# Software for Real-time and Embedded Systems: Project

Jesse Geens, r0661523

December 2020

## 1 System Overview

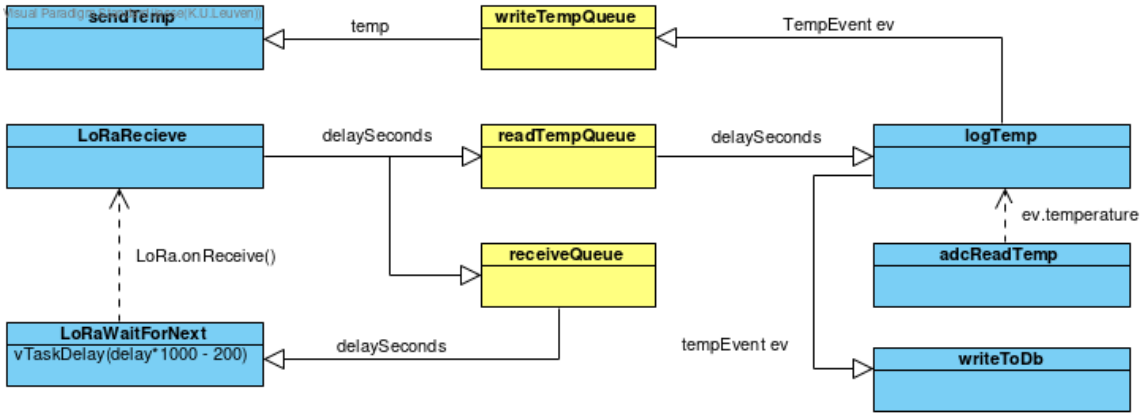
### 1.1 Tasks and queues

In order to be able to harness FreeRTOS' mulititasking capabilities, I have divided the main parts of the software into five seperate `Tasks`, detailed below:

Tasks

| Task | Human readable name | Stack | Param | Prio | Handle |
|---|---|---|---|---|---|
| `xTaskCreate(ReadSerialComand,` | `"Read Serial",` | `128,` | `NULL,` | `2,` | `&readSerial);` |
| `xTaskCreate(logTemp,` | `"Write temp to db",` | `128,` | `NULL,` | `1,` | `NULL);` |
| `xTaskCreate(LoRaWaitForNext,` | `"Wait for next beacon",` | `128,` | `NULL,` | `3,` | `NULL);` |
| `xTaskCreate(sendTemp,` | `"Send temperature to GW",` | `128,` | `NULL,` | `1,` | `NULL);` |
| `xTaskCreate(enableDeepSleep,` | `"Turn on deep sleep",` | `128,` | `NULL,` | `0,` | `NULL);` |

I will discuss the implementation and responsibility of each of the tasks in the sections below. In order to facilitate Inter-Process Communication between the tasks, I have created three `Queues` for several steps in the process. The `receiveQueue` processes integers, namely the delay until the next beacon. The `readTempQueue` also receives these delays, but these are immediately processed by the `logTemp Task`. Finally, there is the `writeTempQueue`, which receives the temperature and submits it over the LoRa network. Below, this has been illustrated graphically.




Furthermore, there are some functions that do not run as seperate tasks but that are called from within a task. This includes reading the temperature from the ADC (`adcReadTemp()` and writing a `TempEvent` to the database (`writeToDb()`). These are also discussed in the sections below.

### 1.2 Design decisions

For matters of readability, I have split up the project among logical part in seperate .ino files, if they are in the correct folder the Arduino IDE will automatically merge them when uploading and neatly displays them in seperate tabs. Furthermore, I have decided to use GRASP-like rules for the tasks: each task has a responsibility that is clear and has a high cohesion (e.g. reading commands from the serial and executing them, waiting a specified delay before turning on the LoRa radio, enabling deep sleep mode, sending temperature to the GateWay, etc).

### 1.3 Current draw

Finally, some key numbers. At start-up, the board draws about 20mA of current. After the first beacon has arrived, the board uses about 17mA of current when the LoRa antenna is in use. In idle mode, this drops to about 5.5mA. In ultra-low power mode, I'm able to go as low as 0.16mA ($\approx$ 164µA).

## 2 Executing commands over Serial

I have created a seperate task, `ReadSerialCommand`, to handle reading tasks from the Serial console. One of the requirements of this is that this functionality is only available up until the first beacon is received. That's why I've created a `taskHandle` for this task. When the first beacon is received, a special function is run (`enableLowPowerNoSerial()`, which stops the task from running. The task keeps a loop running, and checks through `Serial.available()` if there is something available to be processed. This gets then converted to a `char`, and the right command will be executed.

Additionally, I have implemented two extra commands because I found them useful for debugging purposes. Entering `4` will erase the database, while `5` will print some basic debugging information.

## 3 Receiving beacons

I have two functions related to receiving beacons. On the one hand, there is `LoRaReceive(int packetSize)`. In the `setup()` function, I have added this function as a callback to handle receiving a beacon. This function thus does the literal receiving and parsing of the packets. When it has received the beacon, it converts the delay to an integer, and then posts this integer onto two `Queues`: `readTempQueue` and `receiveQueue`. Finally, it shuts down the LoRa antenna with `LoRa.end()`.

On the other hand, there is the `LoRaWaitForNext` function, which is run as a `Task`. This function receives the delay from the `receiveQueue`, then delays the function using FreeRTOS' `vTaskDelay`. I then start the antenna 200ms before the beacon is supposed to arrive. This task has the highest priority, ensuring that this task will always be executed when it needs to, so that the antenna is guaranteed to be ready by the time the beacon starts arriving.

## 4 Low Power mode (idle)

### 4.1 Low power at boot time

When the device boots, there is already some room to lower the power usage of the board. For that, there is the `enableLowPower()` function, which turns the ADC off, disables the watchdog timer and timers 1, 2 and 3, and finally disables the I2C interface since this is not used.

### 4.2 Low power mode after first beacon received

Then, when the first beacon has been received, there is some additional room to lower the power usage. First, I suspend the `ReadSerialCommand` task, since we no longer need to listen to the Serial interface. Then, using the `avr/power` library, I disable USART0, USART1 and USB. However, the `power_usb_disable()` function does not disable everything, so I run a few additional commands:

USB Disable

```
USBCON |=  (1 << FRZCLK);   // Freeze the USB Clock
PLLCSR &= ~(1 << PLLE);     // Disable the USB Clock (Phase Lock Loop)
USBCON &= ~(1 << USBE);    // Disable the USB Connection
```

Through this, I'm able to get idle power usage down to 5.5mA.

## 5 Database implementation and synchronization

When the `logTemp` task receives a new delay through the `readTempQueue`, it creates a new `TempEvent`: a `struct` containing both the delay and the temperature at that time. It will then proceed to read the current temperature from the ADC, and send this temperature on the `writeTempQueue` so that it can be submitted over LoRa. Finally, this task will write the temperature to the database.

The database is implemented using the `EEPROM` library. It has the following layout in memory:

Memory layout

```
| 0         | 2       | 4     | 8     | ... | 84
| curr_addr | counter | temp0 | temp1 | ... | temp19
```

I keep track of two variables to keep the database consistent: `curr_addr`, which holds the current address to write to, and `counter`, which ranges from 0 to 20 and keeps track of how many elements are currently stored in the database. If the database is full (more than 20 beacons are being stored), the database wraps around (the 21st beacon will be stored at the location of temp0). Writing to the database involves the following steps: first, the database semaphore is taken. Then, we write the temp event to the EEPROM at address `curr_addr`. We check if the counter is less than 20 and increment it if necessary. Then we update `curr_addr`, taking the wrap-around into account. Finally, we write the updated `curr_addr` to the EEPROM, and give the semaphore. Using the semaphore ensures that atmost one instanciation of this function can concurrently update the database, thereby guaranteeing the consistency.

## 6 Reading temperature and calibrating

An important step in the process is getting a valid temperature measurement. For this, I have mostly referred to the datasheet of the LoRa32u4's processor, namely the Atmel ATMega32u4. It has an on-chip temperature sensor that can be read through the A/D Converter (ADC). Using the `MUX[5..0]` bits in the `ADMUX` register, we can enable the temperature sensor. We also must select the internal 2.56V voltage as a reference source. As per the datasheet, two conversions are required before a correct value is obtained, so we discard the first one. I check the ADC Control and Status Register A (ADCSRA) to check when the conversion has finished with the ADC Start Conversion bit (ADSC). Because we use a 16-bit integer, we read the output from two registers: ADC Low (ADCL) and ADC High (ADCH) for respectively the lower- and higher-order bits. Then we merge these together into our 16-bit integer. Finally, we turn the ADC off to save power.

# 7 Deep sleep

Finally, when enough (20) beacons have been received, the board will go in to deep sleep mode, utitlising only 0.16mA of power. Launching the sleep mode is done if two conditions are met: the number of beacons received is at least twenty (held in a counter called `beaconCount`), and a dedicated semaphore is released. This semaphore is created but never released upon creation. When the receiving LoRa function detects that the minimum amount of beacons is received, it will release the semaphore allowing the board to go into sleep mode. Conversely, when command '3' is executed, the semaphore is released and the counter is set to 20.

When we are finally allowed to enter sleep mode, we first use the `avr/power` library to disable all timers, USART0 and USART1, the ADC, and the USB, I2C and SPI interfaces. We also disable the analog comparator, and finally we enter the sleep mode `SLEEP_MODE_POWER_DOWN`. Interrupts have been disabled here since this is a critical section.

# References


[1] Atmel ATMega32u4 Datasheet, `https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7766-8-bit-AVR-ATmega16U4-32U4_Datasheet.pdf`, 04 12 2020

[2] Power saving techniques on the Atmega32u4, Martin Harizanov, `https://harizanov.com/2013/02/power-saving-techniques-on-the-atmega32u4/`, 04 12 2020

[3] ATMega32U4 on-chip temperature sensor, `https://www.avrfreaks.net/forum/teensy-atmega32u4-chip-temperature-sensor-0`, 04 12 2020