KU LEUVEN, MASTER OF COMPUTER SCIENCE

Project in Software for Real-time and Embedded Systems

Benjamin Heath, Christina Athanasiadou, D'Angelo Villarico Fitzpatrick, Spyridon Chatzigeorgiou

Group 11

December 12, 2023

1 Introduction

This report is for the Software for Real-time and Embedded Systems project, done by Group 11. The project revolves around designing 2 BLE embedded systems that collect and transmit average temperature data, which has potential uses in an IoT application, such as in a smart home, as some form of thermostat. This report describes how we implemented some aspects of the sensor and receiver and is meant to be read along with the supplied code.

2 BLE communication

For ease of programming and due to time constraints, we went with a scan-advertise model for our Bluetooth communication scheme. The message is stored in the "manufacturing data" field of the advertisement sent out by the sensor. For each message to be sent, it is advertised continuously for 400ms by the sensor (controlled with the semaphore ***btReady***). The receiver scans for this data for a period of 30ms, leaving an interval of 60ms in between each scan. By having a much longer advertising period than the scanning period, the receiver is less likely to miss the sensor's messages, especially in a high-traffic environment. However, this comes at the cost of having the receiver potentially receive duplicate messages which needs to be accounted for.
All messages sent by the sensor follow a uniform format for ease of upgradability and to allow multiple receivers to listen to the sensor. The message is in the form of the string ***TNNXXYY***:

- **T**: The type of message received.
- **N**: The group number.
- **X**: The message number (or other value).
- **Y**: The message value.

Currently, there are only 2 message types used by the system: *T* (for average temperature), and *H* (for high temp warning).

- *T* uses the field *X* to transmit the message number, allowing for duplicate messages to be accounted for, and the field *Y* to list the average temperature.
- *H* uses the field *X* to transmit the message number, allowing for duplicate messages to be accounted for, and uses the field *Y* for the alert, setting it to *01* when the sensor detects a high temperature, and setting it to *00* when the sensor detects a low temperature. It will only send a message on the transition from low to high and high to low in the interest of power saving and to avoid sending too many BLE advertisements.

Upon the receiver detecting an advertisement, it first checks to see whether the group number is correct. If it is, it then proceeds to parse the message, first checking field *T* to see what type of message is being sent. When it is of type *T*, it then checks the message number (field *X*), storing the most recent one. If the message number matches the most recent message number, then it stops parsing there (as the message is a duplicate), if it is different then it prints out the average temperature stored in field *Y*. When it is of type *H* it simply checks the value of field *Y*. If it is *00* then it turns off the red LED, if it is *01* then it turns on the red LED. All message fields are processed as strings to avoid any complexity in converting strings to integers (as there is no function for this in Zephyr). However, if any more processing were to be done on the average temperature, then the string-to-integer conversion would be easy to implement.

3 Thread Synchronization in Sensor Node

The sensor nodes communicate via a mix of semaphores and message queues. First of all the message queue and the semaphores were declared. To be more specific four semaphores were created, named:

- ***hTempSemaphore*** with initial count 0 and count limit 1. Used by Thread A to signal to Thread C that the (sensor) LED should be turned on.
- ***lTempSemaphore*** with initial count 0 and count limit 1. Used by Thread A to signal to Thread C that the (sensor) LED should be turned off.
- ***hTempSemaphoreSend*** with initial count 0 and count limit 1. Used by Thread A to signal to Thread D that the receiver LED should be turned on.
- ***lTempSemaphoreSend*** with initial count 0 and count limit 1. Used by Thread A to signal to Thread D that the receiver LED should be turned off.
- ***avgTempNotify*** with initial count 0 and count limit 1. Used by Thread A to signal to Thread B that an average temperature should be sent.
- ***btReady*** with initial count 1 and count limit 1. Used to signal that the blue-tooth connection is free for transmission.

The message Queue is a crucial part of the implementation and is used for passing the temperature measurement data from thread A to thread B. It has a length of 10. Thread A has the highest priority and executes as follows:

1. Reads the temperature from the onboard sensor.
2. If the received temperature is above 30°c then the ***hTempSemaphore*** should increment, signaling to thread C that the led 0 should be turned on. Otherwise, it increments the ***lTempSemaphore***.
3. Puts the temperature measurements inside the Message Queue
4. Increments the ***avgTempNotify*** semaphore, which will be used to signal Thread B to send the average temperature measurement to the base station over Bluetooth

In this implementation, Thread A (named ***tReadTemperature***) is the master thread that operates until the 20 average temperatures have been sent, signaling and passing messages to the other three threads. The information is handled such that:

- Thread A reads the measurements,
- Thread B (named ***tSendTemperature***) calculates the average of the 10 measurements and sends the results over Bluetooth to the base station,
- Thread C (named ***tHighTempWarning***) manages the sensor node LED,
- and Thread D (named ***tSendHighTempWarning***) sends "H" type messages to the base station to signal that the base station LED should be turned on/off, as per the requirements of the open-ended functionality.

To be more thorough, Thread B is named ***tSendTemprature*** and it executes as follows:

1. It checks if the ***avgTempNotify*** semaphore is empty and if it is not, it takes all the measurements that were stored in the message queue by thread A.
2. It calculates the average of the 10 measurements and rounds it to the nearest integer because the Adafruit boards used do not support floating point arithmetic
3. It sends the average over the ***bluetooth advertise*** protocol as it was introduced in the BLE communication section.

Thread's C and D operate very similarly, with the key difference being that Thread C interacts with the sensor LED directly, whilst thread D has to advertise to the receiver that the LED should be turned on/off via the Bluetooth connection. They operate as so (in an infinite loop):

1. Try to take from the high-temperature semaphore (***hTempSemaphore*/ *hTempSemaphoreSend***) waiting until the semaphore value is incremented to 1.
2. Turn the LED on (directly for Thread C, by advertisements for Thread D).
3. Try to take from the low-temperature semaphore (***lTempSemaphore*/ *lTempSemaphoreSend***) waiting until the semaphore value is incremented to 1.
4. Turn the LED off (directly for Thread C, by advertisements for Thread D).

4 MINIMISING LATENCY FOR THE ALERT LED SYNCHRONISATION

In order to minimize latency between a temperature being recognized as high and a high-temperature warning message being sent, the separation of tasks into multiple threads and thread prioritization was carefully considered and implemented. First of all, it is essential that every thread performs a single simple task in order for the processing to take the shortest time possible. Therefore, the sending of the temperature warning and the LED powering on and off happen on different threads. This is done to ensure that no time will be wasted waiting for the Bluetooth to broadcast the message when the toggling of the LED can be performed on the sensor immediately. Attention should be given to the thread priorities as well. To be more specific:

- ***tReadTemprature*** (Thread A), priority 7
- ***tSendTemperature*** (Thread B), priority 4
- ***tHighTempWarning*** (Thread C). priority 5
- ***tSendHighTempreture*** (Thread D), priority 6

The thread with the highest priority is Thread B because it processes all the data that Thread A produces and it is essential that the message is sent to the receiver, every 10 seconds. Thread C has a higher priority than Thread D because Thread C does not send data, it only toggles the LED of the sensor, therefore it makes more sense that the fastest process executes first. Thread D will send the data over Bluetooth to the receiver and the receiver's LED will be toggled, merely some milliseconds after the LED of the sensor powers on/off. In practice, the delay is hardly noticeable. The receiver will always be connected to a laptop, therefore the creation of threads was found unnecessary as it would introduce more latency. That is because when the message arrives over Bluetooth, the thread that powers the LED would have to ask for CPU time from the scheduler, taking more time before it could actually toggle the LED. For those reasons, the receiver will always be scanning for messages, but it will not process any messages that are duplicates or that are not sent from the group 11 sensor.

5 APPROACH TO SENDER SLEEP/LOW POWER MODE AFTER REQUIRED 20 MESSAGES ARE SENT

Another required feature for the sender is to apply sleep/low power mode after sending the required 20 messages (one every 10 seconds). In order to do this, a counter (sCount) is implemented inside the ***tSendTemperature*** thread. This counter iterates through the sent messages and as soon as 20 messages with the average temperature are sent, it calls ***enterLowSleep()*** function, This function does the following actions

- turns off Bluetooth
- turns off the LEDs
- suspends all the threads with ***k_thread_suspend()***
- suspends the whole system with ***pm_system_suspend()***.

The ***tSendTemperature*** thread suspends itself since after 20 iterations it exits the while loop and stops.