Gianluca Scopelliti - r0778946
Benedek József Tasi - r0778958

SoRTES Assignment Report - Team 06

Code (.ino files) is attached in the *Scopelliti_Tasi.zip*, including all required external libraries (lib folder). The code is thoroughly commented to provide a detailed explanation supplementing this report.

1. System setup

For the completion of the assignment, several Arduino libraries were used. The most prominent ones are:

- **Arduino_FreeRTOS.h** [1][2]: Library including methods for programming real-time systems (tasks).
- **LoRa.h** (provided in class): contains function to realize communication between two LoRa microcontrollers.
- **EEPROM.h**: native Arduino library for providing EEPROM access.
- **EDB.h** [3]: a simple to use database handling lib. Combined with the EEPROM, it is used to create a Log struct, the contents of which can be written into the EEPROM.
- **semphr.h**: provides semaphores for coordinating access to a shared data structure, in our case, the database. This will ensure that no variable is written and read in the same time instant.
- **queue.h**: provides a queue buffer that allows for writing and reading data in a FIFO manner, to ensure that data is never disregarded or overwritten, even if the receiver and logger tasks are running asynchronously.
- **SleepMode.h**: our custom library, based on **LowPower.h** [4], defining required sleep functions, built around avr/sleep.h and avr/power.h. Used for disabling modules, and calling sleep().

2. Task architecture

To manage all the requirements in real-time, three FreeRTOS tasks are created. These are:

- **GatewayComm**: This handles communication between the gateway and the board: receiving the message, and sending the acknowledgement. This has the highest priority.
- **DatabaseHandler**: This task handles the database: writing and reading from it needs to be consistent at all times. In this project, incoming data is configured so we only attempt to add a record to the DB once every few seconds, and only read it in the end, but the system should nevertheless be prepared for handling more simultaneous requests.
- **SerialCommPC**: This monitors the native serial port, to detect any incoming messages (user inputs via the serial monitor, opened on the PC), and print the results of the specified database query. By default, this task is not operational when launching the system (it starts in a suspended state): it is turned on only after waking the processor from deep sleep.

In addition to these, the idle application hook (*vApplicationIdleHook()*) is also configured. This is the task running when no other tasks are executing, and here its role is to put the processor to sleep, if the need is signaled by a flag.
For providing access to suspending/resuming these tasks, TaskHandle_t variables are created for each.

3. Real-time communication

For communication with the gateway, the proper protocol needs to be known. As specified in the assignment description, the bandwidth is set to 869100000, to avoid crosstalk with other teams' interfaces. The LoRa interface needs to be initialized by *LoRa.setPins(<args>)*. For its other parameters, the values specified in class were used.
At the start of the system, the device actively waits for the first message. It continuously monitors for an incoming packet. Upon detection, the *onReceive(int)* function will check if the first 4 bytes of the message correspond to the defined header ("GW06" in our case), and ignore the message if it does not. If a matching header arrives, the rest of the message (the sleep duration) is read, and saved into the first field of a local Log struct after ASCII conversion into integer. The temperature is then measured immediately, and saved into the second field of the Log. This Log is then returned by the function, and the temperature value is immediately sent back to the gateway (*sendData(float)*) by the task, as an acknowledgment of the message. As we only want to receive 20 messages in this example, a counter is also increased to keep track of this.

Gianluca Scopelliti - r0778946
Benedek József Tasi - r0778958

The saved log, holding the sleep time and the measured temperature, is then loaded into a queue (*logQueue*), which will be consumed by the database for saving the record. Afterwards, the task is delayed for the duration specified in the message, minus the safe wake guard time (an experimentally determined safe 300 ms in our implementation).

4. Temperature measurement and sensor calibration

Temperature is measured using the internal diode of the processor, that produces temperature dependent voltage [5]. For this, an internal reference voltage of 1.1V and a mux is set, then the ADC has to be enabled. After the conversion, the raw value can be read from the ADCW register (that takes care of the ADCL and ADCH). As the Arduino's ADC is 10 bit, the raw value will be between 0 and 1023. This needs to be scaled in the software with a gain (multiplier) and offset (add or subtract), warranting the need for calibration [6].
To calibrate the sensor, two extreme values were measured. The first is normal operation (warm), and the second was holding a bag of ice against the board, and using a simple thermometer to verify the external temperature. It should be noted that the internal temperature sensor of the Arduino is inaccurate, especially with respect to the external temperature, so it can only be used for vague approximation. Generally, normal operating temperature was measured to be around 24-25 °C, and the bag of ice has a temperature of 1-2 °C. During these calibration measurements, the minimum and maximum measured raw value had a maximal difference of 51. From this, and the maximum temperature difference of 24°C, the gain can be calculated: (51/24)=~2.1. This result is still offseted by a significant amount: to reach the ~24°C measured during normal operation, a value of 190 is subtracted to finish the calibration.

5. Database handling

During system initialization, a database has to be created. The maximum size of this corresponds to the available space in the EEPROM. For this purpose, the EDB library was used. Since the EEPROM preserves the data upon shutting down (unplugging) and rebooting the system, the code is also prepared to enable opening an existing database, instead of overwriting it (uncomment *db.open(0)* in the code, instead of using *db.create(<args>)*). This task has a lower priority: additionally, in every iteration, it is delayed for one tick via *vTaskDelay()*, to allow the gateway communication to run smoothly.
Consistency must always be ensured when handling data, in order to avoid data loss, or writing waste information into the EEPROM. For this reason, a queue is used for transmitting data from the communication task to the database handler: any incoming data is immediately added to the queue (specified to have a maximum size of 10, as in this assignment, we only receive data once in every few seconds), instead of being stored in a single global variable that the reading function writes. This way, we can keep it from being overwritten if a new message arrives.
When the database handler task is running, it will constantly check if data was loaded into the *logQueue*: if yes, it will take the *SemaphoreHndl,* and append the record to the EEPROM DB. If the semaphore is taken, all other tasks are prevented from accessing the database. Afterwards, it will give the semaphore back. Since the assignment requires us to save data once every couple of seconds, we can assume that after saving a data, a sleep period will need to follow: for this, a flag is set after saving the data, based on the value of the GWcounter: if it is below 20, we need to go to idle mode (*idleFlag*), but after reaching 20 intervals, we need deep sleep (*powerDownFlag*).

6. Low power modes

A crucial task of the microcontroller is to reduce the power consumption when there are no operations to perform. For this purpose, two different sleep modes have been implemented.

Idle Mode: This is the sleep mode which the board enters when a beacon is received, and it has to wait until the next message is expected to be sent by the gateway. Since the board has to wait for a specific amount of time, a timer is needed to wake it up at the correct timestamp (specified by the message sent by the GW, and the guard time has to be subtracted). Considering that timers are disabled in all sleep modes except for the idle, this is the most suitable option to choose. Before entering in IDLE mode, the following modules are disabled: ADC, all the timers except for Timer3, SPI, USART1, TWI and USB. These are then enabled again after the timer is expired. Moreover, while the board is in idle mode, all created tasks are suspended (except for the Idle task).

Gianluca Scopelliti - r0778946
Benedek József Tasi - r0778958

Timer setup: We use Timer3 with a factor of 1024 as a prescaler for the clock (thus, timer clock runs at 8kHz); mode 7 was chosen, which means that the counter is continuously incremented until it reaches the value saved in the OCR3A register, then a timer interrupt is triggered, and the timer is reset.
Since OCR3A is a 16-bit register, we cannot store high values inside it (for example, if we wanted to set a timeout of 9 seconds we would have 9s * 8kHz = 73728 clock ticks, which is higher than 2^{16}-1); a workaround has been used, that is storing the number of ticks into an unsigned long variable called *cnt_left* (32 bits) and setting the timer multiple times if that value cannot be stored entirely into the OCR3A register. For this reason, the ISR routine *TIMER3_COMPA_vect* contains a conditional statement: if *cnt_left* has reached 0, it means that the time is over and the board has to wake up; if not, we just set the OCR3A register, decrement *cnt_left* and go to sleep again.
Note that, for the assignment, the timer is set more than once only for waiting times of more than 8 seconds, as the register would overflow in case of values larger than 8. In all the other cases, OCR3A can be set with the correct value, and thus the ISR routine is called only once.

Power Down Mode: After 20 beacons are received, or after the "3" command is sent through the serial monitor, the board has to enter ultra low power mode. We used the power down mode to achieve that, because it is the mode with the lowest available power consumption. Waking up from it is done via an external interrupt.
The setup is easier in this case: we used PIN 3 as the source of the interrupt, thus in the setup() function we set it to *INPUT_PULLUP* mode. Before going to sleep, an interrupt (LOW mode, as this is the only IT configuration that can be triggered in PWR_DOWN mode) is attached to it, with its corresponding handler. After that, the board enters in power down mode and it will continue sleeping until the interrupt is triggered (by connecting a jumper wire between the GND and WAKE_PIN pins). Then, the interrupt is detached and the execution continues.
The tasks are suspended before entering to power down mode; after the wake up, according to the functionality we want to provide, one or more of them are resumed. For instance, if we want to start receiving beacons again we have to resume all the tasks (and reset the GWcounter). If we only want to handle the serial communication, only the serialHandle task has to be resumed (default in the code).
However, we have noticed that the USB module does not work properly after the board wakes up from power down mode; we solved this by disabling it before going to sleep, and re-enabling it manually after: using this workaround the serial communication works, but the monitor has to be closed and reopened manually before it is possible to send any commands.

Sleep Modes and FreeRTOS: The FreeRTOS library uses the Watchdog Timer for its scheduling purposes. Specifically, such timer is set so that an interrupt is triggered every X milliseconds. This is needed for running the internal "clock" of the library, in order to manage correctly the tasks and the real-time paradigm.
For this reason, two consequences have been discovered during our tests:
1. The Watchdog Timer cannot be used by the developer. We cannot declare a ISR routine for it, because it would create conflicts with the FreeRTOS library.
2. Triggering the interrupt would inevitably wake up the board from sleep [7].
The former is not a problem for us, because we used a different timer for the idle mode. For the power down mode, instead, we do not need timers at all.
The latter is much more critical: if not properly handled, this could cause a malfunctioning of the sleep modes (waking up immediately), and as a consequence a high power consumption.
The following logic has been implemented to work around this issue: the call of the sleep mode has been put inside a while loop, checking a flag as a condition. This way, every time the watchdog interrupt is triggered the board can go to sleep again, until the flag is set to false by another routine. In particular, the flag is unset either when the timer has expired (for the Idle mode), or an external interrupt is triggered (for the Power Down mode).
Therefore, the board actually sleeps for around X milliseconds in sleep mode, then it is woken up for Y milliseconds, after that it is put to sleep again and the cycle restarts. By default, X is set to 15 milliseconds. Y, instead, is the time spent for serving the ISR routine of the watchdog timer, and for calling the sleep function again. Thus, Y is only the time spent to execute a very limited number of instructions, and it is much lower than X

(Y<<X). The power consumption is thus lowered successfully; for better results, X can be increased, paying attention to modify the real-time constraints accordingly.
For debugging purposes, serial prints were added after each sleep command, to monitor when the processor enters and leaves idle mode. However, as serial printing takes time, and it is halted by entering sleep mode, we would need short delays after each print, which could risk violating the real-time constraint. For this reason, we have decided to only print the messages received before each sleep interval for confirmation purposes, and a note signaling the end of receiving, when we enter deep sleep.

7. Serial commands

The role of this task is to handle incoming user commands, provided via the serial monitor. These can be:

- **1**: The latest data is read from the database, and printed to the serial monitor. In order to avoid reading from the database while the databasehandler is currently writing data into it, the *SemaphoreHndl* is taken after receiving the command, and given back once the *readRecord(<index>)* function has finished.
- **2**: Reads and prints all the values from the database. This is done by a simple for loop, with similar semaphore handling as the previous command.
- **3**: This simply sets the *powerDownFlag* to true, so that the idle application hook will send the process to deep sleep in its next iteration. As the idle task calls the same function every time, the interrupt is attached as usual, to enable waking the processor up.

By default, this task is only running after the processor wakes up from Power Down mode. By configuration.

8. Performance testing

Proper operation was validated by flashing the Gateway onto one of our microcontrollers: With the other board that runs our code, we have received every message, the Gateway has received every (20) ack sent by us immediately, and we woke up slightly before the gateway sent the next message, thus always catching it in time. By this, we can conclude that our implementation satisfies the specified real-time constraint.
To validate the sleep modes, a multimeter was used to measure current draw through the USB (a prepared (sectioned) USB cable). During normal operation, the board generally consumes ~16 mA. If FreeRTOS tasks are running, it can go up as high as ~25 mA. In Idle mode, this is reduced to around ~10-11 mA, and in Power Down mode, it is only ~5 mA. It is important to note that we measure the consumption of the entire board, not only the CPU: for example, a LED is constantly on when the board draws power, independently from the processor, and we also measure its draw. This means that (based on the general current draw of such SMD leds), at least about ~5-6 mA can be subtracted from our measurements to get the correct consumption for the processor: ~11 mA (normal), ~19 mA (FreeRTOS tasks), ~5 mA (Idle) and around ~0.5 mA (I<1mA) (Power Down) respectively.
After waking up by an interrupt (jumper wire placed between GND and WAKE_UP pins), the system reacts correctly to all commands: both the latest message (1) and the entire database (2) can be printed properly, meaning that all received messages and measured temperatures were stored correctly in the EEPROM database, and our synchronization efforts succeeded in keeping the database consistent. The board can be put back into Power Down mode by command 3, and woken again by the interrupt anytime afterwards.

9. Library sources and other references

[1] Arduino FreeRTOS library: https://github.com/feilipu/Arduino_FreeRTOS_Library
[2] FreeRTOS documentation: https://www.freertos.org/index.html
[3] Arduino EDB library: https://github.com/henla464/arduino-edb
[4] Arduino LowPower library: https://github.com/LowPowerLab/LowPower
[5] Arduino Internal Temperature Measurement: http://www.theorycircuit.com/arduino-internal-temperature-sensor/
[6] Arduino temperature sensor calibration: http://www.electronhacks.com/2014/04/how-to-scale-a-thermistor-for-an-arduino/?fbclid=IwAR3lcXMeTWF9YdoXaylKIqITElQ1zLh9l57lngAGdYMLg8OEpwqBG80tGc4
[7] Arduino FreeRTOS and sleep: https://create.arduino.cc/projecthub/feilipu/battery-powered-arduino-applications-through-freertos-3b7401?fbclid=IwAR1GNQRLYzUUsEDewU0Z79eTuI0aEl8eGNvxoi9NnPLOV9idVuIE6ymYErI