

**PROJECT REPORT**

**ADVANCED EMBEDDED SOFTWARE DEVELOPMENT**

**SPRING 2019**

***SMART ENVIRONMENT MONITORING SYSTEM***

**GROUP MEMBERS:**

**OM RAHEJA**

**SORABH GANDHI**

**INTRODUCTION**

The project is designed to demonstrate a system which monitors various environmental conditions with the help of numerous off-board sensors. The project implements a multithreaded application which is intended to run on Beagle Bone Green (Linux). The design of the project includes contemporaneous software concepts for Linux that helps interact with both User-Space and Kernel-Space in addition to multiple connected devices. While developing this project, we made use of two sensors to simulate our design, a TMP102 Temperature sensor and an APDS 9301 Light sensor. Data from these two sensors were read using the I2C protocol. These sensors made use of the same I2C bus and the received data were logged into a single file on the system along with any exceptional conditions that occurred (disconnected sensor, missing sensor etc.). Additionally, an external interfacing task serviced requests from an off-system/simulated host inquiring the system’s log/status.

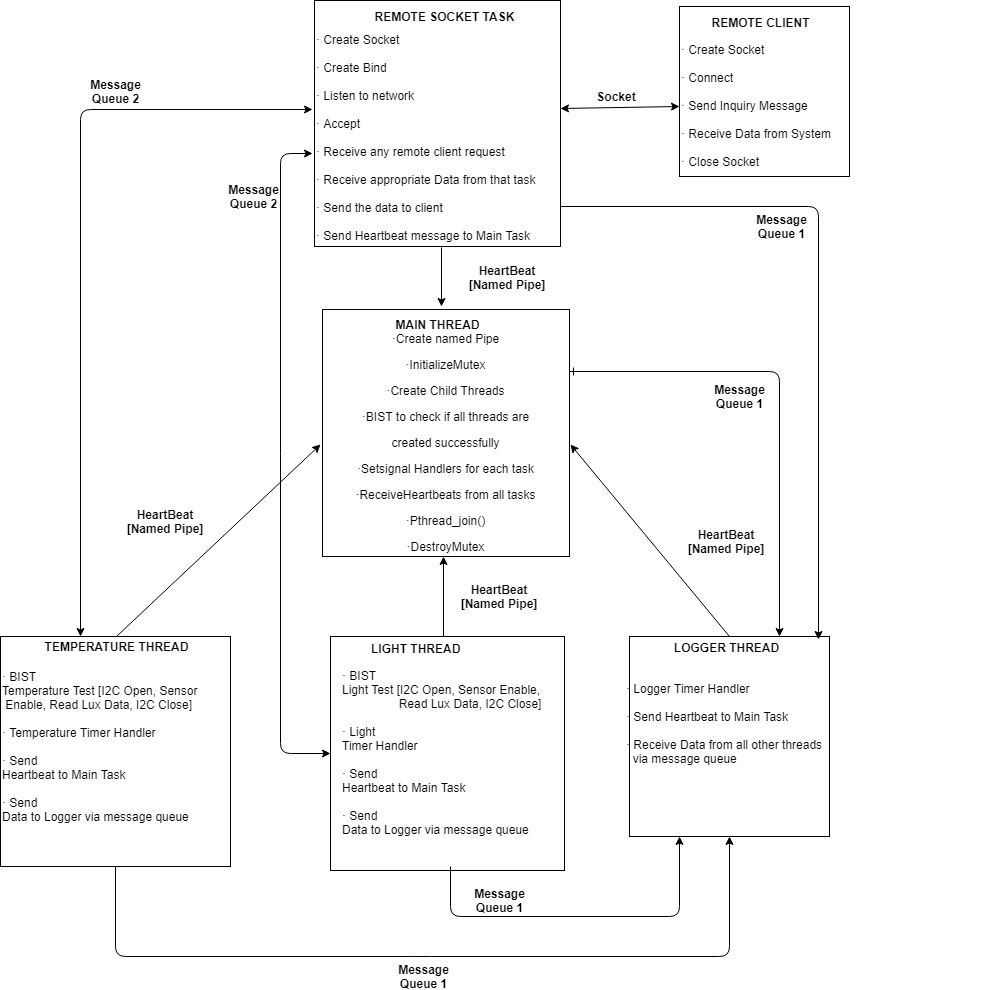
The architecture for this project was such that it consisted of five threads/tasks in total. The main thread spawns 4 child threads, each of which performed an independent task.

The 4 threads/tasks were as mentioned below:

* Temperature task – Measures Temperature (in C/K/F) from the sensor and logs it to the log file.
* Light task – Measures Light (Lux values) from the sensor and logs it to the log file.
* Logger task – Receives data from each thread/task and logs it into a single log file.
* Remote Socket Task – Services requests to the main task to inquire about the system’s log/status.

A POSIX timer is used to read the data from each of the two sensors and log this information to the log file at specified intervals. Each of these tasks/threads send a “Heartbeat” message to the main task indicating that it is alive and functional by making use of a “named pipe”. The data that is being logged to the log file is being sent to the logger task via a Message queue. The system also has a functionality such that any remote client can inquire about the system status (log, temperature data in various units like Kelvin, Celsius, Fahrenheit and Lux data). This part of the system, which emulates a Remote Client – Server architecture makes use of Sockets to achieve a successful Inter Process Communication.

**BLOCK DIAGRAM**



**DESIGN CONSIDERATIONS**

**CHOICE OF IPC**

In this project, we used a total of three different mechanisms for inter process communication.

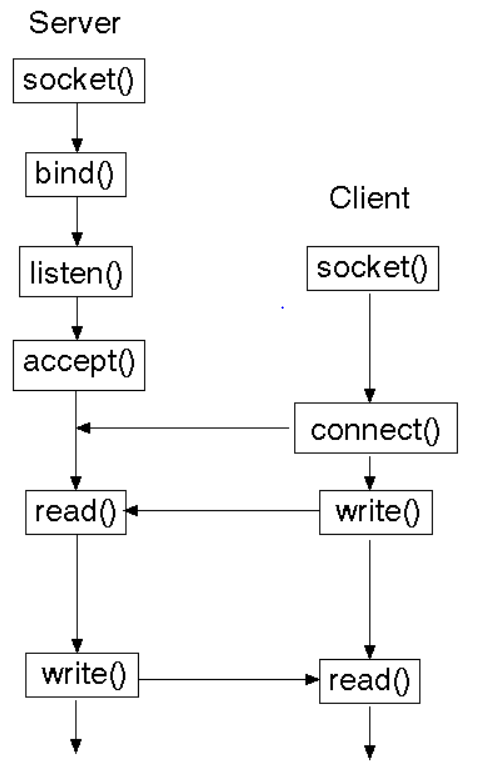
1. **Named pipes** – Used for sending heartbeat messages from each thread/task to the main thread.
2. **Message Queues** – Two separate message queues were implemented in order to communicate messages between different threads. One for sending information unique to each thread to the logger thread and the other is used to send information to the socket thread on occurrence of a Remote Client request.
3. **Sockets** – The system also has a feature where in an off-system host can create a socket connection with the remote socket task to inquire about the sensor data from the system.

**Purpose:**

A Named pipe is an extension to the traditional pipe concept on Unix. A traditional pipe is “unnamed” and lasts only as long as the process. A named pipe, however, can last as long as the system is up, beyond the life of the process [1]. In our project, we made use of a named pipe in order to communicate the heartbeat messages from each task/thread to the main task/thread. A major advantage of using a named pipe is that they provide a useful way to send one-line requests. Multiple processes can send requests through the same named pipe and each request is removed from the pipe as it is received [First-In-First-Out]. Output through a named pipe is more efficient that writing a complete response to an ordinary file, closing the file, and then informing the receiver that the results are available. The receiving process can read the result through a named pipe as soon as it is written [2]. Using a named pipe to send a multi-line message can prove to be a disadvantage unless we define a more complex protocol to control the message interleaving. As in our case, we are sending only a byte of data to indicate whether a task is alive or dead, named pipe prove to be a reliable inter process communication mechanism.

With respect to our implementation of the project, message queues are used to send information from temperature thread, light thread, socket thread to the logger thread. Each task sends data unique to their task to the logger thread every two seconds. A POSIX timer is used to trigger the timer handler associated with each task. As all the threads read and write at nearly the same time, some problems related to concurrency can occur. A message queue can guarantee the order of messages being sent by queuing them up. Another reason of using message queue is its ability to improve efficiency. It is much more efficient to insert multiple messages in the queue at a time instead of one at a time. Thus, message queues serve the purpose of communicating each thread’s data reliably to the logger thread.

An off-system host can create a socket connection with the remote socket task to inquire about the system status. Sockets are used such that the socket task can receive requests successfully from a remote client and send the desired data to the client. The system calls for input/output are open (), create (), close (), read (), write (). A figure below shows a time line of a typical scenario that takes place for a connection-orientated transfer. First, the server is started, then sometime later a client is started that connects to the server. The transfer of data can then take place between the server and the client.



**PROJECT IMPLEMENTATION**

This section of the report will provide a brief explanation of the implementation of the project.

As mentioned before, the project makes use of five threads/tasks in total. One being the main task which is responsible for spawning four child tasks, each of which performs a task independent of the other.

1. **MAIN TASK:**

The first process which is invoked at the start of the program is the main process. This particular process is responsible for numerous tasks. The very first task that the main process carries out is the initialization and creation of one single named pipe. This named pipe is responsible for conveying the heartbeat sent by each thread/task to the main process. As all the threads will try to send the heartbeat and messages at the same instance, mutex is used to achieve synchronization between these threads. All the signal handlers associated with each thread is configured. A POSIX timer is configured to trigger the timer handler associated with the main process every five seconds. In this timer handler, the main task checks if it received a heartbeat from each of the four threads. It then sends this information/error message to the logger task via a message queue. The logger task then logs this message into the log file. If the main task does not receive heartbeat from any of the threads, it indicates this using an on-board LED on the Beagle Bone Green. On reception of a SIGALRM signal, the main thread is terminated after gracefully terminating all the other threads and deleting all the timers.

A list of APIs called from the main task:

* **Mkfifo (const char \*pathname, mode\_t mode)** – Creates a FIFO file.

Param1 – mkfifo () function call makes a special file with name *pathname.*

Param2 – This mode indicates the permissions related to the queue.

* **Pthread\_mutex\_init ()** – Initializes the mutex.
* **Mq\_open ()** – Opens the message queue.
* **Mq\_send ()** – Sends a message on the message queue.
* **Pthread\_create () –** Creates a thread.
* **Pthread\_join () –** Waits for all the child threads to terminate.
* **Pthread\_mutex\_destroy () –** Destroys the mutex object.

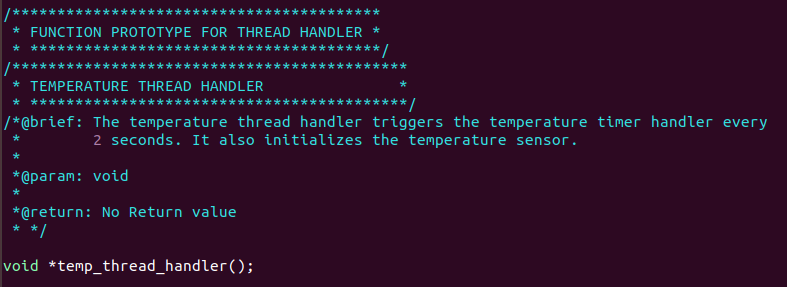
1. **TEMPERATURE TASK:**

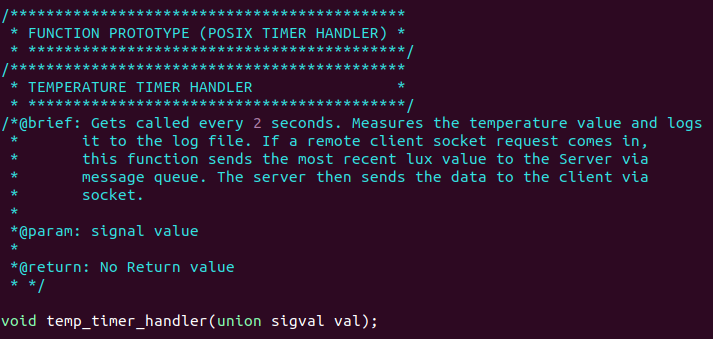
The temperature task is responsible for acquiring data from the temperature sensor at regular intervals. A POSIX timer is setup right after the creation of the temperature thread. The timer is configured such that the timer handler is triggered every two seconds. In the timer handler, two tasks are performed. One, the task reads the temperature data from the sensor using the I2C protocol. Second, the temperature task sends a heartbeat to the main task indicating that it is still alive. Apart from performing the abovementioned tasks, the temperature task also checks if it has received any request from the socket task. If a request from socket task has been received, temperature is sent in either Celsius, Fahrenheit or Kelvin as per the received request via message queue.

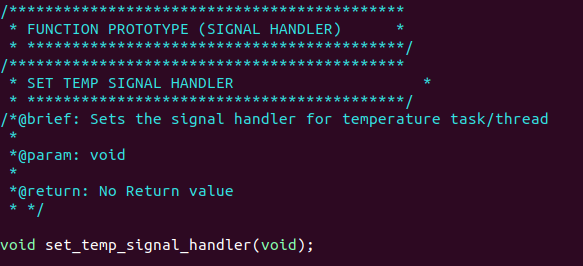
A list of APIs called from the temperature task:

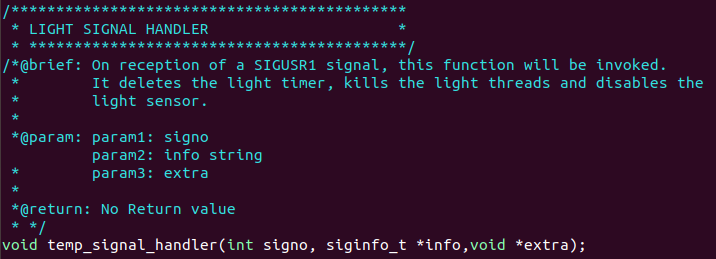
* **Mq\_send ()** – Sends a message on the message queue.
* **Write ()** – Sends a heartbeat message on the named pipe.
* **Pthread\_mutex\_lock** () – Locks the mutex object referenced by mutex.
* **Pthread\_mutex\_unlock** () – Unlocks the mutex object referenced by mutex.
* **Timer\_delete () –** Delete a POSIX per-process timer.
* **Pthread\_cancel () –** Sends a cancellation request to a thread.

A list of Functions used & their brief description:









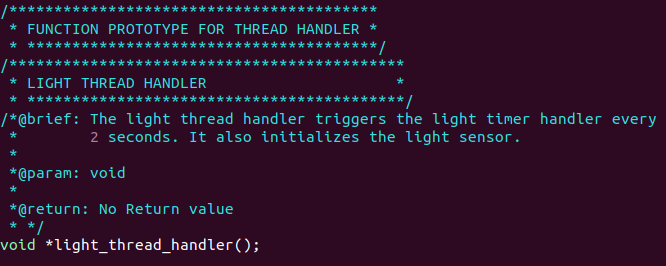
1. **LIGHT TASK:**

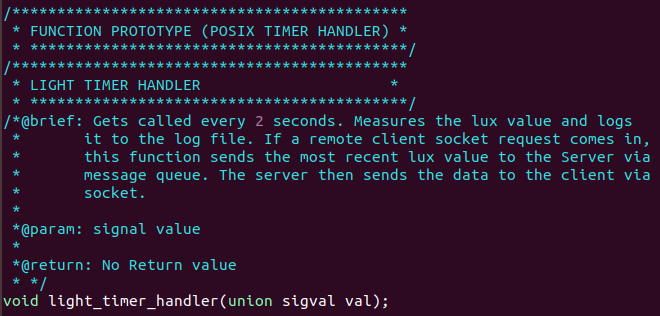
The light task is responsible for acquiring data from the light sensor at regular intervals. A POSIX timer is setup right after the creation of the light thread. The timer is configured such that the timer handler is triggered every two seconds. In the timer handler, three tasks are performed. One, the task reads the Lux data from the sensor using the I2C protocol. Second, the light task sends a heartbeat to the main task indicating that it is still alive. Third, it keeps a check on the current light state so that if it crosses a threshold, then the system indicates if the surrounding is Dark or Light. Apart from performing the abovementioned tasks, the light task also checks if it has received any request from the socket task. If a request from socket task has been received, Lux data is sent to the socket task via a message queue.

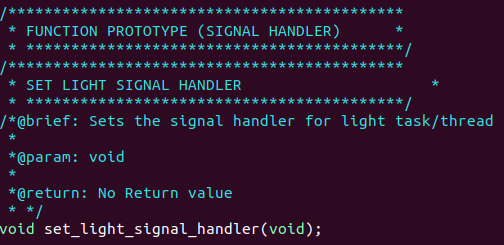
A list of APIs called from the Light task:

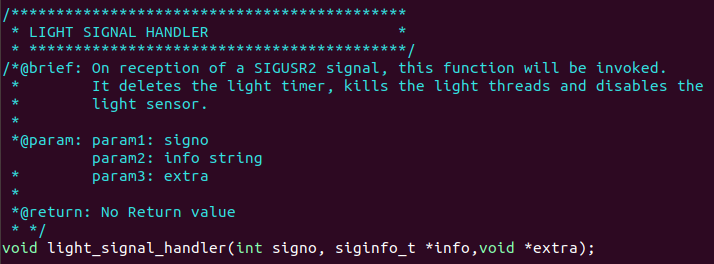
* **Mq\_send ()** – Sends a message on the message queue.
* **Write ()** – Sends a heartbeat message on the named pipe.
* **Pthread\_mutex\_lock** () – Locks the mutex object referenced by mutex.
* **Pthread\_mutex\_unlock** () – Unlocks the mutex object referenced by mutex.
* **Timer\_delete () –** Delete a POSIX per-process timer.
* **Pthread\_cancel () –** Sends a cancellation request to a thread.

A list of Functions used & their brief description:









1. **LOGGER TASK:**

**REFERENCES**

[1] <https://www.geeksforgeeks.org/named-pipe-fifo-example-c-program/>

[2]<https://documentation.progress.com/output/ua/OpenEdge_latest/index.html#page/dvpin/advantages-and-disadvantages-of-named-pipes.html>