int tmp102_write_tlow (float tlow)

  Write value tlow (in celsius) to Tlow register for TMP102 sensor.
- int tmp102_read_thigh (float *thigh)

  Read value of THigh register of TMP102 sensor and store value (in celsius) in thigh.
- int tmp102_read_tlow (float *tlow)

  Read value of TLow register of TMP102 sensor and store value (in celsius) in tlow.
- mqd_t get_temperature_queue (void)

  Get file descriptor for temperature sensor thread. Called by watchdog thread in order to be able to send heartbeat check via queue.
- int temp_queue_init (void)

  Initialize message queue for temperature sensor thread.
- void * temperature_fn (void *thread_args)

  Entry point for temperature sensor processing thread.

5.7.1 Detailed Description

Header for temperature sensor thread.

==========

Author

Roberto Baquerizo (baquerrj), roba8460@colorado.edu

5.7.2 Macro Definition Documentation

5.7.2.1 TMP102_REG_TEMP

```
#define TMP102_REG_TEMP (0x00)
```

Regsiter addresses for TMP102

5.7.2.2 TMP102_SHUTDOWN_MODE

```
#define TMP102_SHUTDOWN_MODE (1)
```

Default configuration

5.7.2.3 TMP102_SLAVE

```
#define TMP102_SLAVE (0x48)
```

Default address for Temperature Sensor TMP102

5.7.3 Function Documentation

5.7.3.1 get_temperature()

```
float get_temperature (
            void  )
```

Returns last temperature reading we have.

========== Function↩
: get_temperature

Parameters

void	

Returns

last_temp_value - last temperature reading we have

<+DETAILED+>

```
00066 {
```

```
00067     return last_temp_value;
00068 }
```

5.7.3.2 get_temperature_queue()

```
mqd_t get_temperature_queue (
              void  )
```

Get file descriptor for temperature sensor thread. Called by watchdog thread in order to be able to send heartbeat check via queue.

========== Function↩
: get_temperature_queue

Parameters

void	**temp_queue - file descriptor for temperature sensor thread message queue**

```
00415 {
00416     return temp_queue;
00417 }
```

5.7.3.3 temp_queue_init()

```
int temp_queue_init (
              void  )
```

Initialize message queue for temperature sensor thread.

========== Function↩
: temp_queue_init

Parameters

void	**msg_q - file descriptor for initialized message queue**

```
00429 {
00430     /* unlink first in case we hadn't shut down cleanly last time */
00431     mq_unlink( TEMP_QUEUE_NAME );
00432
00433     struct mq_attr attr;
00434     attr.mq_flags = 0;
00435     attr.mq_maxmsg = MAX_MESSAGES;
```

```
00436    attr.mq_msgsize = sizeof( msg_t );
00437    attr.mq_curmsgs = 0;
00438
00439    int msg_q = mq_open( TEMP_QUEUE_NAME, O_CREAT | O_RDWR, 0666, &attr );
00440    if( 0 > msg_q )
00441    {
00442       int errnum = errno;
00443       sem_wait(&shm->w_sem);
00444       print_header(shm->header);
00445       sprintf( shm->buffer, "ERROR: Encountered error creating message queue %s: (%s)\n",
00446                TEMP_QUEUE_NAME, strerror( errnum ) );
00447       sem_post(&shm->r_sem);
00448    }
00449    return msg_q;
00450 }
```

5.7.3.4 temperature_fn()

```
void* temperature_fn (
             void * thread_args )
```

Entry point for temperature sensor processing thread.

========== Function↩
: temperature_fn

Parameters

thread_args	- void ptr to arguments used to initialize thread

Returns

NULL - We don't really exit from this function,

since the exit point is thread_exit()

```
00463 {
00464    /* Get time that thread was spawned */
00465    struct timespec time;
00466    clock_gettime(CLOCK_REALTIME, &time);
00467    shm = get_shared_memory();
00468
00469    /* Write initial state to shared memory */
00470    sem_wait(&shm->w_sem);
00471    print_header(shm->header);
00472    sprintf( shm->buffer, "Hello World! Start Time: %ld.%ld secs\n",
00473             time.tv_sec, time.tv_nsec );
00474    /* Signal to logger that shared memory has been updated */
00475    sem_post(&shm->r_sem);
00476
00477    signal(SIGUSR1, sig_handler);
00478    signal(SIGUSR2, sig_handler);
00479
00480    temp_queue = temp_queue_init();
00481    if( 0 > temp_queue )
00482    {
00483       thread_exit( EXIT_INIT );
00484    }
00485
00486    int retVal = i2c_init( &i2c_tmp102 );
00487    if( EXIT_INIT == retVal )
00488    {
00489       sem_wait(&shm->w_sem);
00490       print_header(shm->header);
00491       sprintf( shm->buffer, "ERROR: Failed to initialize I2C for temperature sensor!\n" );
00492       sem_post(&shm->r_sem);
```

```
00493       thread_exit( EXIT_INIT );
00494    }
00495
00496    timer_setup( &timerid, &timer_handler );
00497
00498    timer_start( &timerid, 1000000 );
00499    cycle();
00500
00501    thread_exit( 0 );
00502    return NULL;
00503 }
```

5.7.3.5 tmp102_get_temp()

```
int tmp102_get_temp (
            float * temperature )
```

Read temperature registers fo TMP102 sensor and decode temperature value.

========== Function↩
: tmp102_get_temp

Parameters

∗temperature	- pointer to location to write decoded value to

Returns

EXIT_CLEAN if successful, otherwise EXIT_ERROR

<+DETAILED+>

========== Function↩
: tmp102_get_temp

Parameters

∗temperature	- pointer to location to write decoded value to **EXIT_CLEAN if successful, otherwise EXIT_ERROR**

```
00097 {
00098    uint8_t buffer[2] = {0};
00099    int retVal = i2c_read( TMP102_SLAVE, TMP102_REG_TEMP, buffer, sizeof(
       buffer ) );
00100    if( 0 > retVal )
00101    {
00102       return EXIT_ERROR;
00103    }
00104
00105    uint16_t tmp = 0;
00106    tmp = 0xfff & ( ((uint16_t)buffer[0] << 4 ) | (buffer[1] >> 4 ) ); /* buffer[0] = MSB(15:8)
00107                                                                          buffer[1] = LSB(7:4) */
00108    if( 0x800 & tmp )
00109    {
00110       tmp = ( (~tmp ) + 1 ) & 0xfff;
```

```
00111       *temperature = -1.0 * (float)tmp * 0.0625;
00112    }
00113    else
00114    {
00115       *temperature = ((float)tmp) * 0.0625;
00116    }
00117
00118    return EXIT_CLEAN;
00119 }
```

5.7.3.6 tmp102_read_thigh()

```
int tmp102_read_thigh (
              float * thigh )
```

Read value of THigh register of TMP102 sensor and store value (in celsius) in thigh.

========== Function↩
: tmp102_read_thigh

Parameters

thigh	- pointer to location to store decoded temperature value to **EXIT_CLEAN if successful, EXIT_ERROR otherwise**

```
00221 {
00222    uint16_t tmp = 0;
00223
00224    int retVal = i2c_read( TMP102_SLAVE, TMP102_THIGH, (uint8_t*)&tmp, sizeof( tmp ) );
00225    if( 0 > retVal )
00226    {
00227       sem_wait(&shm->w_sem);
00228       print_header(shm->header);
00229       sprintf( shm->buffer, "Could not read from TLow register!\n" );
00230       sem_post(&shm->r_sem);
00231       return EXIT_ERROR;
00232    }
00233
00234    if( tmp & 0x800 )
00235    {
00236       tmp = ~(tmp) + 1;
00237       *thigh = -1 * ( (float)tmp * 0.0625 );
00238    }
00239    else
00240    {
00241       *thigh = (float)tmp * 0.0625;
00242    }
00243
00244    return EXIT_CLEAN;
00245 }
```

5.7.3.7 tmp102_read_tlow()

```
int tmp102_read_tlow (
              float * tlow )
```

Read value of TLow register of TMP102 sensor and store value (in celsius) in tlow.

========== Function↩
: tmp102_read_tlow

Parameters

tlow	- pointer to location to store decoded temperature value to **EXIT_CLEAN if successful, EXIT_ERROR otherwise**

```
00258 {
00259    uint16_t tmp = 0;
00260
00261    int retVal = i2c_read( TMP102_SLAVE, TMP102_TLOW, (uint8_t*)&tmp, sizeof( tmp ) );
00262    if( 0 > retVal )
00263    {
00264       sem_wait(&shm->w_sem);
00265       print_header(shm->header);
00266       sprintf( shm->buffer, "Could not read from TLow register!\n" );
00267       sem_post(&shm->r_sem);
00268       return retVal;
00269    }
00270
00271    if( tmp & 0x800 )
00272    {
00273       tmp = ~(tmp) + 1;
00274       *tlow = -1 * (float)tmp * 0.0625;
00275    }
00276    else
00277    {
00278       *tlow = (float)tmp * 0.0625;
00279    }
00280
00281    return retVal;
00282 }
```

5.7.3.8 tmp102_write_config()

```
int tmp102_write_config (
             tmp102_config_t * config_reg )
```

Write configuration register of TMP102 sensor.

========== Function↩
: tmp102_write_config

Parameters

∗config_reg	- pointer to struct with values to write to configuration register **see i2c_write()**

```
00081 {
00082    int retVal = i2c_write( TMP102_SLAVE, TMP102_REG_CONFIG, *((uint16_t*)&config_reg)
      );
00083
00084    return retVal;
00085 }
```

5.7.3.9 tmp102_write_thigh()

```
int tmp102_write_thigh (
            float thigh )
```

Write value thigh (in celsius) to Thigh register for TMP102 sensor.

========== Function↩
: tmp102_write_thigh

Parameters

thigh	- value to write to Thigh register **EXIT_CLEAN if successful, otherwise EXIT_ERROR**

```
00131 {
00132    if( (-56.0 > thigh) || (151.0 < thigh) )
00133    {
00134       thigh = 80.0;
00135    }
00136
00137    thigh /= 0.0625;
00138    uint16_t tmp;
00139
00140    if( 0 > thigh )
00141    {
00142       tmp = ( (uint16_t)thigh << 4 );
00143       tmp &= 0x7fff;
00144    }
00145    else
00146    {
00147       thigh *= -1;
00148       tmp = (uint16_t)thigh;
00149       tmp = ~(tmp) + 1;
00150       tmp = tmp << 4;
00151    }
00152
00153    int retVal = i2c_write( TMP102_SLAVE, TMP102_THIGH, tmp );
00154    if( 0 > retVal )
00155    {
00156       sem_wait(&shm->w_sem);
00157       print_header(shm->header);
00158       sprintf( shm->buffer, "Could not write value to THigh register!\n" );
00159       sem_post(&shm->r_sem);
00160       return EXIT_ERROR;
00161    }
00162
00163    return EXIT_CLEAN;
00164 }
```

5.7.3.10 tmp102_write_tlow()

```
int tmp102_write_tlow (
            float tlow )
```

Write value tlow (in celsius) to Tlow register for TMP102 sensor.

========== Function↩
: tmp102_write_tlow

Parameters

tlow	- value to write to Tlow register **EXIT_CLEAN if successful, otherwise EXIT_ERROR**

```
00176 {
00177    if( (-56.0 > tlow) || (151.0 < tlow ) )
00178    {
00179       tlow = 75.0;
00180    }
00181
00182    tlow /= 0.0625;
00183    uint16_t tmp;
00184
00185    if( 0 < tlow )
00186    {
00187       tmp = ( (uint16_t)tlow << 4 );
00188       tmp &= 0x7fff;
00189    }
00190    else
00191    {
00192       tlow *= -1;
00193       tmp = (uint16_t)tlow;
00194       tmp = ~(tmp) + 1;
00195       tmp = tmp << 4;
00196    }
00197
00198    int retVal = i2c_write( TMP102_SLAVE, TMP102_TLOW, tmp );
00199    if( 0 > retVal )
00200    {
00201       sem_wait(&shm->w_sem);
00202       print_header(shm->header);
00203       sprintf( shm->buffer, "Could not write value to TLow register!\n" );
00204       sem_post(&shm->r_sem);
00205       return EXIT_ERROR;
00206    }
00207
00208    return EXIT_CLEAN;
00209 }
```

5.8 /home/baquerrj/boulder/ecen5013/project_1/inc/watchdog.h File Reference

Watchdog thread header.

```
#include "common.h"
#include <mqueue.h>
```

Include dependency graph for watchdog.h: This graph shows which files directly or indirectly include this file:

Macros

- #define **WATCHDOG_QUEUE_NAME** "/watchdog-queue"
- #define **NUM_THREADS** 4

Enumerations

- enum **thread_e** {
  THREAD_TEMP = 0, **THREAD_LIGHT**, **THREAD_LOGGER**, **THREAD_SOCKET**,
  THREAD_MAX }

Functions

- void kill_threads (void)

  Function to kill children threads.
- void check_threads (union sigval sig)

  Periodically send message via message queue for temperature and sensor threads to check for health. This function is registered as the timer hanlder for the timer owned by the watchdog.
- int watchdog_queue_init (void)

  Initalize message queue for watchdog.
- int watchdog_init (void)

  Initialize watchdog, calling appropriate functions to do so. E.g. calling timer_setup and timer_start to set up timer.
- void * watchdog_fn (void *thread_args)

  Entry point for wachtdog.

Variables

- volatile int **threads_status** [NUM_THREADS]
- pthread_mutex_t **alive_mutex**

5.8.1 Detailed Description

Watchdog thread header.

==========

Author

Roberto Baquerizo (baquerrj), roba8460@colorado.edu

5.8.2 Function Documentation

5.8.2.1 check_threads()

```
void check_threads (
            union sigval sig )
```

Periodically send message via message queue for temperature and sensor threads to check for health. This function is registered as the timer hanlder for the timer owned by the watchdog.

========== Function↩
: check_threads

Parameters

sig	**void**

```
00104 {
00105    int retVal = 0;
00106    msg_t request = {0};
00107    request.id = REQUEST_STATUS;
00108    request.src = watchdog_queue;
00109
00110    if( (0 == threads_status[THREAD_TEMP]) && (0 == threads_status[THREAD_LIGHT] ) )
00111    {
00112       pthread_mutex_lock( &alive_mutex );
00113       threads_status[THREAD_TEMP]++;
00114       threads_status[THREAD_LIGHT]++;
00115       pthread_mutex_unlock( &alive_mutex );
00116       retVal = mq_send( thread_msg_q[THREAD_TEMP], (const char*)&request, sizeof( request ), 0 );
00117       if( 0 > retVal )
00118       {
00119          int errnum = errno;
00120          fprintf( stderr, "Encountered error sending status request from watchdog: (%s)\n",
00121                   strerror( errnum ) );
00122       }
00123       retVal = mq_send( thread_msg_q[THREAD_LIGHT], (const char*)&request, sizeof( request ), 0 );
00124       if( 0 > retVal )
00125       {
00126          int errnum = errno;
00127          fprintf( stderr, "Encountered error sending status request from watchdog: (%s)\n",
00128                   strerror( errnum ) );
00129       }
00130    }
00131    else
00132    {
00133       fprintf( stderr, "One of the threads did not return!\n" );
00134       fprintf( stderr, "thread_status[THREAD_TEMP] = %d\nthread_status[THREAD_LIGHT] = %d\n",
00135             threads_status[THREAD_TEMP], threads_status[THREAD_LIGHT] );
00136       kill_threads();
00137       thread_exit( EXIT_ERROR );
00138    }
00139
00140    return;
00141 }
```

5.8.2.2 kill_threads()

```
void kill_threads (
             void  )
```

Function to kill children threads.

========== Function↩
: kill_threads

Parameters

void	
	void

```
00073 {
00074    fprintf( stdout, "watchdog caught signals - killing thread [%ld]\n",
00075             threads->temp_thread );
00076    fflush( stdout );
00077    pthread_kill( threads->temp_thread, SIGUSR1 );
00078
00079    fprintf( stdout, "watchdog caught signals - killing thread [%ld]\n",
00080             threads->light_thread );
00081    fflush( stdout );
00082    pthread_kill( threads->light_thread, SIGUSR1 );
00083
00084    fprintf( stdout, "watchdog caught signals - killing thread [%ld]\n",
```

```
00085             threads->logger_thread );
00086     fflush( stdout );
00087     pthread_kill( threads->logger_thread, SIGUSR1 );
00088     free( threads );
00089     return;
00090 }
```

Here is the caller graph for this function:

5.8.2.3 watchdog_fn()

```
void* watchdog_fn (
              void * thread_args )
```

Entry point for wachtdog.

========== Function↩
: watchdog_fn

Parameters

thread_args	- void ptr used to pass thread identifiers (pthread_t) for child threads we have to check for health

Returns

NULL - We don't really exit from this function,

since the exit point for threads is thread_exit()

```
00217 {
00218     signal( SIGUSR2, sig_handler );
00219     exit_e retVal = EXIT_ERROR;
00220     if( NULL == thread_args )
00221     {
00222        print_header( NULL );
00223        fprintf( stderr, "Encountered null pointer!\n" );
00224        pthread_exit(&retVal);
00225     }
00226     else
00227     {
00228        threads = malloc( sizeof( struct thread_id_s ) );
00229        threads = (struct thread_id_s*)thread_args;
00230     }
00231
00232     watchdog_init();
00233
00234     while(1);
00235     return NULL;
00236 }
```

5.8.2.4 watchdog_init()

```
int watchdog_init (
              void  )
```

Initialize watchdog, calling appropriate functions to do so. E.g. calling timer_setup and timer_start to set up timer.

========== Function↩
: watchdog_init

Parameters

void	

EXIT_CLEAN, otherwise EXIT_INIT

```
00184 {
00185    watchdog_queue = watchdog_queue_init();
00186    if( 0 > watchdog_queue )
00187    {
00188       thread_exit( EXIT_INIT );
00189    }
00190
00191    while( 0 == (thread_msg_q[THREAD_TEMP] = get_temperature_queue()) );
00192    while( 0 == (thread_msg_q[THREAD_LIGHT] = get_light_queue()) );
00193
00194    fprintf( stderr, "Watchdog says: Temp Queue FD: %d\n", thread_msg_q[0] );
00195    fprintf( stderr, "Watchdog says: Light Queue FD: %d\n", thread_msg_q[1] );
00196
00197    pthread_mutex_init( &alive_mutex, NULL );
00198    timer_setup( &timerid, &check_threads );
00199
00200    timer_start( &timerid, 4000000 );
00201
00202    return EXIT_CLEAN;
00203 }
```

5.8.2.5 watchdog_queue_init()

```
int watchdog_queue_init (
             void  )
```

Initalize message queue for watchdog.

========== Function↩
: watchdog_queue_init

Parameters

void	

msg_q - file descriptor for initialized message queue

```
00153 {
00154    /* unlink first in case we hadn't shut down cleanly last time */
00155    mq_unlink( WATCHDOG_QUEUE_NAME );
00156
00157    struct mq_attr attr;
00158    attr.mq_flags = 0;
00159    attr.mq_maxmsg = MAX_MESSAGES;
00160    attr.mq_msgsize = sizeof( msg_t );
00161    attr.mq_curmsgs = 0;
00162
00163    int msg_q = mq_open( WATCHDOG_QUEUE_NAME, O_CREAT | O_RDWR, 0666, &attr );
00164    if( 0 > msg_q )
00165    {
00166       int errnum = errno;
00167       fprintf( stderr, "Encountered error creating message queue %s: (%s)\n",
00168                WATCHDOG_QUEUE_NAME, strerror( errnum ) );
00169    }
00170    return msg_q;
00171 }
```

5.9 /home/baquerrj/boulder/ecen5013/project_1/src/common.c File Reference

Defines types and functions common between the threads of the application.

```
#include "common.h"
#include <errno.h>
#include <string.h>
#include <time.h>
#include <stdlib.h>
#include <signal.h>
#include <fcntl.h>
#include <sys/syscall.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/stat.h>
```

Include dependency graph for common.c:

Functions

- void print_header (char *buffer)

  Write a string formatted with the TID of the thread calling this function and a timestamp to the log buffer.
- void thread_exit (int exit_status)

  Common exit point for all threads.
- void * get_shared_memory (void)

  Sets up shared memory location for logging.
- int sems_init (shared_data_t *shm)
- int timer_setup (timer_t *id, void(*handler)(union sigval))

  Initializes a timer identified by timer_t id.
- int timer_start (timer_t *id, unsigned long usecs)

  Starts the timer with interval usecs.

5.9.1 Detailed Description

Defines types and functions common between the threads of the application.

==========

Author

Roberto Baquerizo (baquerrj), roba8460@colorado.edu

5.9.2 Function Documentation

5.9.2.1 get_shared_memory()

```
void* get_shared_memory (
            void  )
```

Sets up shared memory location for logging.

========== Function↩
: get_shared_memory

Parameters

void	∗**shm_p - pointer to shared memory object**

```
00083 {
00084    struct shared_data *shm_p;
00085
00086    int shm_fd = shm_open( SHM_SEGMENT_NAME, O_CREAT | O_EXCL | O_RDWR, 0666 );
00087    if( 0 > shm_fd )
00088    {
00089       int errnum = errno;
00090       if( EEXIST == errnum )
00091       {
00092            /* Already exists: open again without O_CREAT */
00093            shm_fd = shm_open(SHM_SEGMENT_NAME, O_RDWR, 0);
00094        }
00095       else
00096       {
00097          fprintf( stderr, "Encountered error opening shared memory: %s\n",
00098                   strerror( errnum ) );
00099          exit(EXIT_FAILURE);
00100       }
00101    }
00102    else
00103    {
00104       fprintf( stdout, "Creating shared memory and setting size to %u bytes\n",
00105                sizeof( shared_data_t ) );
00106
00107       if( 0 > ftruncate( shm_fd, sizeof( shared_data_t )) )
00108       {
00109          int errnum = errno;
00110          fprintf( stderr, "Encountered error setting size of shared memroy: %s\n",
00111                   strerror( errnum ) );
00112          exit(EXIT_FAILURE);
00113       }
00114    }
00115
00116    /* Map the shared memory */
00117    shm_p = mmap(NULL, sizeof( shared_data_t ), PROT_READ | PROT_WRITE,
00118             MAP_SHARED, shm_fd, 0);
00119
00120    if( NULL == shm_p )
00121    {
00122       int errnum = errno;
00123       fprintf( stderr, "Encountered error memory mapping shared memory: %s\n",
00124                strerror( errnum ) );
00125         exit(EXIT_FAILURE);
00126    }
00127    return shm_p;
00128 }
```

Here is the caller graph for this function:

5.9.2.2 print_header()

```
void print_header (
             char * buffer )
```

Write a string formatted with the TID of the thread calling this function and a timestamp to the log buffer.

========== Function↩
: print_header

Parameters

**buffer*	- pointer to where we should copy formatted string to if NULL, we print to stderr **void**

```
00036 {
00037
00038    struct timespec time;
00039    clock_gettime(CLOCK_REALTIME, &time);
00040
00041    if( NULL == buffer )
00042    {
00043       fprintf( stderr, "\n=======================================================\n" );
00044       fprintf( stderr, "Thread [%d]: %ld.%ld secs\n",
00045                (pid_t)syscall(SYS_gettid), time.tv_sec, time.tv_nsec );
00046       fflush( stderr );
00047    }
00048    else if( NULL != buffer )
00049    {
00050       char tmp[100]  = "\n=======================================================\n";
00051       char tmp2[100];
00052       sprintf( tmp2, "Thread [%d]: %ld.%ld secs\n",
00053                (pid_t)syscall(SYS_gettid), time.tv_sec, time.tv_nsec );
00054       strcat( tmp, tmp2 );
00055       strcpy( buffer, tmp );
00056    }
00057    return;
00058 }
```

Here is the caller graph for this function:

5.9.2.3 sems_init()

```
int sems_init (
              shared_data_t * shm )
```

========== Function↩
: sems_init Initialize semaphores for shared memory

Parameters

**shm*	- pointer to shared memory object **EXIT_CLEAN if successful, otherwise EXIT_INIT**

```
00132 {
00133    int retVal = 0;
00134    retVal = sem_init( &shm->w_sem, 1, 1 );
00135    if( 0 > retVal )
00136    {
00137       int errnum = errno;
00138       fprintf( stderr, "Encountered error initializing write semaphore: %s\n",
00139                strerror( errnum ) );
00140       return EXIT_INIT;
00141    }
00142    retVal = sem_init( &shm->r_sem, 1, 0 );
00143    if( 0 > retVal )
00144    {
00145       int errnum = errno;
00146       fprintf( stderr, "Encountered error initializing read semaphore: %s\n",
00147                strerror( errnum ) );
```

```
00148       return EXIT_INIT;
00149    }
00150    return EXIT_CLEAN;
00151 }
```

5.9.2.4 thread_exit()

```
void thread_exit (
             int exit_status )
```

Common exit point for all threads.

========== Function↩
: thread_exit

Parameters

exit_status	- reason for exit (signal number) **void**

```
00061 {
00062    struct timespec time;
00063    clock_gettime(CLOCK_REALTIME, &time);
00064
00065    switch( exit_status )
00066    {
00067       case SIGUSR1:
00068          fprintf( stdout, "Caught SIGUSR1 Signal! Exiting...\n");
00069          break;
00070       case SIGUSR2:
00071          fprintf( stdout, "Caught SIGUSR2 Signal! Exiting...\n");
00072          break;
00073       default:
00074          break;
00075    }
00076    fprintf( stdout, "Goodbye World! End Time: %ld.%ld secs\n",
00077             time.tv_sec, time.tv_nsec );
00078
00079    pthread_exit(EXIT_SUCCESS);
00080 }
```

Here is the caller graph for this function:

5.9.2.5 timer_setup()

```
int timer_setup (
             timer_t * id,
             void(*)(union sigval) timer_handler )
```

Initializes a timer identified by timer_t id.

========== Function↩
: timer_setup

Parameters

∗id	- identifier for new timer
∗handler	- pointer to function to register as the handler for the timer ticks **EXIT_CLEAN if successful, otherwise EXIT_INIT**

```
00156 {
00157    int retVal = 0;
00158    /* Set up timer */
00159    struct sigevent sev;
00160
00161    memset(&sev, 0, sizeof(struct sigevent));
00162
00163    sev.sigev_notify = SIGEV_THREAD;
00164    sev.sigev_notify_function = handler;
00165    sev.sigev_value.sival_ptr = NULL;
00166    sev.sigev_notify_attributes = NULL;
00167
00168    retVal = timer_create( CLOCK_REALTIME, &sev, id );
00169    if( 0 > retVal )
00170    {
00171       int errnum = errno;
00172       fprintf( stderr, "Encountered error creating new timer: (%s)\n",
00173                strerror( errnum ) );
00174       return EXIT_INIT;
00175    }
00176    return EXIT_CLEAN;
00177 }
```

5.9.2.6 timer_start()

```
int timer_start (
              timer_t * id,
              unsigned long usecs )
```

Starts the timer with interval usecs.

========== Function↩
: timer_start

Parameters

∗id	- identifier for new timer
usecs	- timer interval **EXIT_CLEAN if successful, otherwise EXIT_INIT**

```
00181 {
00182    int retVal = 0;
00183    struct itimerspec trigger;
00184
00185    trigger.it_value.tv_sec = usecs / MICROS_PER_SEC;
00186    trigger.it_value.tv_nsec = ( usecs % MICROS_PER_SEC ) * 1000;
00187
00188    trigger.it_interval.tv_sec = trigger.it_value.tv_sec;
00189    trigger.it_interval.tv_nsec = trigger.it_value.tv_nsec;
```

```
00190
00191     retVal = timer_settime( *id, 0, &trigger, NULL );
00192     if( 0 > retVal )
00193     {
00194        int errnum = errno;
00195        fprintf( stderr, "Encountered error starting new timer: (%s)\n",
00196                 strerror( errnum ) );
00197        return EXIT_INIT;
00198     }
00199     return EXIT_CLEAN;
00200 }
```

5.10 /home/baquerrj/boulder/ecen5013/project_1/src/i2c.c File Reference

```
#include "i2c.h"
#include "common.h"
#include <errno.h>
#include <string.h>
```

Include dependency graph for i2c.c:

Functions

- int **i2c_set** (int slave, int addr)
- int i2c_write_byte (int slave, int reg, uint8_t data)

  Writes byte to register address.
- int i2c_write (int slave, int reg, uint16_t data)

  Writes data to register address.
- int i2c_read (int slave, int reg, uint8_t *data, size_t len)

  Reads data from register address.
- int i2c_init (i2c_handle_t *i2c)

  Initialize singleton master i2c context.
- int i2c_stop (i2c_handle_t *i2c)

  Stops i2c instance.

Variables

- static i2c_handle_t * my_i2c = NULL

5.10.1 Detailed Description

==========

Author

Roberto Baquerizo (baquerrj), roba8460@colorado.edu

5.10.2 Function Documentation

5.10.2.1 i2c_init()

```
int i2c_init (
             i2c_handle_t * i2c )
```

Initialize singleton master i2c context.

========== Function↩
: i2c_init

Parameters

∗i2c	- pointer to handle to be master **EXIT_CLEAN on success, otherwise EXIT_INIT**

```
00203 {
00204    if( NULL != my_i2c )
00205    {
00206       i2c = my_i2c;
00207       return EXIT_CLEAN;
00208    }
00209
00210    if( NULL != i2c )
00211    {
00212       i2c->context = mraa_i2c_init_raw( 2 );
00213
00214       if( NULL == i2c->context )
00215       {
00216          int errnum = errno;
00217          fprintf( stderr, "Failed to initialize I2C master instance: (%s)\n",
00218                   strerror( errnum ) );
00219          my_i2c = NULL;
00220          return EXIT_INIT;
00221       }
00222
00223       int retVal = pthread_mutex_init( &i2c->mutex, NULL );
00224       if( 0 > retVal )
00225       {
00226          int errnum = errno;
00227          fprintf( stderr, "Failed to initialize mutex for I2C master instance: (%s)\n",
00228                   strerror( errnum ) );
00229          my_i2c = NULL;
00230          retVal = mraa_i2c_stop( i2c->context );
00231          if( 0 > retVal )
00232          {
00233             mraa_result_print( retVal );
00234          }
00235          return EXIT_INIT;
00236       }
00237       my_i2c = i2c;
00238    }
00239    return EXIT_CLEAN;
00240 }
```

5.10.2.2 i2c_read()

```
int i2c_read (
              int slave,
              int reg,
              uint8_t * data,
              size_t len )
```

Reads data from register address.

========== Function↩
: i2c_read

Parameters

slave	- address of i2c slave
reg	- address to read from
∗data	- pointer to location to store read data
len	- size of memory to read in bytes
	EXIT_CLEAN on success, otherwise one of exit_e

```
00151 {
00152    if( NULL == my_i2c )
00153    {
00154       fprintf( stderr, "I2C master has not been initialized!\n" );
00155       return EXIT_INIT;
00156    }
00157
00158    pthread_mutex_lock( &my_i2c->mutex );
00159
00160    mraa_result_t retVal = mraa_i2c_address( my_i2c->context, slave );
00161    if( 0 != retVal )
00162    {
00163       mraa_result_print( retVal );
00164       pthread_mutex_unlock( &my_i2c->mutex );
00165       return EXIT_ERROR;
00166    }
00167
00168    if( len )
00169    {
00170       retVal = mraa_i2c_read_bytes_data( my_i2c->context, reg, data, len );
00171       pthread_mutex_unlock( &my_i2c->mutex );
00172       if( len != retVal )
00173       {
00174          fprintf( stderr, "Could not read all data from register!\n" );
00175          return EXIT_ERROR;
00176       }
00177    }
00178    else
00179    {
00180       /* only read one byte */
00181       retVal = mraa_i2c_read_byte_data( my_i2c->context, reg );
00182       pthread_mutex_unlock( &my_i2c->mutex );
00183       if( -1 != retVal )
00184       {
00185          *data  = retVal;
00186       }
00187    }
00188
00189    return EXIT_CLEAN;
00190 }
```

Here is the caller graph for this function:

5.10.2.3 i2c_stop()

```
int i2c_stop (
             i2c_handle_t * i2c )
```

Stops i2c instance.

========== Function↩
: i2c_stop

Parameters

*i2c	- pointer to i2c context handle **EXIT_CLEAN on success, otherwise EXIT_ERROR**

```
00253 {
00254    if( NULL == my_i2c )
00255    {
00256       return EXIT_CLEAN;
00257    }
00258    else if( NULL == i2c )
00259    {
00260       return EXIT_CLEAN;
00261    }
00262
```

```
00263     if( my_i2c != i2c )
00264     {
00265        return EXIT_ERROR;
00266     }
00267
00268     while( EBUSY  == pthread_mutex_destroy( &i2c->mutex ) );
00269
00270     mraa_result_t retVal = mraa_i2c_stop( i2c->context );
00271     if( 0 > retVal )
00272     {
00273        mraa_result_print( retVal );
00274        return EXIT_ERROR;
00275     }
00276
00277     my_i2c = NULL;
00278     return EXIT_CLEAN;
00279 }
```

5.10.2.4 i2c_write()

```
int i2c_write (
              int slave,
              int reg,
              uint16_t data )
```

Writes data to register address.

========== Function↩ : i2c_write

Parameters

slave	- address of i2c slave
reg	- address of register to write to
data	- data to write **EXIT_CLEAN on success, otherwise one of exit_e**

```
00114 {
00115     if( NULL == my_i2c )
00116     {
00117        fprintf( stderr, "I2C master has not been initialized!\n" );
00118        return EXIT_INIT;
00119     }
00120
00121     /* take hardware mutex */
00122     pthread_mutex_lock( &my_i2c->mutex );
00123
00124     mraa_result_t retVal = mraa_i2c_address( my_i2c->context, slave );
00125     if( 0 != retVal )
00126     {
00127        mraa_result_print( retVal );
00128        pthread_mutex_unlock( &my_i2c->mutex );
00129        return EXIT_ERROR;
00130     }
00131
00132     retVal = mraa_i2c_write_word_data( my_i2c->context, data, reg );
00133     pthread_mutex_unlock( &my_i2c->mutex );
00134
00135     return EXIT_CLEAN;
00136 }
```

Here is the caller graph for this function:

5.10.2.5 i2c_write_byte()

```
int i2c_write_byte (
            int slave,
            int reg,
            uint8_t data )
```

Writes byte to register address.

========== Function↩
: i2c_write_byte

Parameters

slave	- address of i2c slave
reg	- address of register to write to
data	- data to write **EXIT_CLEAN on success, otherwise one of exit_e**

```
00078 {
00079    if( NULL == my_i2c )
00080    {
00081       fprintf( stderr, "I2C master has not been initialized!\n" );
00082       return EXIT_INIT;
00083    }
00084
00085    /* take hardware mutex */
00086    pthread_mutex_lock( &my_i2c->mutex );
00087
00088    mraa_result_t retVal = mraa_i2c_address( my_i2c->context, slave );
00089    if( 0 != retVal )
00090    {
00091       mraa_result_print( retVal );
00092       pthread_mutex_unlock( &my_i2c->mutex );
00093       return EXIT_ERROR;
00094    }
00095
00096    retVal = mraa_i2c_write_byte_data( my_i2c->context, data, reg );
00097    pthread_mutex_unlock( &my_i2c->mutex );
00098
00099    return EXIT_CLEAN;
00100 }
```

Here is the caller graph for this function:

5.10.3 Variable Documentation

5.10.3.1 my_i2c

```
i2c_handle_t* my_i2c = NULL  [static]
```

Keep around a singleton instance of the master handle

5.11 /home/baquerrj/boulder/ecen5013/project_1/src/led.c File Reference

<+DETAILED+>

```
#include "led.h"
#include <errno.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
```

Include dependency graph for led.c:

Functions

- void get_status (const char *led)
- int set_trigger (const char *led, char *trigger)
- int set_delay (const char *led, int delay)
- void led_on (const char *led)
- void led_off (const char *led)
- void led_toggle (const char *led)

5.11.1 Detailed Description

<+DETAILED+>

==========

Author

Roberto Baquerizo (baquerrj), roba8460@colorado.edu

5.11.2 Function Documentation

5.11.2.1 get_status()

```
void get_status (
            const char * led )
```

========== Function↩
: get_status

Parameters

<+NAME+>	<+DESCRIPTION+>

Returns

<+DESCRIPTION+>

<+DETAILED+>

```
00041 {
00042     return;
00043 }
```

5.11.2.2 led_off()

```
void led_off (
             const char * led )
```

== Function↩
: led_off

Parameters

<+NAME+>	<+DESCRIPTION+>

Returns

<+DESCRIPTION+>

<+DETAILED+>

```
00126 {
00127     FILE *fp;
00128     fp = fopen( led, "w+" );
00129     fprintf( fp, "0" );
00130     fclose( fp );
00131     return;
00132 }
```

Here is the caller graph for this function:

5.11.2.3 led_on()

```
void led_on (
             const char * led )
```

== Function↩
: led_on

Parameters

<+NAME+>	<+DESCRIPTION+>

Returns

<+DESCRIPTION+>

<+DETAILED+>

```
00106 {
00107    FILE *fp;
00108    fp = fopen( led, "w+" );
00109    fprintf( fp, "1" );
00110    fclose( fp );
00111    return;
00112 }
```

Here is the caller graph for this function:

5.11.2.4 led_toggle()

```
void led_toggle (
             const char * led )
```

========== Function↩
: led_toggle

Parameters

<+NAME+>	<+DESCRIPTION+>

Returns

<+DESCRIPTION+>

<+DETAILED+>

```
00145 {
00146    FILE *fp;
00147    fp = fopen( led, "rt" );
00148    fseek( fp, 0, SEEK_END );
00149    long size = ftell( fp );
00150    rewind( fp );
00151
00152    char *value = (char*) malloc( sizeof(char) * size );
00153    fread( value, 1, size, fp );
00154    fclose( fp );
00155    switch( *value )
00156    {
00157       case '0':
00158          led_on( led );
00159          break;
00160       case '1':
00161          led_off( led );
00162          break;
00163       default:
00164          break;
00165    }
00166    return;
00167 }
```

Here is the caller graph for this function:

5.11.2.5 set_delay()

```
int set_delay (
              const char * led,
              int delay )
```

========== Function↩
: set_delay

Parameters

<+NAME+>	<+DESCRIPTION+>

Returns

<+DESCRIPTION+>

<+DETAILED+>

```
00081 {
00082    FILE *fp = fopen( led, "w+" );
00083    if( NULL == fp )
00084    {
00085       int errnum = errno;
00086       fprintf( stderr, "Encuntered error trying to set delay for %s (%s)\nAre you sure LED is in correct
       configuration?\n",
00087                led, strerror ( errnum ) );
00088       return -1;
00089    }
00090    fprintf( fp, "%u", delay );
00091    fclose( fp );
00092    return delay;
00093 }
```

5.11.2.6 set_trigger()

```
int set_trigger (
              const char * led,
              char * trigger )
```

========== Function↩
: get_trigger

Parameters

<+NAME+>	<+DESCRIPTION+>

Returns

<+DESCRIPTION+>

<+DETAILED+>

```
00056 {
```

```
00057    FILE *fp = fopen( led, "w+" );
00058    if( NULL == fp )
00059    {
00060       int errnum = errno;
00061       fprintf( stderr, "Encountered error trying to set trigger %s for %s (%s)\n",
00062                trigger, led, strerror ( errnum ) );
00063       return -1;
00064    }
00065    fprintf( fp, "%s", trigger );
00066    fclose( fp );
00067    return 0;
00068 }
```

5.12 /home/baquerrj/boulder/ecen5013/project_1/src/light.c File Reference

Interface to APDS9301 Light Sensor.

```
#include "watchdog.h"
#include "light.h"
#include "led.h"
#include <errno.h>
#include <time.h>
#include <string.h>
#include <math.h>
```

Include dependency graph for light.c:

Functions

- static void sig_handler (int signo)

  Signal handler for light sensor thread. On normal operation, we should be receving SIGUSR1/2 signals from watchdog when prompted to exit. So, we close the message queue and timer this thread owns.
- static void timer_handler (union sigval sig)

  Timer handler function for light sensor thread When woken up by the timer, get lux reading and write state to shared memory.
- float get_lux (void)

  Returns last lux reading.
- int is_dark (void)

  Returns int speciyfing if it is night or day.
- int apds9301_set_config (void)

  Set configuration of light sensor. For the APDS9301, the configuration is spread out across the: Timing Register, Interrupt Control Register, and Control Register. So, I have to write to all of these to set the config.
- int apds9301_set_integration (uint8_t val)

  Sets the integration time for APDS9301 by writing a value to bits INTEG of the Timing Register.
- int apds9301_clear_interrupt (void)

  Clears any pending interrupt for APDS9301 by writing a 1 to the CLEAR bit of the Command Register.
- int apds9301_set_interrupt (uint8_t enable)

  Enables or disables interrupts for APDS9301 by setting or clearing the INTR bits of the Interrupt Control Register.
- int apds9301_set_gain (uint8_t gain)

  Sets gain for APDS9301 by setting or clearing the GAIN bit of the Timing Register.
- int apds9301_read_control (uint8_t *data)

  Read contents of Control Register.
- int apds9301_write_threshold_low (uint16_t threshold)

  Write value to low threshold register.
- int apds9301_read_threshold_low (uint16_t *threshold)

  Read value from low threshold register.

- int apds9301_write_threshold_high (uint16_t threshold)

  Write value to high threshold register.
- int apds9301_read_threshold_high (uint16_t *threshold)

  Read value from high threshold register.
- int apds9301_read_id (uint8_t *id)

  Read APDS9301 Identification Register.
- int apds9301_get_lux (float *lux)

  Read ADC Registers and calculate lux in lumen using equations from APDS9301 datasheet.
- int apds9301_read_data0 (uint16_t *data)

  Read ADC register for channel 0.
- int apds9301_read_data1 (uint16_t *data)

  Read ADC register for channel 1.
- int apds9301_power (uint16_t on)

  power on (or off) APDS9301 as set by paramater
- static void cycle (void)

  Cycle function for light sensor thread We wait in this while loop checking for requests from watchdog for health status.
- mqd_t get_light_queue (void)

  Get file descriptor for light sensor thread. Called by watchdog thread in order to be able to send heartbeat check via queue.
- int light_queue_init (void)

  Initialize message queue for light sensor thread.
- void * light_fn (void *thread_args)

  Entry point for light sensor processing thread.

Variables

- static timer_t **timerid**
- struct itimerspec **trigger**
- static i2c_handle_t **i2c_apds9301**
- static float **last_lux_value** = -5
- static mqd_t **light_queue**
- static shared_data_t * **shm**

5.12.1 Detailed Description

Interface to APDS9301 Light Sensor.

==========

<+DETAILED+>

Author

Roberto Baquerizo (baquerrj), roba8460@colorado.edu

5.12.2 Function Documentation

5.12.2.1 apds9301_clear_interrupt()

```
int apds9301_clear_interrupt (
            void  )
```

Clears any pending interrupt for APDS9301 by writing a 1 to the CLEAR bit of the Command Register.

========== Function↩
: apds9301_clear_interrupt

Parameters

void	**see i2c_set()**

```
00236 {
00237    uint8_t clear = APDS9301_REG_CMD | CMD_CLEAR_INTR;
00238
00239    int retVal = i2c_set( APDS9301_ADDRESS, clear );
00240
00241    return retVal;
00242 }
```

5.12.2.2 apds9301_get_lux()

```
int apds9301_get_lux (
              float * lux )
```

Read ADC Registers and calculate lux in lumen using equations from APDS9301 datasheet.

Read ADC Registers and calculate lux in lumen.

========== Function↩
: apds9301_get_lux

Parameters

**lux*	- pointer to location to write decoded lux to **EXIT_CLEAN if successful, otherwise EXIT_ERROR**

```
00413 {
00414    float ratio = 0;
00415    uint16_t data0 = 0;
00416    uint16_t data1 = 0;
00417
00418    int retVal = apds9301_read_data0( &data0 );
00419    if( EXIT_CLEAN != retVal )
00420    {
00421       return EXIT_ERROR;
00422    }
00423
00424    retVal = apds9301_read_data1( &data1 );
00425    if( EXIT_CLEAN != retVal )
00426    {
00427       return EXIT_ERROR;
00428    }
00429
00430    if( 0 == data0 )
00431    {
00432       ratio = 0.0;
00433    }
00434    else
00435    {
00436       ratio = (float)data1 / (float)data0;
00437    }
00438
00439    if( (0 < ratio) && (0.50 >= ratio) )
```

```
00440    {
00441       *lux = 0.0304*data0 - 0.062*data0*(pow(ratio, 1.4));
00442     }
00443     else if( (0.50 < ratio) && (0.61 >= ratio) )
00444     {
00445         *lux = 0.0224*data0 - 0.031*data1;
00446     }
00447     else if( (0.61 < ratio) && (0.80 >= ratio) )
00448     {
00449         *lux = 0.0128*data0 - 0.0153*data1;
00450     }
00451     else if( (0.80 < ratio) && (1.30 >= ratio) )
00452     {
00453         *lux = 0.00146*data0 - 0.00112*data1;
00454     }
00455     else if( 1.30 < ratio )
00456     {
00457         *lux = 0;
00458    }
00459
00460    return EXIT_CLEAN;
00461 }
```

5.12.2.3 apds9301_power()

```
int apds9301_power (
             uint16_t on )
```

power on (or off) APDS9301 as set by paramater

========== Function↩
: apds9301_power

Parameters

on	- specifies if sensor is to be powered on or off **see i2c_write_byte()**

```
00538 {
00539    int retVal = 0;
00540    if( on )
00541    {
00542       /* power on */
00543       retVal = i2c_write_byte( APDS9301_ADDRESS, APDS9301_REG_CNTRL,
      POWER_ON );
00544    }
00545    else
00546    {
00547       /* power off */
00548       retVal = i2c_write_byte( APDS9301_ADDRESS, APDS9301_REG_CNTRL, POWER_OFF );
00549    }
00550
00551    return retVal;
00552 }
```

5.12.2.4 apds9301_read_control()

```
int apds9301_read_control (
             uint8_t * data )
```

Read contents of Control Register.

========== Function↩
: apds9301_read_control

Parameters

**data*	- where to store contents **see i2c_read()**

```
00322 {
00323     int retVal = i2c_read( APDS9301_ADDRESS, APDS9301_REG_CNTRL, data, sizeof( *data ) );
00324     return retVal;
00325 }
```

5.12.2.5 apds9301_read_data0()

```
int apds9301_read_data0 (
            uint16_t * data )
```

Read ADC register for channel 0.

========== function↩
: apds9301_read_data0

Parameters

**data*	- pointer to location to write decoded value to **EXIT_CLEAN if successful, otherwise exit_error**

```
00473 {
00474     uint8_t low  = 0;
00475     uint8_t high = 0;
00476     int retVal = i2c_read( APDS9301_ADDRESS, APDS9301_REG_DLOW_0, &low, 0 );
00477
00478     if( EXIT_CLEAN != retVal )
00479     {
00480       return EXIT_ERROR;
00481     }
00482
00483     retVal = i2c_read( APDS9301_ADDRESS, APDS9301_REG_DHIGH_0, &high, 0 );
00484
00485     if( EXIT_CLEAN == retVal )
00486     {
00487        *data = ( low | (high << 8 ) );
00488     }
00489     else
00490     {
00491        return EXIT_ERROR;
00492     }
00493     return EXIT_CLEAN;
00494 }
```

Here is the caller graph for this function:

5.12.2.6 apds9301_read_data1()

```
int apds9301_read_data1 (
            uint16_t * data )
```

Read ADC register for channel 1.

========== function↩
: apds9301_read_data1

Parameters

**data*	- pointer to location to write decoded value to **EXIT_CLEAN if successful, otherwise exit_error**

```
00505 {
00506    uint8_t low  = 0;
00507    uint8_t high = 0;
00508    int retVal = i2c_read( APDS9301_ADDRESS, APDS9301_REG_DLOW_1, &low, 0 );
00509
00510    if( EXIT_CLEAN != retVal )
00511    {
00512      return EXIT_ERROR;
00513    }
00514
00515    retVal = i2c_read( APDS9301_ADDRESS, APDS9301_REG_DHIGH_1, &high, 0 );
00516
00517    if( EXIT_CLEAN == retVal )
00518    {
00519       *data = ( low | (high << 8 ) );
00520    }
00521    else
00522    {
00523       return EXIT_ERROR;
00524    }
00525    return EXIT_CLEAN;
00526 }
```

5.12.2.7 apds9301_read_id()

```
int apds9301_read_id (
            uint8_t * id )
```

Read APDS9301 Identification Register.

========== Function↩
: apds9301_read_id

Parameters

**id*	- where to write ID from register **see i2c_read()**

```
00397 {
00398     int retVal = i2c_read( APDS9301_ADDRESS, APDS9301_REG_ID, id, sizeof( *id ) );
00399     return retVal;
00400 }
```

5.12.2.8 apds9301_read_threshold_high()

```
int apds9301_read_threshold_high (
            uint16_t * threshold )
```

Read value from high threshold register.

========== Function↩
: apds9301_write_threshold_high

Parameters

∗threshold	- where to write value read **see i2c_write()**

```
00382 {
00383     int retVal = i2c_read( APDS9301_ADDRESS, APDS9301_REG_TH_HL, (uint8_t*)threshold, sizeof( *
      threshold ) );
00384     return retVal;
00385 }
```

5.12.2.9 apds9301_read_threshold_low()

```
int apds9301_read_threshold_low (
            uint16_t * threshold )
```

Read value from low threshold register.

========== Function↩
: apds9301_write_threshold_low

Parameters

∗threshold	- where to write value read **see i2c_write()**

```
00352 {
00353     int retVal = i2c_read( APDS9301_ADDRESS, APDS9301_REG_TH_LL, (uint8_t*)threshold, sizeof( *
```

```
      threshold ) );
00354    return retVal;
00355 }
```

5.12.2.10 apds9301_set_config()

```
int apds9301_set_config (
              void  )
```

Set configuration of light sensor. For the APDS9301, the configuration is spread out across the: Timing Register, Interrupt Control Register, and Control Register. So, I have to write to all of these to set the config.

========== Function↩
: apds9301_set_config

Parameters

void	

EXIT_CLEAN if successful, otherwise see i2c_write()

```
00166 {
00167    int retVal = apds9301_set_gain( DEFAULT_GAIN );
00168    if( retVal )
00169    {
00170       return retVal;
00171    }
00172    else
00173    {
00174       retVal = apds9301_set_interrupt( DEFAULT_INTERRUPT );
00175       if( retVal )
00176       {
00177          return retVal;
00178       }
00179       else
00180       {
00181          retVal = apds9301_set_integration( DEFAULT_INTEGRATION_TIME );
00182          if( retVal )
00183          {
00184             return retVal;
00185          }
00186       }
00187    }
00188    return EXIT_CLEAN;
00189 }
```

5.12.2.11 apds9301_set_gain()

```
int apds9301_set_gain (
              uint8_t gain )
```

Sets gain for APDS9301 by setting or clearing the GAIN bit of the Timing Register.

========== Function↩
: apds9301_set_gain

Parameters

gain	- set if we want high gain see i2c_write_byte()

```
00289 {
00290    uint8_t data;
00291    int retVal = i2c_read( APDS9301_ADDRESS, APDS9301_REG_TIME, &data, sizeof( data ) );
00292    if( retVal )
00293    {
00294       return EXIT_ERROR;
00295    }
00296
00297    /* if gain != 0, high gain */
00298    if( gain )
00299    {
00300       data |= (1<<4);
00301    }
00302    else
00303    {
00304       data &= ~(1<<4);
00305    }
00306
00307    retVal = i2c_write_byte( APDS9301_ADDRESS, APDS9301_REG_TIME, data );
00308
00309    return retVal;
00310 }
```

Here is the caller graph for this function:

5.12.2.12 apds9301_set_integration()

```
int apds9301_set_integration (
             uint8_t val )
```

Sets the integration time for APDS9301 by writing a value to bits INTEG of the Timing Register.

========== Function↩
: apds9301_set_integration

Parameters

val	- value to write to timing register see i2c_write_byte() - if val is not an allowed value, EXIT_ERROR

```
00202 {
00203    if( 3 < val )
00204    {
00205       /* invalid value */
00206       return EXIT_ERROR;
00207    }
00208    uint8_t data;
00209    int retVal = i2c_read( APDS9301_ADDRESS, APDS9301_REG_TIME, &data, sizeof( data ) );
00210
00211    if( retVal )
00212    {
00213       return EXIT_ERROR;
00214    }
00215
```

```
00216     data &= ~(0b11);  /* clears lower 2 bits of TIMING REG */
00217     data |= val;
00218
00219     retVal = i2c_write_byte( APDS9301_ADDRESS, APDS9301_REG_TIME, data );
00220
00221     return retVal;
00222 }
```

5.12.2.13 apds9301_set_interrupt()

```
int apds9301_set_interrupt (
              uint8_t enable )
```

Enables or disables interrupts for APDS9301 by setting or clearing the INTR bits of the Interrupt Control Register.

========== Function↩
: apds9301_set_interrupt

Parameters

enable	- set if we want to enable interrupts see i2c_write_byte()

```
00256 {
00257     uint8_t data;
00258     int retVal = i2c_read( APDS9301_ADDRESS, APDS9301_REG_INT_CNTRL, &data, sizeof( data ) );
00259     if( retVal )
00260     {
00261        return EXIT_ERROR;
00262     }
00263
00264     if( enable )
00265     {
00266        data |= (1<<4);
00267     }
00268     else
00269     {
00270        data &= ~(1<<4);
00271     }
00272
00273     retVal = i2c_write_byte( APDS9301_ADDRESS, APDS9301_REG_INT_CNTRL, data );
00274
00275     return retVal;
00276 }
```

Here is the caller graph for this function:

5.12.2.14 apds9301_write_threshold_high()

```
int apds9301_write_threshold_high (
              uint16_t threshold )
```

Write value to high threshold register.

========== Function↩
: apds9301_write_threshold_high

Parameters

threshold	- value to write see i2c_write()

```
00367 {
00368     int retVal = i2c_write( APDS9301_ADDRESS, APDS9301_REG_TH_HL, threshold );
00369     return retVal;
00370 }
```

5.12.2.15 apds9301_write_threshold_low()

```
int apds9301_write_threshold_low (
              uint16_t threshold )
```

Write value to low threshold register.

========== Function↩
: apds9301_write_threshold_low

Parameters

threshold	- value to write see i2c_write()

```
00337 {
00338     int retVal = i2c_write( APDS9301_ADDRESS, APDS9301_REG_TH_LL, threshold );
00339     return retVal;
00340 }
```

5.12.2.16 cycle()

```
static void cycle (
              void  )  [static]
```

Cycle function for light sensor thread We wait in this while loop checking for requests from watchdog for health status.

========== Function↩
: cycle

Parameters

void	**void**

```
00566 {
00567    int retVal = 0;
00568    msg_t request = {0};
00569    msg_t response = {0};
00570    while( 1 )
00571    {
00572       memset( &request, 0, sizeof( request ) );
00573       retVal = mq_receive( light_queue, (char*)&request, sizeof( request ), NULL );
00574       if( 0 > retVal )
00575       {
00576          int errnum = errno;
00577          fprintf( stderr, "Encountered error receiving from message queue %s: (%s)\n",
00578                   LIGHT_QUEUE_NAME, strerror( errnum ) );
00579          continue;
00580       }
00581       switch( request.id )
00582       {
00583          case REQUEST_STATUS:
00584             sem_wait(&shm->w_sem);
00585             print_header(shm->header);
00586             sprintf( shm->buffer, "(Light) I am alive!\n" );
00587             sem_post(&shm->r_sem);
00588             fprintf( stdout, "(Light) I am alive!\n" );
00589             response.id = request.id;
00590             sprintf( response.info, "(Light) I am alive!\n" );
00591             retVal = mq_send( request.src, (const char*)&response, sizeof( response ), 0 );
00592
00593             pthread_mutex_lock( &alive_mutex );
00594             threads_status[THREAD_LIGHT]--;
00595             pthread_mutex_unlock( &alive_mutex );
00596             break;
00597          default:
00598             break;
00599       }
00600    }
00601    return;
00602 }
```

5.12.2.17 get_light_queue()

```
mqd_t get_light_queue (
              void  )
```

Get file descriptor for light sensor thread. Called by watchdog thread in order to be able to send heartbeat check via queue.

========== Function↩
: get_light_queue

Parameters

void	**temp_queue - file descriptor for light sensor thread message queue**

```
00615 {
```

```
00616     return light_queue;
00617 }
```

5.12.2.18 get_lux()

```
float get_lux (
              void  )
```

Returns last lux reading.

========== Function↩
: get_lux

Parameters

void	**last_lux_value - last lux reading we have**

```
00130 {
00131     return last_lux_value;
00132 }
```

5.12.2.19 is_dark()

```
int is_dark (
              void  )
```

Returns int speciyfing if it is night or day.

========== Function↩
: is_dark

Parameters

void	**night - 0 if it is day, 1 if night, i.e. below DARK_THRESHOLD**

```
00145 {
00146     int dark = 0;
00147     if( DARK_THRESHOLD > last_lux_value )
00148     {
00149        dark = 1;
00150     }
00151     return dark;
00152 }
```

5.12.2.20 light_fn()

```
void* light_fn (
            void * thread_args )
```

Entry point for light sensor processing thread.

========== Function↩
: light_fn

Parameters

thread_args	- void ptr to arguments used to initialize thread

Returns

NULL - We don't really exit from this function,

since the exit point is thread_exit()

```
00663 {
00664    /* Get time that thread was spawned */
00665    struct timespec time;
00666    clock_gettime(CLOCK_REALTIME, &time);
00667    shm = get_shared_memory();
00668
00669    /* Write initial state to shared memory */
00670    sem_wait(&shm->w_sem);
00671    print_header(shm->header);
00672    sprintf( shm->buffer, "Hello World! Start Time: %ld.%ld secs\n",
00673             time.tv_sec, time.tv_nsec );
00674    /* Signal to logger that shared memory has been updated */
00675    sem_post(&shm->r_sem);
00676
00677    signal(SIGUSR1, sig_handler);
00678    signal(SIGUSR2, sig_handler);
00679
00680    light_queue = light_queue_init();
00681    if( 0 > light_queue )
00682    {
00683       thread_exit( EXIT_INIT );
00684    }
00685
00686    int retVal = i2c_init( &i2c_apds9301 );
00687    if( EXIT_INIT == retVal )
00688    {
00689       sem_wait(&shm->w_sem);
00690       print_header(shm->header);
00691       sprintf( shm->buffer, "ERROR: Failed to initialize I2C for light sensor!\n" );
00692       sem_post(&shm->r_sem);
00693       thread_exit( EXIT_INIT );
00694    }
00695    retVal = apds9301_power( POWER_ON );
00696    if( retVal )
00697    {
00698       sem_wait(&shm->w_sem);
00699       print_header(shm->header);
00700       sprintf( shm->buffer, "ERROR: Failed to power on light sensor!\n" );
00701       sem_post(&shm->r_sem);
00702       thread_exit( EXIT_INIT );
00703    }
00704
00705    timer_setup( &timerid, &timer_handler );
00706
00707    timer_start( &timerid, 5000000 );
00708    cycle();
00709
00710    thread_exit( 0 );
00711    return NULL;
00712 }
```

5.12.2.21 light_queue_init()

```
int light_queue_init (
            void  )
```

Initialize message queue for light sensor thread.

========== Function↩
: light_queue_init

Parameters

void	**msg_q - file descriptor for initialized message queue**

```
00629 {
00630    /* unlink first in case we hadn't shut down cleanly last time */
00631    mq_unlink( LIGHT_QUEUE_NAME );
00632
00633    struct mq_attr attr;
00634    attr.mq_flags = 0;
00635    attr.mq_maxmsg = MAX_MESSAGES;
00636    attr.mq_msgsize = sizeof( msg_t );
00637    attr.mq_curmsgs = 0;
00638
00639    int msg_q = mq_open( LIGHT_QUEUE_NAME, O_CREAT | O_RDWR, 0666, &attr );
00640    if( 0 > msg_q )
00641    {
00642       int errnum = errno;
00643       sem_wait(&shm->w_sem);
00644       print_header(shm->header);
00645       sprintf( shm->buffer, "Encountered error creating message queue %s: (%s)\n",
00646                LIGHT_QUEUE_NAME, strerror( errnum ) );
00647       sem_post(&shm->r_sem);
00648    }
00649    return msg_q;
00650 }
```

5.12.2.22 sig_handler()

```
static void sig_handler (
            int signo )  [static]
```

Signal handler for light sensor thread. On normal operation, we should be receving SIGUSR1/2 signals from watchdog when prompted to exit. So, we close the message queue and timer this thread owns.

========== Function↩
: sig_handler

Parameters

signo	- enum with signal number of signal being handled **void**

```
00051 {
00052    if( signo == SIGUSR1 )
00053    {
00054       printf("Received SIGUSR1! Exiting...\n");
00055       mq_close( light_queue );
00056       timer_delete( timerid );
00057       apds9301_power( POWER_OFF );
00058       i2c_stop( &i2c_apds9301 );
00059       thread_exit( signo );
00060    }
00061    else if( signo == SIGUSR2 )
00062    {
00063       printf("Received SIGUSR2! Exiting...\n");
00064       mq_close( light_queue );
00065       timer_delete( timerid );
00066       apds9301_power( POWER_OFF );
00067       i2c_stop( &i2c_apds9301 );
00068       thread_exit( signo );
00069    }
00070    return;
00071 }
```

Here is the caller graph for this function:

5.12.2.23 timer_handler()

```
static void timer_handler (
             union sigval sig )  [static]
```

Timer handler function for light sensor thread When woken up by the timer, get lux reading and write state to shared memory.

========== Function↩ : timer_handler

Parameters

sig	**void**

```
00084 {
00085    static int i = 0;
00086    led_toggle( LED1_BRIGHTNESS );
00087    sem_wait(&shm->w_sem);
00088
00089    print_header(shm->header);
00090    float lux = -5;
00091    int retVal = apds9301_get_lux( &lux );
00092
00093    i++;
00094    if( retVal )
00095    {
00096       /* save new lux value */
00097       last_lux_value = lux;
```

```
00098       if( DARK_THRESHOLD > lux )
00099       {
00100          sprintf( shm->buffer, "cycle[%d]: State: %s, Lux: %0.5f\n",
00101                   i, "NIGHT", lux );
00102       }
00103       else
00104       {
00105          sprintf( shm->buffer, "cycle[%d]: State: %s, Lux: %0.5f\n",
00106                   i, "DAY", lux );
00107       }
00108    }
00109    else
00110    {
00111       sprintf( shm->buffer, "cycle[%d]: could not get light reading!\n", i );
00112    }
00113
00114    sem_post(&shm->r_sem);
00115    led_toggle( LED1_BRIGHTNESS );
00116    return;
00117 }
```

5.13 /home/baquerrj/boulder/ecen5013/project_1/src/logger.c File Reference

Takes care of logging for other threads.

```
#include "led.h"
#include "logger.h"
#include <errno.h>
#include <string.h>
#include <time.h>
#include <stdlib.h>
#include <signal.h>
#include <fcntl.h>
#include <sys/syscall.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/stat.h>
```

Include dependency graph for logger.c:

Functions

- static void sig_handler (int signo)

  Signal handler for logger thread. On normal operation, we should be receving SIGUSR1/2 signals from watchdog when prompted to exit. So, we close the message queue and timer this thread owns.

- void * logger_fn (void *arg)

  Entry point for logger thread.

Variables

- struct itimerspec **trigger**
- static FILE * **log**
- static shared_data_t * **shm**

5.13.1 Detailed Description

Takes care of logging for other threads.

==========

This logger works in background to log the state of other threads to a common file. It is responsible for reading the shared memory segment written to by the sensor threads. It sleeps waiting for a semaphore to be posted by another thread signaling that new data has been written to shared memory and that it should read it.

Author

Roberto Baquerizo (baquerrj), roba8460@colorado.edu

5.13.2 Function Documentation

5.13.2.1 logger_fn()

```
void* logger_fn (
            void * arg )
```

Entry point for logger thread.

========== Function↩
: logger_fn

Parameters

thread_args	- void ptr to arguments used to initialize thread

Returns

NULL - We don't really exit from this function,

since the exit point is thread_exit()

```
00079 {
00080    struct timespec time;
00081    clock_gettime(CLOCK_REALTIME, &time);
00082    static int failure = 1;
00083
00084    signal(SIGUSR1, sig_handler);
00085    signal(SIGUSR2, sig_handler);
00086
00087    /* Initialize thread */
00088    if( NULL == arg )
00089    {
00090       fprintf( stderr, "Thread requires name of log file!\n" );
00091       pthread_exit(&failure);
00092    }
00093
00094    log = (FILE *)arg;
00095    if( NULL == log )
00096    {
00097       perror( "Encountered error opening log file" );
```

```
00098       pthread_exit(&failure);
00099    }
00100
00101    shm = get_shared_memory();
00102    if( NULL == shm )
00103    {
00104       int errnum = errno;
00105       fprintf( stderr, "Encountered error memory mapping shared memory: %s\n",
00106                strerror( errnum ) );
00107    }
00108
00109
00110    shared_data_t *buf = malloc( sizeof( shared_data_t ) );
00111    if( NULL == buf )
00112    {
00113       int errnum = errno;
00114       fprintf( stderr, "Encountered error allocating memory for local buffer %s\n",
00115                strerror( errnum ) );
00116    }
00117
00118    while( 1 )
00119    {
00120       sem_wait(&shm->r_sem);
00121       memcpy( buf, shm, sizeof(*shm) );
00122
00123       fprintf( log, "%s\n%s", buf->header, buf->buffer );
00124       fflush( log );
00125
00126       led_toggle( LED3_BRIGHTNESS );
00127       sem_post(&shm->w_sem);
00128    }
00129
00130    return NULL;
00131 }
```

5.13.2.2 sig_handler()

```
static void sig_handler (
             int signo )  [static]
```

Signal handler for logger thread. On normal operation, we should be receving SIGUSR1/2 signals from watchdog when prompted to exit. So, we close the message queue and timer this thread owns.

========== Function↩
: sig_handler

Parameters

signo	- enum with signal number of signal being handled **void**

```
00053 {
00054    if( signo == SIGUSR1 )
00055    {
00056       printf("Received SIGUSR1! Exiting...\n");
00057       thread_exit( signo );
00058    }
00059    else if( signo == SIGUSR2 )
00060    {
00061       printf("Received SIGUSR2! Exiting...\n");
00062       thread_exit( signo );
00063    }
00064    return;
00065 }
```

Here is the caller graph for this function:

5.14 /home/baquerrj/boulder/ecen5013/project_1/src/main.c File Reference

<+DETAILED+>

```
#include "temperature.h"
#include "light.h"
#include "logger.h"
#include "common.h"
#include "watchdog.h"
#include "socket.h"
#include "led.h"
#include <fcntl.h>
#include <signal.h>
#include <errno.h>
#include <stdlib.h>
#include <string.h>
#include <sys/syscall.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/stat.h>
```

Include dependency graph for main.c:

Functions

- static void signal_handler (int signo)
- void turn_off_leds (void)
- int main (int argc, char *argv[])

Variables

- static pthread_t temp_thread
- static pthread_t **light_thread**
- static pthread_t **logger_thread**
- static pthread_t **socket_thread**
- static pthread_t **watchdog_thread**
- static shared_data_t * **shm**

5.14.1 Detailed Description

<+DETAILED+>

==========

Author

Roberto Baquerizo (baquerrj), roba8460@colorado.edu

5.14.2 Function Documentation

5.14.2.1 main()

```
int main (
              int argc,
              char * argv[] )
```

== Function↩
: main

Parameters

<+NAME+>	<+DESCRIPTION+>

Returns

<+DESCRIPTION+>

<+DETAILED+>

```
00099 {
00100    signal( SIGINT, signal_handler );
00101    static file_t *log;
00102    printf( "Number of arguments %d\n", argc );
00103    if( argc > 1 )
00104    {
00105       log = malloc( sizeof( file_t ) );
00106       log->fid = fopen( argv[1], "w" );
00107       log->name = argv[1];
00108       printf( "Opened file %s\n", argv[1] );
00109    }
00110    else
00111    {
00112       fprintf( stderr, "Name of log file required!\n" );
00113       return 1;
00114    }
00115
00116    /* Initialize Shared Memory */
00117    shm = get_shared_memory();
00118    if( 0  > sems_init( shm ) )
00119    {
00120       fprintf( stderr, "Encountered error initializing semaphores!\n" );
00121       return 1;
00122    }
00123
00124    struct timespec time;
00125    clock_gettime(CLOCK_REALTIME, &time);
00126
00127    print_header( NULL );
00128    fprintf( stdout, "Starting Threads! Start Time: %ld.%ld secs\n",
00129             time.tv_sec, time.tv_nsec );
00130
00131    struct thread_id_s* threads = malloc( sizeof( struct thread_id_s ) );
00132
00133    led_on( LED2_BRIGHTNESS );
00134
00135    set_trigger( LED2_TRIGGER, "timer" );
00136    set_delay( LED2_DELAYON, 50 );
00137    /* Attempting to spawn child threads */
00138    pthread_create( &logger_thread, NULL, logger_fn, (void*)log->fid );
00139    pthread_create( &temp_thread, NULL, temperature_fn, NULL );
00140    pthread_create( &light_thread, NULL, light_fn, NULL );
00141    pthread_create( &socket_thread, NULL , socket_fn, NULL );
00142
00143    threads->temp_thread = temp_thread;
00144    threads->logger_thread = logger_thread;
00145    threads->light_thread = light_thread;
00146    threads->socket_thread = socket_thread;
00147
00148    pthread_create( &watchdog_thread, NULL, watchdog_fn, (void*)threads );
00149
00150    pthread_join( watchdog_thread, NULL );
00151
00152    clock_gettime(CLOCK_REALTIME, &time);
00153
00154
00155    print_header( NULL );
00156    fprintf( stdout, "All threads exited! Main thread exiting... " );
00157    fprintf( stdout, "End Time: %ld.%ld secs\n",
00158             time.tv_sec, time.tv_nsec );
00159
00160    free( log );
00161    free( threads );
00162    munmap( shm, sizeof( shared_data_t ) );
00163    shm_unlink( SHM_SEGMENT_NAME );
00164    turn_off_leds();
00165    return 0;
00166 }
```

5.14.2.2 signal_handler()

```
static void signal_handler (
             int signo )  [static]
```

========== Function↩
: signal_handler

Parameters

<+NAME+>	<+DESCRIPTION+>

Returns

<+DESCRIPTION+>

<+DETAILED+>

```
00060 {
00061    switch( signo )
00062    {
00063       case SIGINT:
00064          fprintf( stderr, "Master caught SIGINT!\n" );
00065          pthread_kill( watchdog_thread, SIGUSR2 );
00066    }
00067 }
```

Here is the caller graph for this function:

5.14.2.3 turn_off_leds()

```
void turn_off_leds (
             void  )
```

========== Function↩
: turn_off_leds

Parameters

<+NAME+>	<+DESCRIPTION+>

Returns

<+DESCRIPTION+>

<+DETAILED+>

```
00080 {
00081    led_off( LED0_BRIGHTNESS );
00082    led_off( LED1_BRIGHTNESS );
00083    led_off( LED2_BRIGHTNESS );
00084    led_off( LED3_BRIGHTNESS );
00085    return;
00086 }
```

5.14.3 Variable Documentation

5.14.3.1 temp_thread

```
pthread_t temp_thread  [static]
```

/sys includes

5.15 /home/baquerrj/boulder/ecen5013/project_1/src/socket.c File Reference

Remote Socket task capable of requesting sensor readings from temperature and light sensor threads.

```
#include "socket.h"
#include "common.h"
#include "light.h"
#include "temperature.h"
#include <string.h>
#include <stdlib.h>
#include <errno.h>
#include <time.h>
#include <signal.h>
#include <unistd.h>
#include <semaphore.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
```

Include dependency graph for socket.c:

Macros

- #define **PORT** 8080

Functions

- msg_t process_request (msg_t *request)

  Process a request from remote client.
- static void cycle (int server)

  Cycle function for remote socket task. Spins in this infinite while-loop checking for new connections to make. When it receives a new connection, it starts processing requests from the client.
- int socket_init (void)

  Initliaze the server socket.
- void * socket_fn (void *thread_args)

  Entry point for remote socket thread.

Variables

- static shared_data_t * **shm**

5.15.1 Detailed Description

Remote Socket task capable of requesting sensor readings from temperature and light sensor threads.

==========

Author

Roberto Baquerizo (baquerrj), roba8460@colorado.edu

5.15.2 Function Documentation

5.15.2.1 cycle()

```
static void cycle (
            int server )  [static]
```

Cycle function for remote socket task. Spins in this infinite while-loop checking for new connections to make. When it receives a new connection, it starts processing requests from the client.

========== Function↩
: cycle

Parameters

server	- server socket file descriptor **void**

```
00136 {
00137    const char *client_info;
00138    int client = 1;
00139    char ip[20] = {0};
00140    struct sockaddr_in addr;
00141    int addrlen = sizeof( addr );
00142
00143    /* Buffer to copy status to shared memory for logger */
00144    //char *status;
00145    while( 1 )
00146    {
00147       int kill = 0;
00148       client = accept( server, (struct sockaddr*)&addr, (socklen_t*)&addrlen );
00149       if( 0 > client )
00150       {
00151          int errnum = errno;
00152
00153          fprintf( stderr, "Could not accept new connection (%s)\n",
00154                   strerror( errnum ) );
00155          continue;
00156       }
00157
00158       client_info = inet_ntop( AF_INET, &addr.sin_addr, ip, sizeof( ip ) );
00159       fprintf( stdout, "New connection accepted: %s\n", client_info );
00160
00161       while( 1 )
00162       {
00163          msg_t request = {0};
00164          msg_t response = {0};
```

```
00165          int bytes = 0;
00166          while( ( -1 != bytes ) && ( sizeof( request ) > bytes ) )
00167          {
00168             bytes = recv( client, ((char*)&request + bytes), sizeof( request ), 0 );
00169          }
00170
00171          response = process_request( &request );
00172
00173          if( REQUEST_CLOSE == response.id )
00174          {
00175             break;
00176          }
00177          if( REQUEST_KILL == response.id )
00178          {
00179             kill = 1;
00180             break;
00181          }
00182
00183          /* Send out response to client */
00184          bytes = send( client, (char*)&response, sizeof( response ), 0 );
00185          if( sizeof( response ) > bytes )
00186          {
00187             if( -1 == bytes )
00188             {
00189                int errnum = errno;
00190                fprintf( stderr, "Encountered error sending data to client: (%s)\n",
00191                         strerror( errnum ) );
00192                break;
00193             }
00194             else
00195             {
00196                fprintf( stderr, "Could not transmit all data: %u out of %u bytes sent.\n",
00197                         bytes, sizeof( response ) );
00198                break;
00199             }
00200          }
00201          fprintf( stdout, "%u out of %u bytes sent.\n",
00202                   bytes, sizeof( response ) );
00203       }
00204
00205       client_info = inet_ntop( AF_INET, &addr.sin_addr, ip, sizeof( ip ) );
00206       close( client );
00207       fprintf( stdout, "Client connection closed: %s\n", client_info );
00208
00209       if( 1 == kill )
00210       {
00211          close( server );
00212          fprintf( stdout, "Closed server.\n" );
00213          break;
00214       }
00215    }
00216    return;
00217 }
```

5.15.2.2 process_request()

```
msg_t process_request (
            msg_t * request )
```

Process a request from remote client.

========== Function↩
: process_request

Parameters

**request*	- request from client **response - our response**

```
00050 {
00051    msg_t response = {0};
00052    switch( request->id )
00053    {
00054       case REQUEST_LUX:
00055          response.id = request->id;
00056          response.data.data = get_lux();
00057          sem_wait(&shm->w_sem);
00058          print_header(shm->header);
00059          sprintf( shm->buffer, "Request Lux: %.5f\n",
00060                   response.data.data );
00061          sem_post(&shm->r_sem);
00062          break;
00063       case REQUEST_DARK:
00064          response.id = request->id;
00065          response.data.night = is_dark();
00066          sem_wait(&shm->w_sem);
00067          print_header(shm->header);
00068          sprintf( shm->buffer, "Request Day or Night: %s\n",
00069                   (response.data.night == 0) ? "day" : "night");
00070          sem_post(&shm->r_sem);
00071          break;
00072       case REQUEST_TEMP:
00073          response.id = request->id;
00074          response.data.data = get_temperature();
00075          sem_wait(&shm->w_sem);
00076          print_header(shm->header);
00077          sprintf( shm->buffer, "Request Temperature: %.5f C\n",
00078                   response.data.data );
00079          sem_post(&shm->r_sem);
00080          break;
00081       case REQUEST_TEMP_K:
00082          response.id = request->id;
00083          response.data.data = get_temperature() + 273.15;
00084          sem_wait(&shm->w_sem);
00085          print_header(shm->header);
00086          sprintf( shm->buffer, "Request Temperature: %.5f K\n",
00087                   response.data.data );
00088          sem_post(&shm->r_sem);
00089          break;
00090        case REQUEST_TEMP_F:
00091          response.id = request->id;
00092          response.data.data = (get_temperature() *1.80) + 32.0;
00093          sem_wait(&shm->w_sem);
00094          print_header(shm->header);
00095          sprintf( shm->buffer, "Request Temperature: %.5f F\n",
00096                   response.data.data );
00097          sem_post(&shm->r_sem);
00098          break;
00099       case REQUEST_CLOSE:
00100          response.id = request->id;
00101          sem_wait(&shm->w_sem);
00102          print_header(shm->header);
00103          sprintf( shm->buffer, "Request Close Connection\n" );
00104          sem_post(&shm->r_sem);
00105          break;
00106       case REQUEST_KILL:
00107          response.id = request->id;
00108          sem_wait(&shm->w_sem);
00109          print_header(shm->header);
00110          sprintf( shm->buffer, "Request Kill Application\n" );
00111          sem_post(&shm->r_sem);
00112          break;
00113       default:
00114          sem_wait(&shm->w_sem);
00115          print_header(shm->header);
00116          sprintf( shm->buffer, "Invalid Request\n" );
00117          sem_post(&shm->r_sem);
00118          break;
00119    }
00120
00121    return response;
00122 }
```

Here is the caller graph for this function:

5.15.2.3 socket_fn()

```
void* socket_fn (
            void * thread_args )
```

Entry point for remote socket thread.

========== Function↩
: socket_fn

Parameters

**thread_args*	- thread arguments (if any)

Returns

NULL - We don't really exit from this function,

since the exit point is thread_exit()

```
00306 {
00307    /* Get time that thread was spawned */
00308    struct timespec time;
00309    clock_gettime(CLOCK_REALTIME, &time);
00310
00311    /* Get pointer to shared memory struct */
00312    shm = get_shared_memory();
00313
00314    int server = socket_init();
00315    if( -1 == server )
00316    {
00317       fprintf( stderr, "Failed to set up server!\n" );
00318       thread_exit( EXIT_INIT );
00319    }
00320
00321    /* Write initial state to shared memory */
00322    sem_wait(&shm->w_sem);
00323    print_header(shm->header);
00324    sprintf( shm->buffer, "Hello World! Start Time: %ld.%ld secs\n",
00325             time.tv_sec, time.tv_nsec );
00326    /* Signal to logger that shared memory has been updated */
00327    sem_post(&shm->r_sem);
00328
00329    cycle( server );
00330
00331    thread_exit( EXIT_CLEAN );
00332    return NULL;
00333 }
```

5.15.2.4 socket_init()

```
int socket_init (
             void  )
```

Initliaze the server socket.

Cycle function for remote socket task. Spins in this infinite while-loop checking for new connections to make. When it receives a new connection, it starts processing requests from the client.

========== Function↩
: socket_init

Parameters

void	**server - file descriptor for newly created socket for server**

```
00230 {
00231    int retVal = 0;
00232    int opt = 1;
00233    struct sockaddr_in addr;
00234
00235    int server = socket( AF_INET, SOCK_STREAM, 0 );
00236    if( 0 == server )
00237    {
00238       int errnum = errno;
00239       fprintf( stderr, "Encountered error creating new socket (%s)\n",
00240                strerror( errnum ) );
00241       return -1;
00242    }
00243
00244    retVal = setsockopt( server, SOL_SOCKET, SO_REUSEPORT | SO_REUSEADDR, &(opt), sizeof(opt) );
00245    if( 0 != retVal )
00246    {
00247       int errnum = errno;
00248       sem_wait(&shm->w_sem);
00249       print_header(shm->header);
00250       sprintf( shm->buffer, "Encountered error setting socket options (%s)\n",
00251                strerror( errnum ) );
00252       sem_post(&shm->r_sem);
00253       return -1;
00254    }
00255
00256    addr.sin_family = AF_INET;
00257    addr.sin_addr.s_addr = INADDR_ANY;
00258    addr.sin_port = htons( PORT );
00259
00260    /* Attempt to bind socket to address */
00261    retVal = bind( server, (struct sockaddr*)&addr, sizeof( addr ) );
00262    if( 0 > retVal )
00263    {
00264       int errnum = errno;
00265       sem_wait(&shm->w_sem);
00266       print_header(shm->header);
00267       sprintf( shm->buffer, "Encountered error binding the new socket (%s)\n",
00268                strerror( errnum ) );
00269       sem_post(&shm->r_sem);
00270       return -1;
00271    }
00272
00273    /* Try to listen */
00274    retVal = listen( server, 10 );
00275    if( 0 > retVal )
00276    {
00277       int errnum = errno;
00278       sem_wait(&shm->w_sem);
00279       print_header(shm->header);
00280       sprintf( shm->buffer, "Encountered error listening with new socket (%s)\n",
00281                strerror( errnum ) );
00282       sem_post(&shm->r_sem);
00283       return -1;
00284    }
00285
00286    sem_wait(&shm->w_sem);
00287    print_header(shm->header);
00288    sprintf( shm->buffer, "Created new socket [%d]!\n", server );
00289    sem_post(&shm->r_sem);
00290
00291    return server;
00292 }
```

Here is the caller graph for this function:

5.16 /home/baquerrj/boulder/ecen5013/project_1/src/temperature.c File Reference

Source file implementing temperature.h.

```
#include "watchdog.h"
#include "temperature.h"
#include "led.h"
#include <errno.h>
#include <time.h>
#include <string.h>
```

Include dependency graph for temperature.c:

Functions

- float get_temperature (void)

  Returns last temperature reading we have.
- int tmp102_write_config (tmp102_config_t *config_reg)

  Write configuration register of TMP102 sensor.
- int tmp102_get_temp (float *temperature)

  Read temperature registers fo TMP102 sensor and decode temperature value.
- int tmp102_write_thigh (float thigh)

  Write value thigh (in celsius) to Thigh register for TMP102 sensor.
- int tmp102_write_tlow (float tlow)

  Write value tlow (in celsius) to Tlow register for TMP102 sensor.
- int tmp102_read_thigh (float *thigh)

  Read value of THigh register of TMP102 sensor and store value (in celsius) in thigh.
- int tmp102_read_tlow (float *tlow)

  Read value of TLow register of TMP102 sensor and store value (in celsius) in tlow.
- static void sig_handler (int signo)

  Signal handler for temperature sensor thread. On normal operation, we should be receving SIGUSR1/2 signals from watchdog when prompted to exit. So, we close the message queue and timer this thread owns.
- static void timer_handler (union sigval sig)

  Timer handler function for temperature sensor thread When woken up by the timer, get temperature and write state to shared memory.
- static void cycle (void)

  Cycle function for temperature sensor thread We wait in this while loop checking for requests from watchdog for health status.
- mqd_t get_temperature_queue (void)

  Get file descriptor for temperature sensor thread. Called by watchdog thread in order to be able to send heartbeat check via queue.
- int temp_queue_init (void)

  Initialize message queue for temperature sensor thread.
- void * temperature_fn (void *thread_args)

  Entry point for temperature sensor processing thread.

Variables

- static timer_t **timerid**
- struct itimerspec **trigger**
- static i2c_handle_t **i2c_tmp102**
- static float **last_temp_value** = -5
- static mqd_t **temp_queue**
- static shared_data_t * **shm**
- const tmp102_config_t **tmp102_default_config**

5.16.1 Detailed Description

Source file implementing temperature.h.

==========

Author

Roberto Baquerizo (baquerrj), roba8460@colorado.edu

5.16.2 Function Documentation

5.16.2.1 cycle()

```
static void cycle (
            void  )  [static]
```

Cycle function for temperature sensor thread We wait in this while loop checking for requests from watchdog for health status.

========== Function↩
: cycle

Parameters

void
void

```
00364 {
00365    int retVal = 0;
00366    msg_t request = {0};
00367    msg_t response = {0};
00368    while( 1 )
00369    {
00370       memset( &request, 0, sizeof( request ) );
00371       retVal = mq_receive( temp_queue, (char*)&request, sizeof( request ), NULL );
00372       if( 0 > retVal )
00373       {
00374          int errnum = errno;
00375          sem_wait(&shm->w_sem);
00376          sprintf( shm->buffer, "ERROR: Encountered error receiving from message queue %s: (%s)\n",
00377                   TEMP_QUEUE_NAME, strerror( errnum ) );
00378          sem_post(&shm->r_sem);
00379          continue;
00380       }
00381       switch( request.id )
00382       {
00383          case REQUEST_STATUS:
00384             sem_wait(&shm->w_sem);
00385             print_header(shm->header);
00386             sprintf( shm->buffer, "(Temperature) I am alive!\n" );
00387             sem_post(&shm->r_sem);
00388             fprintf( stdout, "(Temperature) I am alive!\n" );
00389             response.id = request.id;
00390             sprintf( response.info, "(Temperature) I am alive!\n" );
00391             retVal = mq_send( request.src, (const char*)&response, sizeof( response ), 0 );
00392
00393             pthread_mutex_lock( &alive_mutex );
```

```
00394             threads_status[THREAD_TEMP]--;
00395             pthread_mutex_unlock( &alive_mutex );
00396             break;
00397          default:
00398             break;
00399       }
00400    }
00401    return;
00402 }
```

5.16.2.2 get_temperature()

```
float get_temperature (
              void  )
```

Returns last temperature reading we have.

========== Function↩
: get_temperature

Parameters

void	

Returns

last_temp_value - last temperature reading we have

<+DETAILED+>

```
00066 {
00067    return last_temp_value;
00068 }
```

5.16.2.3 get_temperature_queue()

```
mqd_t get_temperature_queue (
              void  )
```

Get file descriptor for temperature sensor thread. Called by watchdog thread in order to be able to send heartbeat check via queue.

========== Function↩
: get_temperature_queue

Parameters

void	**temp_queue - file descriptor for temperature sensor thread message queue**

```
00415 {
00416     return temp_queue;
00417 }
```

5.16.2.4 sig_handler()

```
static void sig_handler (
             int signo )  [static]
```

Signal handler for temperature sensor thread. On normal operation, we should be receving SIGUSR1/2 signals from watchdog when prompted to exit. So, we close the message queue and timer this thread owns.

========== Function↩
: sig_handler

Parameters

signo	- enum with signal number of signal being handled **void**

```
00298 {
00299     if( signo == SIGUSR1 )
00300     {
00301        printf("Received SIGUSR1! Exiting...\n");
00302        mq_close( temp_queue );
00303        timer_delete( timerid );
00304        i2c_stop( &i2c_tmp102 );
00305        thread_exit( signo );
00306     }
00307     else if( signo == SIGUSR2 )
00308     {
00309        printf("Received SIGUSR2! Exiting...\n");
00310        mq_close( temp_queue );
00311        timer_delete( timerid );
00312        i2c_stop( &i2c_tmp102 );
00313        thread_exit( signo );
00314     }
00315     return;
00316 }
```

Here is the caller graph for this function:

5.16.2.5 temp_queue_init()

```
int temp_queue_init (
             void  )
```

Initialize message queue for temperature sensor thread.

========== Function↩
: temp_queue_init

Parameters

void	**msg_q - file descriptor for initialized message queue**

```
00429 {
00430    /* unlink first in case we hadn't shut down cleanly last time */
00431    mq_unlink( TEMP_QUEUE_NAME );
00432
00433    struct mq_attr attr;
00434    attr.mq_flags = 0;
00435    attr.mq_maxmsg = MAX_MESSAGES;
00436    attr.mq_msgsize = sizeof( msg_t );
00437    attr.mq_curmsgs = 0;
00438
00439    int msg_q = mq_open( TEMP_QUEUE_NAME, O_CREAT | O_RDWR, 0666, &attr );
00440    if( 0 > msg_q )
00441    {
00442       int errnum = errno;
00443       sem_wait(&shm->w_sem);
00444       print_header(shm->header);
00445       sprintf( shm->buffer, "ERROR: Encountered error creating message queue %s: (%s)\n",
00446                TEMP_QUEUE_NAME, strerror( errnum ) );
00447       sem_post(&shm->r_sem);
00448    }
00449    return msg_q;
00450 }
```

5.16.2.6 temperature_fn()

```
void* temperature_fn (
             void * thread_args )
```

Entry point for temperature sensor processing thread.

========== Function↩
: temperature_fn

Parameters

thread_args	- void ptr to arguments used to initialize thread

Returns

NULL - We don't really exit from this function,

since the exit point is thread_exit()

```
00463 {
00464    /* Get time that thread was spawned */
00465    struct timespec time;
00466    clock_gettime(CLOCK_REALTIME, &time);
00467    shm = get_shared_memory();
00468
00469    /* Write initial state to shared memory */
00470    sem_wait(&shm->w_sem);
00471    print_header(shm->header);
00472    sprintf( shm->buffer, "Hello World! Start Time: %ld.%ld secs\n",
```

```
00473             time.tv_sec, time.tv_nsec );
00474    /* Signal to logger that shared memory has been updated */
00475    sem_post(&shm->r_sem);
00476
00477    signal(SIGUSR1, sig_handler);
00478    signal(SIGUSR2, sig_handler);
00479
00480    temp_queue = temp_queue_init();
00481    if( 0 > temp_queue )
00482    {
00483       thread_exit( EXIT_INIT );
00484    }
00485
00486    int retVal = i2c_init( &i2c_tmp102 );
00487    if( EXIT_INIT == retVal )
00488    {
00489       sem_wait(&shm->w_sem);
00490       print_header(shm->header);
00491       sprintf( shm->buffer, "ERROR: Failed to initialize I2C for temperature sensor!\n" );
00492       sem_post(&shm->r_sem);
00493       thread_exit( EXIT_INIT );
00494    }
00495
00496    timer_setup( &timerid, &timer_handler );
00497
00498    timer_start( &timerid, 1000000 );
00499    cycle();
00500
00501    thread_exit( 0 );
00502    return NULL;
00503 }
```

5.16.2.7 timer_handler()

```
static void timer_handler (
             union sigval sig )  [static]
```

Timer handler function for temperature sensor thread When woken up by the timer, get temperature and write state to shared memory.

========== Function↩ : timer_handler

Parameters

sig
void

```
00329 {
00330    static int i = 0;
00331    led_toggle( LED0_BRIGHTNESS );
00332    sem_wait(&shm->w_sem);
00333
00334    print_header(shm->header);
00335    float temperature;
00336    int retVal = tmp102_get_temp( &temperature );
00337    i++;
00338    if( retVal )
00339    {
00340       sprintf( shm->buffer, "cycle[%d]: %0.5f Celsius\n", i, temperature );
00341    }
00342    else
00343    {
00344       sprintf( shm->buffer, "cycle[%d]: could not get temperature reading!\n", i );
00345    }
00346
```

```
00347     sem_post(&shm->r_sem);
00348     led_toggle( LED0_BRIGHTNESS );
00349     return;
00350 }
```

5.16.2.8 tmp102_get_temp()

```
int tmp102_get_temp (
             float * temperature )
```

Read temperature registers fo TMP102 sensor and decode temperature value.

========== Function↩
: tmp102_get_temp

Parameters

∗temperature	- pointer to location to write decoded value to **EXIT_CLEAN if successful, otherwise EXIT_ERROR**

```
00097 {
00098     uint8_t buffer[2] = {0};
00099     int retVal = i2c_read( TMP102_SLAVE, TMP102_REG_TEMP, buffer, sizeof(
       buffer ) );
00100     if( 0 > retVal )
00101     {
00102         return EXIT_ERROR;
00103     }
00104
00105     uint16_t tmp = 0;
00106     tmp = 0xfff & ( ((uint16_t)buffer[0] << 4 ) | (buffer[1] >> 4 ) ); /* buffer[0] = MSB(15:8)
00107                                                                           buffer[1] = LSB(7:4) */
00108     if( 0x800 & tmp )
00109     {
00110         tmp = ( (~tmp ) + 1 ) & 0xfff;
00111         *temperature = -1.0 * (float)tmp * 0.0625;
00112     }
00113     else
00114     {
00115         *temperature = ((float)tmp) * 0.0625;
00116     }
00117
00118     return EXIT_CLEAN;
00119 }
```

5.16.2.9 tmp102_read_thigh()

```
int tmp102_read_thigh (
             float * thigh )
```

Read value of THigh register of TMP102 sensor and store value (in celsius) in thigh.

========== Function↩
: tmp102_read_thigh

Parameters

thigh	- pointer to location to store decoded temperature value to **EXIT_CLEAN if successful, EXIT_ERROR otherwise**

```
00221 {
00222    uint16_t tmp = 0;
00223
00224    int retVal = i2c_read( TMP102_SLAVE, TMP102_THIGH, (uint8_t*)&tmp, sizeof( tmp ) );
00225    if( 0 > retVal )
00226    {
00227       sem_wait(&shm->w_sem);
00228       print_header(shm->header);
00229       sprintf( shm->buffer, "Could not read from TLow register!\n" );
00230       sem_post(&shm->r_sem);
00231       return EXIT_ERROR;
00232    }
00233
00234    if( tmp & 0x800 )
00235    {
00236       tmp = ~(tmp) + 1;
00237       *thigh = -1 * ( (float)tmp * 0.0625 );
00238    }
00239    else
00240    {
00241       *thigh = (float)tmp * 0.0625;
00242    }
00243
00244    return EXIT_CLEAN;
00245 }
```

5.16.2.10 tmp102_read_tlow()

```
int tmp102_read_tlow (
              float * tlow )
```

Read value of TLow register of TMP102 sensor and store value (in celsius) in tlow.

========== Function↩
: tmp102_read_tlow

Parameters

tlow	- pointer to location to store decoded temperature value to **EXIT_CLEAN if successful, EXIT_ERROR otherwise**

```
00258 {
00259    uint16_t tmp = 0;
00260
00261    int retVal = i2c_read( TMP102_SLAVE, TMP102_TLOW, (uint8_t*)&tmp, sizeof( tmp ) );
00262    if( 0 > retVal )
00263    {
00264       sem_wait(&shm->w_sem);
00265       print_header(shm->header);
00266       sprintf( shm->buffer, "Could not read from TLow register!\n" );
00267       sem_post(&shm->r_sem);
00268       return retVal;
```

```
00269     }
00270
00271     if( tmp & 0x800 )
00272     {
00273        tmp = ~(tmp) + 1;
00274        *tlow = -1 * (float)tmp * 0.0625;
00275     }
00276     else
00277     {
00278        *tlow = (float)tmp * 0.0625;
00279     }
00280
00281     return retVal;
00282 }
```

5.16.2.11 tmp102_write_config()

```
int tmp102_write_config (
              tmp102_config_t * config_reg )
```

Write configuration register of TMP102 sensor.

========== Function↩
: tmp102_write_config

Parameters

*config_reg	- pointer to struct with values to write to configuration register see i2c_write()

```
00081 {
00082     int retVal = i2c_write( TMP102_SLAVE, TMP102_REG_CONFIG, *((uint16_t*)&config_reg)
      );
00083
00084     return retVal;
00085 }
```

5.16.2.12 tmp102_write_thigh()

```
int tmp102_write_thigh (
              float thigh )
```

Write value thigh (in celsius) to Thigh register for TMP102 sensor.

========== Function↩
: tmp102_write_thigh

Parameters

thigh	- value to write to Thigh register **EXIT_CLEAN if successful, otherwise EXIT_ERROR**

```
00131 {
00132    if( (-56.0 > thigh) || (151.0 < thigh) )
00133    {
00134       thigh = 80.0;
00135    }
00136
00137    thigh /= 0.0625;
00138    uint16_t tmp;
00139
00140    if( 0 > thigh )
00141    {
00142       tmp = ( (uint16_t)thigh << 4 );
00143       tmp &= 0x7fff;
00144    }
00145    else
00146    {
00147       thigh *= -1;
00148       tmp = (uint16_t)thigh;
00149       tmp = ~(tmp) + 1;
00150       tmp = tmp << 4;
00151    }
00152
00153    int retVal = i2c_write( TMP102_SLAVE, TMP102_THIGH, tmp );
00154    if( 0 > retVal )
00155    {
00156       sem_wait(&shm->w_sem);
00157       print_header(shm->header);
00158       sprintf( shm->buffer, "Could not write value to THigh register!\n" );
00159       sem_post(&shm->r_sem);
00160       return EXIT_ERROR;
00161    }
00162
00163    return EXIT_CLEAN;
00164 }
```

5.16.2.13 tmp102_write_tlow()

```
int tmp102_write_tlow (
             float tlow )
```

Write value tlow (in celsius) to Tlow register for TMP102 sensor.

========== Function↩
: tmp102_write_tlow

Parameters

tlow	- value to write to Tlow register **EXIT_CLEAN if successful, otherwise EXIT_ERROR**

```
00176 {
00177    if( (-56.0 > tlow) || (151.0 < tlow ) )
```

```
00178    {
00179       tlow = 75.0;
00180    }
00181
00182    tlow /= 0.0625;
00183    uint16_t tmp;
00184
00185    if( 0 < tlow )
00186    {
00187       tmp = ( (uint16_t)tlow << 4 );
00188       tmp &= 0x7fff;
00189    }
00190    else
00191    {
00192       tlow *= -1;
00193       tmp = (uint16_t)tlow;
00194       tmp = ~(tmp) + 1;
00195       tmp = tmp << 4;
00196    }
00197
00198    int retVal = i2c_write( TMP102_SLAVE, TMP102_TLOW, tmp );
00199    if( 0 > retVal )
00200    {
00201       sem_wait(&shm->w_sem);
00202       print_header(shm->header);
00203       sprintf( shm->buffer, "Could not write value to TLow register!\n" );
00204       sem_post(&shm->r_sem);
00205       return EXIT_ERROR;
00206    }
00207
00208    return EXIT_CLEAN;
00209 }
```

5.16.3 Variable Documentation

5.16.3.1 tmp102_default_config

```
const tmp102_config_t tmp102_default_config
```

Initial value:

```
= {
   .mode = {
      .shutdown = TMP102_SHUTDOWN_MODE,
      .thermostat = TMP102_THERMOSTAT_MODE
   },
   .polarity = TMP102_POLARITY,
   .fault_queue = TMP102_FAULT_QUEUE,
   .resolution = {
      .res_0 = TMP102_RESOLUTION_0,
      .res_1 = TMP102_RESOLUTION_1
   },
   .one_shot = 1,
   .operation = TMP102_EXTENDED_MODE,
   .alert = 1,
   .conv_rate = TMP102_CONVERSION_RATE
}
```

5.17 /home/baquerrj/boulder/ecen5013/project_1/src/watchdog.c File Reference

Watchdog source file: the watchdog is responsible for checking that the temperature and light sensor threads are alive.

```
#include "watchdog.h"
#include "temperature.h"
#include "light.h"
#include <errno.h>
#include <string.h>
#include <time.h>
```

Include dependency graph for watchdog.c:

Functions

- static void sig_handler (int signo)

  Signal handler for watchdog. On normal operation, we should be receiving a SIGUSR2 signal from the main thread prompting us to call pthread_kill for the other child threads.
- void kill_threads (void)

  Function to kill children threads.
- void check_threads (union sigval sig)

  Periodically send message via message queue for temperature and sensor threads to check for health. This function is registered as the timer hanlder for the timer owned by the watchdog.
- int watchdog_queue_init (void)

  Initalize message queue for watchdog.
- int watchdog_init (void)

  Initialize watchdog, calling appropriate functions to do so. E.g. calling timer_setup and timer_start to set up timer.
- void * watchdog_fn (void *thread_args)

  Entry point for wachtdog.

Variables

- static timer_t **timerid**
- struct itimerspec **trigger**
- static struct thread_id_s * **threads**
- static mqd_t **thread_msg_q** [NUM_THREADS]
- static mqd_t **watchdog_queue**
- pthread_mutex_t **alive_mutex**

5.17.1 Detailed Description

Watchdog source file: the watchdog is responsible for checking that the temperature and light sensor threads are alive.

==========

Author

Roberto Baquerizo (baquerrj), roba8460@colorado.edu

5.17.2 Function Documentation

5.17.2.1 check_threads()

```
void check_threads (
            union sigval sig )
```

Periodically send message via message queue for temperature and sensor threads to check for health. This function is registered as the timer hanlder for the timer owned by the watchdog.

========== Function↩
: check_threads

Parameters

sig	void

```
00104 {
00105    int retVal = 0;
00106    msg_t request = {0};
00107    request.id = REQUEST_STATUS;
00108    request.src = watchdog_queue;
00109
00110    if( (0 == threads_status[THREAD_TEMP]) && (0 == threads_status[THREAD_LIGHT] ) )
00111    {
00112       pthread_mutex_lock( &alive_mutex );
00113       threads_status[THREAD_TEMP]++;
00114       threads_status[THREAD_LIGHT]++;
00115       pthread_mutex_unlock( &alive_mutex );
00116       retVal = mq_send( thread_msg_q[THREAD_TEMP], (const char*)&request, sizeof( request ), 0 );
00117       if( 0 > retVal )
00118       {
00119          int errnum = errno;
00120          fprintf( stderr, "Encountered error sending status request from watchdog: (%s)\n",
00121                   strerror( errnum ) );
00122       }
00123       retVal = mq_send( thread_msg_q[THREAD_LIGHT], (const char*)&request, sizeof( request ), 0 );
00124       if( 0 > retVal )
00125       {
00126          int errnum = errno;
00127          fprintf( stderr, "Encountered error sending status request from watchdog: (%s)\n",
00128                   strerror( errnum ) );
00129       }
00130    }
00131    else
00132    {
00133       fprintf( stderr, "One of the threads did not return!\n" );
00134       fprintf( stderr, "thread_status[THREAD_TEMP] = %d\nthread_status[THREAD_LIGHT] = %d\n",
00135                threads_status[THREAD_TEMP], threads_status[THREAD_LIGHT] );
00136       kill_threads();
00137       thread_exit( EXIT_ERROR );
00138    }
00139
00140    return;
00141 }
```

5.17.2.2 kill_threads()

```
void kill_threads (
             void  )
```

Function to kill children threads.

========== Function↩
: kill_threads

Parameters

void	void

```
00073 {
```

```
00074     fprintf( stdout, "watchdog caught signals - killing thread [%ld]\n",
00075              threads->temp_thread );
00076     fflush( stdout );
00077     pthread_kill( threads->temp_thread, SIGUSR1 );
00078
00079     fprintf( stdout, "watchdog caught signals - killing thread [%ld]\n",
00080              threads->light_thread );
00081     fflush( stdout );
00082     pthread_kill( threads->light_thread, SIGUSR1 );
00083
00084     fprintf( stdout, "watchdog caught signals - killing thread [%ld]\n",
00085              threads->logger_thread );
00086     fflush( stdout );
00087     pthread_kill( threads->logger_thread, SIGUSR1 );
00088     free( threads );
00089     return;
00090 }
```

Here is the caller graph for this function:

5.17.2.3 sig_handler()

```
static void sig_handler (
             int signo )  [static]
```

Signal handler for watchdog. On normal operation, we should be receiving a SIGUSR2 signal from the main thread prompting us to call pthread_kill for the other child threads.

========== Function↩
: sig_handler

Parameters

signo	- enum with signal number of signal being handled **void**

```
00053 {
00054     if( SIGUSR2 == signo )
00055     {
00056        kill_threads();
00057        mq_close( watchdog_queue );
00058        timer_delete( timerid );
00059        thread_exit( 0 );
00060     }
00061 }
```

Here is the caller graph for this function:

5.17.2.4 watchdog_fn()

```
void* watchdog_fn (
             void * thread_args )
```

Entry point for wachtdog.

========== Function↩
: watchdog_fn

Parameters

thread_args	- void ptr used to pass thread identifiers (pthread_t) for child threads we have to check for health

Returns

NULL - We don't really exit from this function,

since the exit point for threads is thread_exit()

```
00217 {
00218    signal( SIGUSR2, sig_handler );
00219    exit_e retVal = EXIT_ERROR;
00220    if( NULL == thread_args )
00221    {
00222       print_header( NULL );
00223       fprintf( stderr, "Encountered null pointer!\n" );
00224       pthread_exit(&retVal);
00225    }
00226    else
00227    {
00228       threads = malloc( sizeof( struct thread_id_s ) );
00229       threads = (struct thread_id_s*)thread_args;
00230    }
00231
00232    watchdog_init();
00233
00234    while(1);
00235    return NULL;
00236 }
```

5.17.2.5 watchdog_init()

```
int watchdog_init (
              void  )
```

Initialize watchdog, calling appropriate functions to do so. E.g. calling timer_setup and timer_start to set up timer.

========== Function↩
: watchdog_init

Parameters

void	**EXIT_CLEAN, otherwise EXIT_INIT**

```
00184 {
00185    watchdog_queue = watchdog_queue_init();
00186    if( 0 > watchdog_queue )
00187    {
00188       thread_exit( EXIT_INIT );
00189    }
00190
00191    while( 0 == (thread_msg_q[THREAD_TEMP] = get_temperature_queue()) );
00192    while( 0 == (thread_msg_q[THREAD_LIGHT] = get_light_queue()) );
00193
00194    fprintf( stderr, "Watchdog says: Temp Queue FD: %d\n", thread_msg_q[0] );
00195    fprintf( stderr, "Watchdog says: Light Queue FD: %d\n", thread_msg_q[1] );
```

```
00196
00197    pthread_mutex_init( &alive_mutex, NULL );
00198    timer_setup( &timerid, &check_threads );
00199
00200    timer_start( &timerid, 4000000 );
00201
00202    return EXIT_CLEAN;
00203 }
```

5.17.2.6 watchdog_queue_init()

```
int watchdog_queue_init (
             void  )
```

Initalize message queue for watchdog.

========== Function↩
: watchdog_queue_init

Parameters

void	**msg_q - file descriptor for initialized message queue**

```
00153 {
00154    /* unlink first in case we hadn't shut down cleanly last time */
00155    mq_unlink( WATCHDOG_QUEUE_NAME );
00156
00157    struct mq_attr attr;
00158    attr.mq_flags = 0;
00159    attr.mq_maxmsg = MAX_MESSAGES;
00160    attr.mq_msgsize = sizeof( msg_t );
00161    attr.mq_curmsgs = 0;
00162
00163    int msg_q = mq_open( WATCHDOG_QUEUE_NAME, O_CREAT | O_RDWR, 0666, &attr );
00164    if( 0 > msg_q )
00165    {
00166       int errnum = errno;
00167       fprintf( stderr, "Encountered error creating message queue %s: (%s)\n",
00168                WATCHDOG_QUEUE_NAME, strerror( errnum ) );
00169    }
00170    return msg_q;
00171 }
```

Index