University of Colorado
Boulder

MULTITHREADED LINUX SYSTEM DESIGN

BY

SIDDHANT JAJOO
SATYA MEHTA

UNDER THE GUIDANCE OF:

PROFESSOR RICHARD HEIDEBRECHT,
UNIVERSITY OF COLORADO BOULDER

3/31/2019

GITHUB LINK: https://github.com/satya45/Satya_Siddhant/tree/master/AESD_Project-1

TABLE OF CONTENTS

OVERVIEW

This project designs and implements a smart environment monitoring device using Beaglebone Green development board along with two off-board sensors. The project design includes concurrent software concepts for Linux that will interact with both User-Space and Kernel-Space in addition to multiple connected devices.
The two off-board sensors are:

- Temperature Sensor TMP102
- Light Sensor APDS 9301

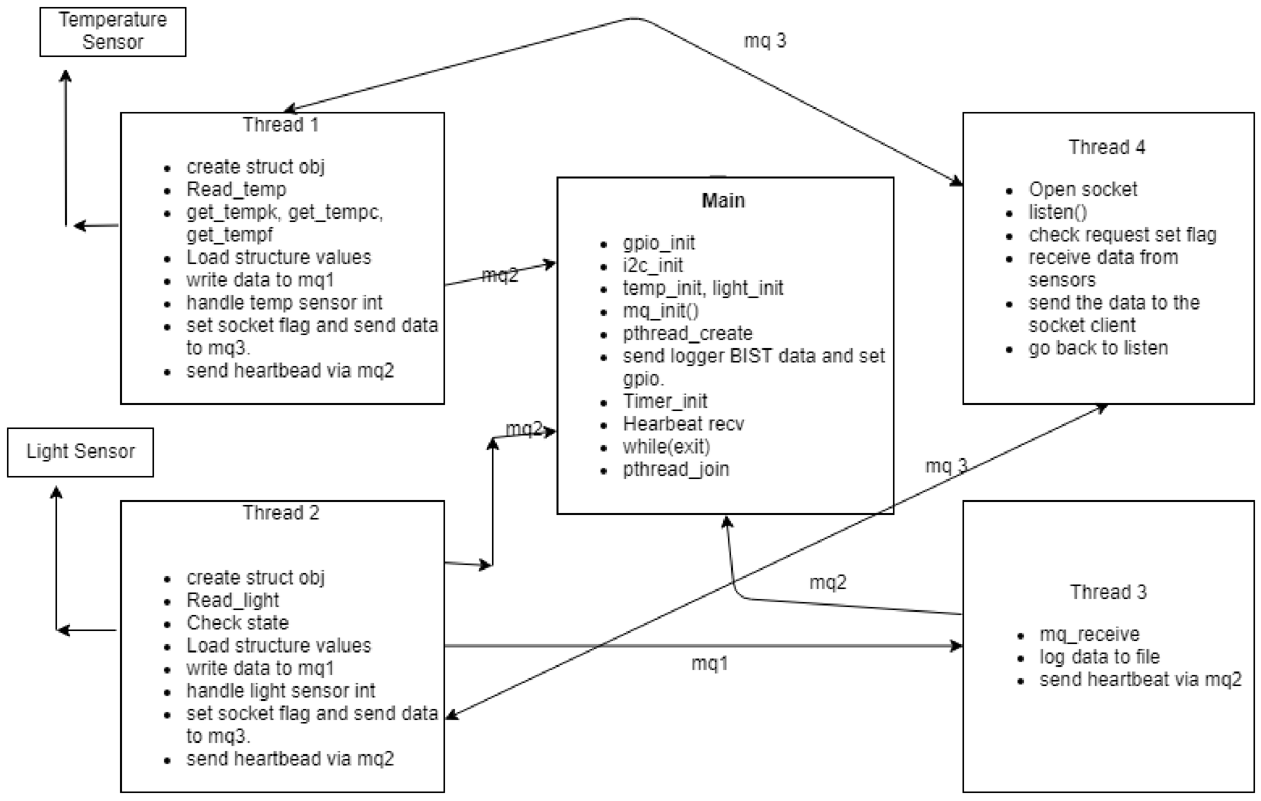
The project is implemented by creating 4 threads named: 1. Temperature task 2. Light Task 3.Logger Task 4. Remote Host Task. Temperature and Light threads monitor the two sensors concurrently using POSIX timers to read data at specified intervals of time. The data is simultaneously recorded from the two sensors in independent threads. The data obtained is then logged into a text file which is obtained at runtime (i.e passing arguments to main). This data is backed up and a new text file is created whenever the program is run again. The logging of data to a text file is implemented by the logger thread. There is also a provision to provide specified amounts of data to a remote off-system host via remote request socket thread i.e the Remote Host thread. This thread uses Socket as an IPC Mechanism to request and send data to the remote host. All the data and each and every update in the system is written to a text file via the logger task.

This Project implements the following features and concepts:

1. Multithreading using POSIX Threads
2. I2C
3. Design Threads to operate in an event loop
4. Error Checking and Handling
5. Unit Testing
6. Built in Self Tests
7. Heartbeat Notification from all threads
8. Implemented Log level – INFO, WARNING, ERROR,DEBUG.
9. POSIX Timers
10. POSIX Clocks
11. Signal Handlers
12. Synchronization Primitives
13. Inter Process Communication (IPC) – Sockets and Message Queues
14. Request data from Remote host

NOTE: *Throughout the report, thread and task would be used interchangeably as they both refer to the same thing in linux.*

BLOCK DIAGRAM



DESIGN CONSIDERATIONS

CHOICE OF STRUCTURE TO STORE MESSAGE DATA

The Sensor data has been stored in a structure which consists of two fields:

1. Field identifying which event has been triggered.
2. A union containing different structures of different sensors.

For this application there are four structures:

- Structure consisting of data required for temperature sensor.
- Structure consisting of data required for light sensor.
- Structure consisting of error values obtained.
- Structure consisting of string to be printed in the text file for debug messages.

Union was used to achieve memory optimization. At one point of time only one structure inside a union will be accessed in their respective threads by a local object of the sensor data structure created in the thread. So there was no point in having different structures inside another structure. There could be an implementation where all the data required for different sensors could have been added as elements in one single structure but after considering worst case scenarios, we reached to the conclusion that having a union is the most optimized way to deal with memory usage.

Example: Assume all the data fields in the structures below having 4 bytes as their size.

```
Structure sensor 1          Structure sensor 2          Structure sensor 3
{                           {                           {
        Data1;                      Data1;                      Data1;
        Data2;                      Data2;                      Data5;
        Data3;                      Data3;                      Data6;
}                                   Data4;                      Data7;
                            }                           }
```

Size of Sensor 1 will be 4*3 = 12 bytes.
Size of Sensor 2 will be 4*4 = 16 bytes.
Size of Sensor 3 will be 4*4 = 16 bytes.
So the maximum size that would be used would be 16 bytes as they are part of a union.
If the structure is as defined below:

```
Structure all_sensors
{
Data1;
Data2;
Data3;
Data4;
Data5;
Data6;
Data7;
}
```

Size of all sensors structure would be 4*7= 28 bytes. This concludes that if we try to add all the elements in a single structure, the memory used would be equal to or greater than using a union of all the structures. Thus, in order to reduce memory wastage, a union of sensor structures was implemented. The **id data field** signifies which thread/sensor has sent the data.

There has been a tradeoff between modularity and memory.

CHOICE OF IPC

The IPC Mechanism used for sending data between the temperature Task and the logger task, the Light Task and the Logger task is a single message queue. Both the tasks use the same message queue to transmit the data to the logger task. The reason for choosing message queues is that they take care of chunking/dechunking, so that your application doesn't have to figure out where one message ends and the next begins**.** Our application consists of different sizes of data, we would never know which data has been sent at a particular point of time. For this purpose message queues were implemented. There can also be a scenario for implementing two pipes, two message queues or two shared memory but that would just add overhead and more lines of code. To maintain modularity and ease of coding, message queues have been implemented wherever IPCs are needed to communicate between threads. Three message queues and one socket has been implemented throughout the system.

The three message queues are:

1. Temperature and Light task communicate to the logger task via one message queue.(named as log_mq)
2. Temperature and Light task communicate to the socket task via second message queue.(named as sock_mq)
3. Heartbeat mechanism is achieved for temperature, light and logger task via the third message queue. (named as hb_mq)

Generalized API functions have been made to maintain modularity.

HEARTBEAT NOTIFICATIONS

Heartbeat Notifications is achieved by using a common message queue between the main and each task. A timer in the main keeps a tab on the heartbeats received. If the timeout occurs the main task checks which task has failed to send a message to the main task via message queue and takes appropriate measures in the form of resetting the threads.

SENSORS

TEMPERATURE SENSOR TMP102

The TMP102 device is a digital temperature sensor ideal for NTC/PTC thermistor replacement where high accuracy is required. The device offers an accuracy of ±0.5°C without requiring calibration or external component signal conditioning. This is an I2C connected sensor with a resolution of 0.0625 Degrees Celsius. The device is specified for operation over a temperature range of –40°C to 125°C. The temperature sensor has an alarm pin and has configurable Thigh and Tlow pins to set the range for the alarm pin. Whenever the temperature crosses this range specified by Tlow and Thigh, the alarm pin pull low. The sensor has the ability to go into shutdown mode.

For further information, refer the links given below:

- ✓ https://www.sparkfun.com/products/13314
- ✓ http://www.ti.com/lit/ds/symlink/tmp102.pdf

LIGHT SENSOR (APDS 9301)

The APDS-9301 is Light-to-Digital Ambient Light Photo Sensor that converts light intensity to digital signal output capable of direct I2C interface. Each device consists of one broadband photodiode (visible plus infrared) and one infrared photodiode. Two integrating ADCs convert the photodiode currents to a digital output that represents the irradiance measured on each channel. This digital output can be input to a microprocessor where illuminance (ambient light level) in lux is derived using an empirical formula to approximate the human-eye response.

For further information, refer the links given below:

- ✓ https://www.broadcom.com/products/optical-sensors/ambient-light-photo-sensors/apds-9301
- ✓ file:///C:/Users/user1/Downloads/AV02-2315EN0.pdf

WORKING SUMMARY

MAIN THREAD

The main task carries out all the initialization functions such as gpio, signal, queue, i2c, mutex, timer for heartbeats. The thread then continuously handles the heartbeats received from different threads and responds if any heartbeat from a particular thread fails to reach it. Upon signal interruption (ctrl + c), it exits the while loop which was used to handle the heartbeats and destroys all the parameters that were initialized at the beginning and frees any memory allocated during the course of the program execution. It also cancels all the threads using pthread_cancel and waits for the threads to join main and finally exits.

TEMPERATURE THREAD

This is a POSIX thread which continuously gathers data from the temperature sensor at regular intervals of time using a POSIX Timer. The timer is initialized at the start of the execution of the thread. The temperature sensor is connected to the Beaglebone Green and acquires data using I2C interface. The thread checks the timer flag which is set in the timer handler at regular intervals of time and then the thread reads data from the sensor using the I2C bus. The API for gathering sensor data is written in such a way that the data obtained is stored in the structure sensor_struct. The structure also contains a field signigfying which event to invoke. This field "id" is used by the logger thread to print appropriate information and is set by the same function that obtains the temperature data. The return value of this API is sensor_struct which is defined in main.h. This structure also gathers and stores the timestamp signifying at what time the data was read from the sensor. This structure is then sent to the logger task using the API queue_send(). After sending the data to the logger thread it checks for any socket request for temperature. If there are any socket requests pending then the thread serves them by sending the structure sensor_struct containing data to the socket thread via socket message queue and to the logger thread too. Heartbeat is sent to the main process using hb_mq at the end of thread. The structure for temperature is as follows:

```
struct
temp_struct
             {
                  float temp_c;
                  struct timespec data_time;
             };
```

Following functions are defined inside temp.c files to explore the temperature sensor.

- Write_pointer()
- Read_temp_data()
- Read_temp_Reg()
- Shutdown_mode()
- Write_tlow()
- Write_thigh()
- Read_temp_reg()

LIGHT THREAD

This thread is specifically created to carry out functions for the light sensor. This thread functions in the same way as the temperature thread. It reads data at certain intervals of time using I2C, stores it in the structure sensor_struct, writes the appropriate event id in the structure, sends them via message queue to the logger thread and then finally services socket requests if any. It also keeps a tab on the current light state based on a threshold value specifying whether it is currently 'LIGHT' or 'DARK'. When the state changes, it is printed in the log file. Heartbeats are sent at the last to ensure that the threads are alive.

Following functions are defined inside the light.c to explore the light sensor.

- Read_light_data()
- Light_id()
- Write_command()
- ADC_CH0()
- ADC_CH1()
- Lux_data()
- Read_light_reg()
- Write_timing_reg()
- Write_int_ctrl()
- Read_int_th()
- Write_int_th()

LOGGER THREAD

The logger thread continuously dequeues data from log_mq. It blocks if the queue is empty and waits for the queue to get filled with data. It sends a heartbeat everytime it reads data. The data is printed in the logfile depending on the id it receives from the structure in log_mq.

SOCKET THREAD

The socket thread initializes socket as server. It waits on listen() for the remote system to initialize the connection. After the connection, the socket thread receives the socket request. On socket request it sets the socket_flag based on the request received. After setting the flag it waits upon the socket queue to receive the structure from the sensor threads. Upon the reception of the data from the sensors it sends the data to the remote host through the socket.

ADDITIONAL FEATURES

REMOTE OFF-SYSTEM HOST

A program was implemented to request for current temperature, light, light state readings etc from the system. This was accomplished using sockets.

The system supports the following commands:

1. TC – Gets temperature in Celsius
2. TK – Gets Temperature in Kelvin.
3. TF – Gets Temperature in Fahrenheit
4. L – Gets Light LUX Value.

STARTUP/BUILT-IN SELF TESTS

Before the application begins operation, some startup tests are run at the beginning to verify that the hardware and software are in working order. These tests consist of:

- Communication with the Temperature sensor that can confirm that I2C works and that the hardware is functioning.
- Communication with the Light sensor that can confirm that I2C works and that the hardware is functioning.
- Communication to the threads to make sure they have all started and up and running.
- Sending log messages to the logger task when individual BIST tests have finished along with an indicator if the hardware is connected and in working order. If something does not startup correctly, an error statement is logged and the USR Led is turned on.
- Queues and Signal Initializations.

UNIT TESTS

This software is designed in a modular way so that software can be tested when not fully integrated. Unit tests are done for:

- Queues.
- Temperature Sensor conversions with mocked data.
- Light Sensor conversions with mocked data.

LOG LEVELS AND HIGHER PRIORITIES TO ERRORS

Different log levels have been implemented in this application. Different bits in uint8_t represent different log levels. The different log levels are as follows:

1. INFO (bit 1) – This level only displays the temperature and light readings.
2. WARNING (bit 2) – All the stuff that does not significantly affect the system and can be let as a warning to the user is displayed in this log level.
3. ERRORS (bit 3) – Only the errors are reported in the text file in this log level.
4. DEBUG (bit 4) – All the debug messages and every other message is printed in the text file in this particular log level.

The log level is implemented by assigning a log level to every printf. The log level is accepted as a command line argument. Only the accepted command line log level messages are displayed.

In order to notify the users of errors as quickly as possible, the errors are given the highest priority. The APIs are made in order to support modularity so that if required to interface any additional sensors in future, such features can be useful.

SIGNAL HANDLER

A signal handler is implemented in order to gracefully close and exit the program. All the objects, file descriptors, message queues, threads created and any memory that was allocated at the beginning or during the program execution is freed or destroyed on the reception of the signal INT (CTRL + C).

STEPS TO ADD A SENSOR:

The application has been coded to keep it as modular and generalized as possible so that if needed in future to add a sensor to the system, it takes the least amount of steps to accomplish it.

Follow and make these specifies changes in the code to add a senor to the system:

1. Create a structure for the sensor data and add it into the structure sensor_struct in union in the file main.h.
2. Add pthread_create() for the particular sensor and write the code according to the functionality of the sensor with generalized queue_send() and queue_receive() functions for the sensor.
3. Create a new macro for a new timer sensor. Add the functionality to initialize a timer in timer_init(new timer macro created) and write the corresponding timer handler as required. Create a respective timer flag to use in the respective sensor thread.
4. Use the same log_mq message queue to send the structure data to the logger thread to log data in the text file.
5. Create an event macro for the particular sensor to include in the logger thread and write printfs as required.
6. Create a read_sensor_data() function to write the appropriate event macro in the id field and data in the sensor data field.
7. Create a macro for the heartbeat field as the rest of the sensor and add it into the existing function.
8. Modify the destroy_all() function to include the function to destroy the thread and timer for the new sensor.
9. Add pthread_join() for the new thread.

CHALLENGES FACED

- The most challenging task was to get the overview of the project and to make design considerations according to the constraints and the application.
- Another task that was difficult was to make generalised API commands to maintain modularity in order to add any sensor to the same system in future.
- As embedded systems have very small amounts of memory it was necessary to implement in such a way that there is usage of least amount of data.
- Another task was to close gracefully in case of failure and free any resources or memory which were used before exiting.
- It was exciting to have log levels and to implement them for the first time.
- Designing codes for interrupts on beaglebone which was eventually achieved using polling a long task.
- Lastly, the most difficult task was to implement I2C kernel.

REFERENCES

- https://www.sparkfun.com/products/13314
- http://www.ti.com/lit/ds/symlink/tmp102.pdf
- https://www.broadcom.com/products/optical-sensors/ambient-light-photo-sensors/apds-9301
- file:///C:/Users/user1/Downloads/AV02-2315EN0.pdf