Project 1 Architecture

Ben Heberlein

ECEN 5013

User Space

Tasks

main

main.c

This task is responsible for creating, initializing, and monitoring the other tasks. The main task will periodically send out a health check to the other tasks to make sure they are still functioning. If the main task does not receive the correct response within 10ms, the other task will be killed and restarted. The main task also makes decisions based on temperature, light, and error conditions. User LEDs 0-3 represent the following conditions - too hot, too cold, too bright, and missed heartbeat check, respectively.

The API to interact with the main task will be discussed in the IPC section.

temp_task

temp.c

This task is responsible for managing interactions with the TMP106 temperature module. It contains commands to initialize, read, write, sleep, wake up, check health, and get temperature. The task will use a timer to wake up periodically and read the temperature from the module, storing it in local data. This data will be converted and sent out upon request from another thread. This task communicates with the TMP106 through the *mraa* I2C library, which runs on top of a device driver */dev/i2c-2*.

If killed abruptly, this thread runs a cleanup routine where it sends a log message and shuts down gracefully.

The API to interact with the temperature task will be discussed in the IPC section.

light_task

light.c

This task is responsible for managing interactions with the APDS-9301 light sensor. It contains commands to initialize, read, write, enable/disable interrupts, check if it is day or night, and check health. The task will use a timer to wake up periodically and read the luminosity from the module, storing it in local data. This data will be converted to lumens and send out upon request by another thread. This task will communicate with the APDS-9301 sensor through the *mraa* I2C library, which runs on top of a driver in */dev/i2c-2*.

If the light reading is more than 2x different than the previous reading, this task will send a message to the logger.

If killed abruptly, this thread runs a cleanup routine where it sends a log message and shuts down gracefully.

The API to interact with the light task will be discussed in the IPC section.

log_task

log.c

This task will log messages to a file. The messages can be of several levels (LOG_LEVEL_ERROR, LOG_LEVEL_WARNING, LOG_LEVEL_INFO, LOG_LEVEL_DEBUG). The message queue used by this task can handle longer data than the other tasks)

If killed abruptly, this thread runs a cleanup routine where it sends a log message and shuts down gracefully, flushing and closing the log file.

The API to interact with the log task will be discussed in the IPC section.

IPC

msg.c

The interprocess communication will be done with POSIX message queues. Each task will have it’s own message queue for handling requests from other threads. A standard message packet will look like the following.

RSP	FROM	CMD	DATA ...	NULL

RSP will be 1 bit.

If RSP is 1, this represents a response from the FROM thread.
If RSP is 0, this represents a request from the FROM thread.

FROM will be 7 bits.

This is the ID of the thread that made the request.
Effectively RSP and FROM will be one 8 bit field in the data structure

CMD will be 8 bits.

This will be the command to request from the receiving task.

DATA can be up to 13 bytes (or up to 253 for log messages).

The data segment should always be terminated by the NULL.
The data segment is defined in more detail for certain operations, such as logging.

NULL will be 1 byte.

This will always follow the DATA segment.
If there is less than the maximum amount of DATA bytes, NULL should be placed at the end and the rest ignored.

The message system also contains a useful macro for building response case statements. When a task sends a message, they will receive the response from the other task (if applicable) in their message queue. In order to handle the large number of possible responses (which could grow as more tasks are created), the user should build response handlers as necessary. The macro MSG_RSP can be useful for this. It combines a FROM field with a CMD field, basically making one larger unique message response value that can be used uniquely in a case statement.

The public API for all tasks is given below.

main

MAIN_EXIT

No data

No response

If this is requested, main will kill all threads gracefully and terminate the program.

API MACROS

Thread IDs

```
#define MAIN_THREAD_TOTAL 4
#define MAIN_THREAD_MAIN  0
#define MAIN_THREAD_LIGHT 1
#define MAIN_THREAD_TEMP  2
```

#define MAIN_THREAD_LOG   3

Main task API
#define MAIN_EXIT 0

Main timer durations
#define MAIN_TIMER_HEARTBEAT_NS 500000000
#define MAIN_TIMER_LOGIC_NS   400000000

LEDs
#define MAIN_LED0   0
#define MAIN_LED1   1
#define MAIN_LED2   2
#define MAIN_LED3   3
#define MAIN_LED_ON   1
#define MAIN_LED_OFF   0

Logic for LEDs
#define MAIN_TOOCOLD   10.0
#define MAIN_TOOHOT   30.0
#define MAIN_TOOBRIGHT 50.0

temp

TEMP_INIT

No data

No response

Initialize the temperature task.

TEMP_READREG

Data should be one address bytes

Response should be two bytes of data followed by one address byte (same as data)

Read a register in the temperature module.

TEMP_WRITEREG

Data should be one address byte followed by two data bytes

No response

Write a register in the temperature module.

TEMP_WRITECONFIG

Data should be two data bytes

No response

Write data to the configuration register.

TEMP_WRITEPTR

Data should be one data byte to write (pointer reg is one byte)

No response

Write data to the pointer register.

TEMP_GETTEMP

Data should be one byte of TEMP_FMT_X macro set

Response should be 4 bytes of type float followed by the same input format data

Gets the temperature in the specified format

TEMP_WRITERES

Data should be one byte for conversion rate of TEMP_CONV_x macro set

No response

Configures the sensor resolution to the specified amount.

TEMP_SHUTDOWN

No data

No response

Puts the temperature module to sleep.

TEMP_WAKEUP

No data

No response

Wakes the temperature module back up from sleep.

TEMP_ALIVE

No data

Response should be one byte for alive (0xA5)

Checks if the temperature task is still alive. If no response after a certain amount of time, the task should be killed and restarted.

TEMP_KILL

No data

No response

Kills the task gracefully. Can only be requested from the main task.

API MACROS

Temperature API

```
#define TEMP_INIT           0
#define TEMP_READREG        1
#define TEMP_WRITEREG       2
#define TEMP_WRITECONFIG    3
#define TEMP_GETTEMP        4
#define TEMP_SETCONV        5
#define TEMP_SHUTDOWN       6
#define TEMP_WAKEUP         7
#define TEMP_ALIVE          8
#define TEMP_KILL           9
#define TEMP_WRITEPTR       10
```

I2C and sensor macros

```
#define TEMP_I2C_BUS  2
#define TEMP_I2C_ADDR 0x48
#define TEMP_REG_TEMP 0x00
#define TEMP_REG_CTRL 0x01
#define TEMP_REG_HIGH 0x02
#define TEMP_REG_LOW  0x03
#define TEMP_REG_CTRL_OS    (1<<15)
#define TEMP_REG_CTRL_R1    (1<<14)
#define TEMP_REG_CTRL_R0    (1<<13)
#define TEMP_REG_CTRL_F1    (1<<12)
#define TEMP_REG_CTRL_F0    (1<<11)
#define TEMP_REG_CTRL_POL   (1<<10)
#define TEMP_REG_CTRL_TM    (1<<9)
#define TEMP_REG_CTRL_SD    (1<<8)
#define TEMP_REG_CTRL_CR1   (1<<7)
#define TEMP_REG_CTRL_CR2   (1<<6)
#define TEMP_REG_CTRL_AL    (1<<5)
#define TEMP_REG_CTRL_EM    (1<<4)
```

Temperature formats
#define TEMP_FMT_CEL 0
#define TEMP_FMT_FAR 1
#define TEMP_FMT_KEL 2

Conversion rates
#define TEMP_CONV_0_25HZ 0
#define TEMP_CONV_1HZ   1
#define TEMP_CONV_4HZ   2
#define TEMP_CONV_8Hz   3

Resolution per DN (C)
#define TEMP_RES 0.0625

Temperature update timer
#define TEMP_TIMER_NS 260000000

light_queue

LIGHT_INIT

No data

No response

Initializes the light task.

LIGHT_READREG

Data should be one byte for address to read

Response should be one byte of read data

Reads a register in the light module.

LIGHT_WRITEREG

Data should be one byte for address to write followed by one data byte

No response

Writes a register in the light module.

LIGHT_WRITEIT

Data should be one byte for integration time of LIGHT_INT_X macro set

No response

Configures the integration time in the light module.

LIGHT_GETLUX

No data

Response is four bytes of floating point representing lux output

Get the luminosity from the light module.

LIGHT_ENABLEINT

No data

No response

Enable interrupts in the light module.

LIGHT_DISABLEINT

No data

No response

Disable interrupts in the light module.

LIGHT_READID

No data

Response is one byte of ID register

Read the identification register in the light module.

LIGHT_ISDAY

No data

Response is one byte, 1 if it is day, and 0 if it is night.

Check if the light sensor thinks it is day or night.

LIGHT_ALIVE

No data

Response should be one byte for alive (0xA5)

Checks if the light task is still alive. If no response after 10ms, the task should be killed forcefully and restarted.

LIGHT_KILL

No data

No response

Kills the task gracefully. Can only be called from the main task.

API MACROS

Light API
```
#define LIGHT_INIT          0
#define LIGHT_READREG       1
#define LIGHT_WRITEREG      2
#define LIGHT_WRITEIT       3
#define LIGHT_GETLUX        4
#define LIGHT_ENABLEINT     5
#define LIGHT_DISABLEINT    6
#define LIGHT_READID        7
#define LIGHT_ISDAY         8
#define LIGHT_ALIVE         9
#define LIGHT_KILL          10
```

I2C and register macros
```
#define LIGHT_I2C_BUS       2
#define LIGHT_I2C_ADDR      0x39
#define LIGHT_CMD_READ      0xA0
#define LIGHT_CMD_WRITE     0x80
#define LIGHT_CMD_ADDR_MASK 0x0f
#define LIGHT_REG_CTRL      0
#define LIGHT_REG_TIME      1
#define LIGHT_REG_THRESHLL  2
#define LIGHT_REG_THRESHLH  3
#define LIGHT_REG_THRESHHL  4
#define LIGHT_REG_THRESHHH  5
#define LIGHT_REG_INT       6
#define LIGHT_REG_CRC       8
#define LIGHT_REG_ID        10
#define LIGHT_REG_DATA0L    12
#define LIGHT_REG_DATA0H    13
#define LIGHT_REG_DATA1L    14
#define LIGHT_REG_DATA1H    15
```

Interrupt options
```
#define LIGHT_INT_EN (0x01 << 4)
#define LIGHT_INT_DIS (0x00 << 4)
```

Integration times in ms
```
#define LIGHT_INT_13_7 0
```

#define LIGHT_INT_101 1
#define LIGHT_INT_402 2

Day or night calculation
#define LIGHT_DAY_THRESH 3.0

Lux get timer
#define LIGHT_TIMER_NS 200000000

log_queue

LOG_INIT

Data is one byte, '1' for new file and '0' for append (useful for reopening after log_kill

No response

Initializes the log task.

LOG_LOG

Data should be of the form

LEVEL	*MSG*	*NULL*

Where LEVEL matches one of the macros in LOG_LEVEL_X set, and MSG can be up to 252 bytes of ASCII formatted data. The log task has a macro LOG_FMT that will format text like printf that can be used to format data before sending. The MSG field should be NULL terminated like a normal message if not using the full data field.

No response

Logs the message to the file at the desired level. This request will include a timestamp with the logged message.

LOG_SETPATH

Data should be an ASCII string for the new file path.

No response

Sets the log path to the specified file and opens the file for write. If the file already exists, it will be overwritten.

LOG_ALIVE

No data

Response should be one byte for alive (0xA5)

Checks if the log task is still alive. If no response after 10ms, the task should be killed forcefully and restarted.

LOG_KILL

No data

No response

Kills the task gracefully. Can only be called from the main task.

API MACROS

Log format macro
Can format strings with sprintf. Can be used by requesting threads to format data before requesting a log. Arguments are FROM, LOG_LEVEL, and the command packet TX to put data into. Should be used for all log calls to make code easy and consistent.

```
#define LOG_FMT(fr, lvl, tx, ...) do { (tx).from = (fr); \
                                      (tx).cmd = LOG_LOG; \
                                      (tx).data[0] = (lvl); \
                                      sprintf((char *) &((tx).data[1]), __VA_ARGS__); \
                                     } while (0)
```

Log task API

```
#define LOG_INIT    0
#define LOG_LOG     1
#define LOG_SETPATH 2
#define LOG_ALIVE   3
#define LOG_KILL    4
```

Log levels

```
#define LOG_LEVEL_DEBUG 0
#define LOG_LEVEL_INFO  1
#define LOG_LEVEL_WARN  2
#define LOG_LEVEL_ERROR 3
```

Third Party Userspace Libraries

MRAA

To facilitate I2C communication in an easy way, the system uses a third party I2C library called *MRAA* to communicate with the I2C device driver. This library provides a very simple read and write interface that maps to file commands on the */dev/i2c-x* device drivers.

Originally, the application was going to use two custom */dev* drivers for the APDS-9301 and TMP102 modules. However, given time and implementation constraints, using the *MRAA* library seemed like a good solution for presenting full thread functionality without writing custom device drivers. The */dev/i2c-x* interface implements the functionality we need in this case.

MRAA can be found at https://iotdk.intel.com/docs/master/mraa.

Cmocka

For unit testing the application, Cmocka was used. Cmocka is a very lightweight but powerful unit testing framework that can easily integrate into embedded C environments.

Cmocka can be found at https://cmocka.org.

Kernel Space

Devices

The application in user space will communicate with the temperature and light sensor modules using the */dev/i2c-2* interface. This is connected to GPIO pins on the Beagle Bone Green, and it easy to interface with. Using the *MRAA* library mentioned previously, we can access this device and use it to send and receive I2C information.

The main thread will also interface with the user LEDs on the board using the */sys/devices/platform/leds/leds/beaglebone:green:usrx/brightness* device driver. This is a very simple interface that can be used to access the user LEDs. Writing a 1 to LED *usrx* turns it on, and writing a 0 turns it off.

System Call Interfaces

The user space application will aso utilize POSIX standard operating system interfaces for thread creation/management as well as message queue implementation. The Pthreads library will be used for thread management and the mqueue library will be used for messaging between threads. These both run on top of lower level POSIX implementations in the kernel.

Kernel Libraries

The I2C device driver in */dev/i2c-x* runs directly on kernel space I2C libraries. Kernel library functions used include the following.

I2c_master_send

I2c_master_recv

I2c_transfer

I2c_add_driver

I2c_del_driver

…

As mentioned previously, we will also be using POSIX libraries through system calls when using POSIX threads and message queues.

Hardware

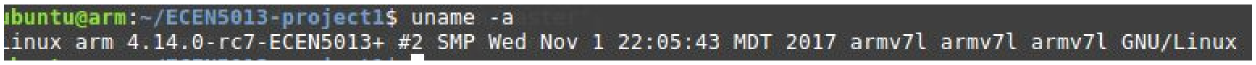
The hardware is perhaps the most straightforward part of the architecture. The whole system will be implemented on a Beagle Bone Green. We will be communicating with the two modules over a standard I2C bus. The two modules, as mentioned earlier are the TMP106 temperature sensor from Texas Instruments and the APDS-9301 light sensor from Broadcom.

The Linux kernel will be built on a host machine for the Beagle Bone Green and transferred over using the SD card. If possible, network booting will be implemented on the Beagle Bone Green for quicker kernel space development and a more controlled deployment system overall.

Other Details

For this project, we made a custom kernel build for the Beagle Bone Green called *4.14.0-rc7-ECEN5013+*, based on the 4.14 version of the Linux Kernel for Beagle Bone found at https://github.com/beagleboard/linux.

The following image shows the output of *uname*, where we can see the custom kernel.



```
ıbuntu@arm:~/ECEN5013-project1$ uname -a
.inux arm 4.14.0-rc7-ECEN5013+ #2 SMP Wed Nov 1 22:05:43 MDT 2017 armv7l armv7l armv7l GNU/Linux
```