MCU Deep Sleep

SODAQ

Introduction

Electronics require electric energy to function. For most uses, it is as simple as plugging a device into a socket in the wall. For other scenarios, this energy is not readily accessible, and must be accumulated through a generator; with solar panels being a commonly chosen option.

In a grid-connected situation a certain price is charged for the power used, usually expressed in a price per kWh (kilo-Watt-hour). Devices connected to the grid, like household electronics, then use a certain capacity per hour, which can be seen in the electricity bill. For this reason, standby modes have been introduced. By regulation in the EU, these devices do not require more than one Watt in this mode, which, compared to the device turned fully on, is insignificant, and is more user-friendly (quick to respond or activate) compared to a fully shutdown device.

In an off-the-grid situation, electric power cannot be assumed to be generated at all times. This requires a battery to store excess energy for the situation where the generator is offline. Since these electronic devices can consume great amounts of energy (even when not actively following an instruction set), it is imperative that the device is set to an inactive state to preserve the battery capacity for the duration of the inactivity of the generator. This one of the reasons for the introduction of sleep modes for microprocessors, controllers and other chip-controlled devices.

Power Consumption in Electronics

What is electric energy; what is electric power? Without having to grasp the physics behind it, the concept is quite simple to understand. Where energy can be defined as the capacity to 'do' (or equivalently, the required capacity for something to be done), power is the energy consumption over time, measured in Watts (W). Power is thus the rate at which that capacity is spent. An alternative unit for power is horsepower, although it is generally only used for mechanical propulsive power.

When considering the electric energy and power, two more variables arise: the potential difference, or voltage (V), and charge flow rate, or current (A). Multiplying the Voltage with the current gives the power consumed by a component or circuit.

Although the SI unit of energy is Joule (J), more commonly used units are Watt-hours (Wh) and Ampère-hours (Ah). The name says it for both: a Watt-hour is the capacity to sustain a Watt of power for an hour; an Ampère-hour is one Ampère sustained for an hour. To get the 'Wh' rating of an Ah-rated battery, the Ah value must be multiplied by the nominal output voltage.

Power Reduction

In the situations where the battery power is insufficient for continuous operation of the device, some power savings must be made at the expense of operational functionality. This is referred to as standby, or sleep, and exists in multiple tiers. These tiers, or modes, range from an idle state, where the device is operating with its entire capacity without activity, to deep sleep, where the waking-up can take multiple thousands of clock cycles.

An example of these modes can be as simple as your laptop display turning off after a configurable amount of time. Before the display can be used again, an interrupt (keystroke, lid opening, or mouse click) must be performed before the display is sent the 'wake-up' command by the processor. Chances are that the processor itself was also in a sleep mode to increase energy savings and component lifetime.

Contrary to the previous example, deep sleep mode can also require switching off the RAM (random access memory, a very fast but volatile data storage used for the active programs). To prevent data loss, storing the most recent RAM state to non-volatile memory, such as flash memory is then necessary. This then allows the processor to turn off RAM and flash memory for energy savings without losing the instruction set, before going to sleep itself.

LoRaWAN and Sleep

Enter LoRaWAN, where devices with sensors and actuators attached need to exchange data over some distance with gateways through radio communication. This LoRa protocol is designed to function with low-power transmission at a moderate bit-rate and low duty-cycle. *low power and low duty-cycle? It sounds like sleep mode fits right in the picture!* Between the data communication cycles, the device idles. Why not sleep if we're not doing anything?

Internet of Things devices can be the cheapest tailored approach for sensing and actuating solutions. These devices can be placed in hard-to-reach places to provide insight in phenomena that are otherwise impossible or expensive to monitor: this is not limited to submerged or subterranean installations, where LoRaWAN can become the data transfer protocol of choice.

Since LoRa sets a maximum air-time duty cycle per sending device, in addition to the power requirement for sending data being extremely high compared to the rest of the device operations.

In order to operate longer periods of time without access to energy sources, low-power optimized operation is a design requirement: this is where sleep modes come into play: energy savings are created by putting the sensors to sleep (or turning them off entirely), followed by the processor entering sleep mode. Current draw can then be limited from mA range to μA range (this increases the battery endurance by a hundredfold, upwards to a thousandfold).

To put this in perspective: a 250mAh battery, supplying a nominal voltage of 3.7V (925mWh) can sustain a current draw of 15mA for 16 hours, whereas the same battery can sustain a current draw of 500μA for 500 hours. If the device is properly optimized, a current draw of approximately 30μA can be reached, which can run continuously for 8000 hours!

Deep Sleep Theory

How does deep sleep, in itself, work? A microprocessor processes tasks, mostly based upon a clock signal. Without this clock, a microprocessor comes to a standstill, and stops using power almost completely. Once a clock is stopped, however, the processor requires a clock signal to process a wake up call.

Deep Sleep Processes

Clock signal structure in a microprocessor starts at the oscillator; often a crystal. This is passed on (possibly divided) to the GCLK (Generic Clock Controller, consisting of Generic Clock Generators and Multiplexers), which then divided to provide a lower frequency and passed on to the CPU core, power manager and peripherals.

Why are clocks so important for power use? The answer lies with their power consumption, and the power consumption as a result of these clocks. For this reason, the clocks are usually rated at a current draw per MHz (or other frequency step). A 48 MHz clock then requires 24 times more power at 48 MHz, than it does when running at 2 MHz.

In order to then initiate sleep, all but the lowest frequency clock is shut down (usually a 32.768 kHz clock that is designed for low power, rather than high accuracy). This means that the processor comes to a standstill, with the interrupt controller monitoring the clock signal and interrupts.

Hence, the microprocessor shuts down almost entirely in deep sleep. Lighter sleep may be initiated where a higher frequency clock is kept active (1 MHz, for example).

In some processors, peripherals such as watchdog timers (WDT) and real-time clocks (RTC) are viable interrupts that can run off the low power clock signal. These are, besides the interrupt controller, low power clock and GCLK, the only components that remain awake.

Wake Up!

Now that the device is asleep, we must be able to wake it up again; otherwise there is no point to sleeping. Waking up the processor is generally dependent on the sleep mode. In almost all scenarios, however, waking up the processor requires an interrupt. This can be any interrupt on a physical pin, or an interrupt generated by the previously mentioned peripherals, namely the watchdog timer and real-time clocks.

Some microprocessors also support sleepwalking, where actions are performed at a lower frequency clock, but without fully waking the processor; a simple receive over serial and write to buffer would be an example action that can be performed in sleepwalking mode.

Different Platforms

Generally, all microprocessors are capable of some form of sleep mode, however this is largely dependent on the microprocessor, clock structure and the designer of the processor. The processor datasheet usually provides instructions on how to invoke sleep state, and how to wake it up again.

The SODAQ ExpLoRer used in the tutorial utilizes an Atmel SAMD central processor, which allows a custom sleep 'mask'.

A standard AVR-based processor, such as the Atmega 328P (as used in the Arduino Uno and Nano) are capable of power reduction modes. The named Atmega 328P datasheet, for example, claims an active power of 0.3 mA, a power-save mode (with RTC active) of 0.8 μA, and a full powered down leakage current of 0.1 μA.

Similarly, The Things Uno utilizes an AVR-based processor, namely the ATmega 32U4. Its datasheet specifically provides instructions on how to invoke sleep modes.

Single versus Multiple Processors

Multi-processor boards differ from single processor boards in that there is a single 'central' processor. In computing and electronics, this is generally referred to as the CPU.

Invoking the overall sleep mode, requires the CPU to send instructions to the other processors, detailing that a sleep mode must be invoked, before going to sleep itself. An example of such an external processor is the LoRa module, run by a controller loaded with firmware specifically for its operation; a certain sleep sequence and wake-up sequence are detailed in the list of many communication sentences that can be sent between the CPU and LoRa module.

Sometimes, however, invoking sleep mode can be as simple as asserting or de-asserting a (shared) pin (asserting means setting to logic high), but may also be a combination of the two.

ExpLoRer Tutorial

The SODAQ ExpLoRer is a board developed by SODAQ on behalf of Microchip, as a LoRa development and implementation board (hence the name 'explorer'). The ExpLoRer board was designed to have low power consumption. Before this is achieved, however, some of the components on the ExpLoRer need to be configured for low power mode or disabled.

Before continuing, ensure that the SODAQ SAMD Boards file is installed (through the Boards Manager in the Arduino IDE). Before it can be installed, the following URL must be added to "Additional Boards Manager URLs" in the IDE's preferences:

```
http://downloads.sodaq.net/package_sodaq_samd_index.json
```

Once this is done, it should be noted that when attempting to upload code to the ExpLoRer, the RESET button must be pressed twice within a second when powered on. The blue LED should turn on, indicating that the device is in boot-loader mode.

The ExpLoRer is loaded with a bootloader that allows you to upload code via USB at power-up. In the rare case that it fails, double pressing the RESET button within a second triggers the bootloader mode as well. The blue LED near the RESET button should turn on; an indicator to show that the device is in bootloader mode.

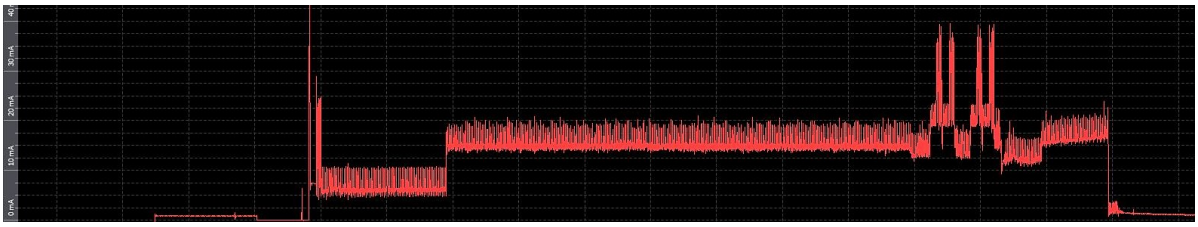

Figure 1: The quickest start-up for the ExpLoRer takes 13 seconds, consuming 13.3mA on average over that period!

Identifying the active components

Before starting to program the ExpLoRer for deep sleep, we must identify what components are on the board. Often, the manufacturer specifies some or all components used, or provides schematics of some sort. If that is not the case, the board may be inspected closely, or if required, under a microscope. Sometimes, chips are provided without identifying print, or have it sanded off to prevent corporate theft.

Components on the ExpLoRer

Luckily for us, SODAQ has made this quite easy for us, as they provide schematics with all the components and their data connections. This means that we know what to connect to, and how to connect to it. Combine this with the freely available data sheet and user guide, and (often) a library written to interface with that component: implementation is quite well supported.

From these schematics, it is clear that the Explorer has quite a number of components, of which some cannot be interfaced with, and will continue to draw current (the MCP73831 LiPo charger, for example).

On the other hand, we have a list of components that are interacted with by the microprocessor (the ATSAMD21), be it through serial ports, I^2C, or SPI:

- RN2483 (EU) or RN2903 (US): LoRa module,
- RN4871: Bluetooth 4.0 module,

- SST25PF040C: flash storage,
- ATECC508A: Crypto-authentication chip.

Inspecting all data sheets, it is clear that each of the chips listed have some form of sleep or power-down mode, which is implemented to limit the power consumption when the component is not required. Note that from this point forward, RN2483 is used to identify the LoRa module, as implementation is identical for the RN2903.

Component Sleep Mode

As previously mentioned, apart from the microprocessor (deep) sleep power reduction, the components, too, can be set to sleep for minimum power consumption.

ATSAMD21

Setting the microprocessor (the ATSAMD21) to deep sleep requires very few steps. Ensuring that the microprocessor wakes up again, some interrupts, such as a watchdog timer, must be set. To implement this, the Sodaq_wdt library must be installed through the Arduino Library manager. This watchdog timer as implemented draws an insignificant amount of current. Basic setup:

```
#include <Sodaq_wdt.h>

void setup()
{
    sodaq_wdt_enable(WDT_PERIOD_8X);        // set wdt at 8 second interval
    sodaq_wdt_reset();
    sodaq_wdt_safe_delay(1000);             // not required, but this should
                                            // be used instead of delay().

    SCB->SCR |= SCB_SCR_SLEEPDEEP_Msk;      // set sleep mode to deep sleep
}

void loop()
{
    sodaq_wdt_reset();          // Resets the wdt to prevent device resetting
    __WFI();                    // Wait For Interrupt - sleeps according to sleep mode
}
```

RN2483

The RN2483 LoRa module communicates to the microprocessor via serial: Serial2. Multiple Arduino-compatible libraries exist, but the one used is: *Sodaq_RN2483* (this can be found and installed through the Arduino library manager). Using the following piece of code, the library is imported, and the module is put to sleep.

```
#include <Sodaq_RN2483.h>
#define loraSerial Serial2

void setup()
{
    loraSerial.begin(LoRaBee.getDefaultBaudRate());
    LoRaBee.init(loraSerial, LORA_RESET);
    LoRaBee.sleep();
}
```

It's as simple as that! The last command sets the RN2483 to its lowest power mode, with a fast response to wake up. Note that the serial communication cannot be ended, as that wakes up the module, similarly, do not send multiple sleep commands in a row, as the module (while going to sleep) gets confused and wakes up instead.

RN4871

The RN4871 bluetooth module also communicates via serial: Serial1. Here too, multiple libraries are available. The used library is: *Microchip_RN487x*; **make sure that the library version is 1.0.2 or higher!** The code to required to put the module to sleep is quite simple:

```
#include <RN487x_BLE.h>
#define bleSerial Serial1

void setup()
{
    rn487xBle.hwInit();
    bleSerial.begin(rn487xBle.getDefaultBaudRate());
    rn487xBle.initBleStream(&bleSerial);

    // Enter dormant mode
    rn487xBle.enterCommandMode();
    rn487xBle.dormantMode();

    bleSerial.end();
}
```

Here, we do not use the low-power mode and hardware-sleep commands, since these commands set the module into low power mode, but the sleep mode power is not achieved. Rather, using dormantMode, the module is set to its absolute lowest power mode. To use it after this command is issued, requires a power cycle or by calling hwInit again.

Once dormant, it is safe to disconnect the serial line for additional power savings.

SST25PF040C

The SST25PF modules communicate through SPI. a different protocol with a different setup. These chips have been found to enter sleep automatically, or not; supposedly due to different firmware. Since a library is not written for this chip yet, a significant amount of code is required. The following code describes the actions required to set the flash to sleep, whether it went to sleep automatically or not:

```
#include <SPI.h>

void DFlashUltraDeepSleep()
{
    static const uint8_t SS_DFLASH  = 44 ;
    SPI.begin();

    // Initialise the CS pin for the data flash
    pinMode(SS_DFLASH, OUTPUT);
    digitalWrite(SS_DFLASH, HIGH);

    transmit(0xB9);

    SPI.end();      // close SPI and reset pins to low power state
    resetSPIPins();
}

void transmit(uint8_t val)
{
    SPISettings settings;
    digitalWrite(SS_DFLASH, LOW);
    SPI.beginTransaction(settings);

    SPI.transfer(val);

    SPI.endTransaction();
    digitalWrite(SS_DFLASH, HIGH);

    delayMicroseconds(1000);
}

void resetSPIPins()
{
    resetPin(MISO);
    resetPin(MOSI);
    resetPin(SCK);
    resetPin(SS_DFLASH);
}

void resetPin(uint8_t pin)
{
```

```
        PORT->Group[g_APinDescription[pin].ulPort].
        PINCFG[g_APinDescription[pin].ulPin].reg=(uint8_t)(0);

        PORT->Group[g_APinDescription[pin].ulPort].
        DIRCLR.reg = (uint32_t)(1<<g_APinDescription[pin].ulPin);

        PORT->Group[g_APinDescription[pin].ulPort].
        OUTCLR.reg = (uint32_t) (1<<g_APinDescription[pin].ulPin);

    }
```

It should be noted that this chip can remain in low-power for most LoRa related uses, since the maximum duty-cycle is limited per device, and the ATSAMD21 has a significant amount of flash storage on-chip.

The function resetPin resets the pin to the state it is in by default; this is programmed in the board file by SODAQ to be the lowest current leakage state for the microprocessor.

ATECC508A

The ATECC508A communicates via I^2C. Compared to the previous devices, the ATECC is automatically in sleep mode. The data sheet specifies that the sleep mode current draw is only guaranteed if the I^2C SCL (clock line) is either LOW or unconnected; not initializing the Wire library ensures this.

Setting the device to sleep once it has been used will be discussed where the chip is used.

Low Power Sketch

Combining all the above into a single program provides a very low power consumption: less than 15μA at 3.75V (approximately 56μW)!

Needless to say, some additional lines are added, specifically to disable and disconnect the USB port in order to prevent as much leakage current. as possible, it will be called at the end of the setup function.

```
#include <Sodaq_RN2483.h>
#include <Sodaq_wdt.h>
#include <SPI.h>
#include <RN487x_BLE.h>

#define bleSerial       Serial1
#define loraSerial      Serial2

void setup()
{
    // LoRa module connection and sleep
    loraSerial.begin(LoRaBee.getDefaultBaudRate());
    LoRaBee.init(loraSerial, LORA_RESET);
    LoRaBee.sleep();


    // BLE module sleep
    rn487xBle.hwInit();
    bleSerial.begin(rn487xBle.getDefaultBaudRate());
    rn487xBle.initBleStream(&bleSerial);
    rn487xBle.enterCommandMode();
    rn487xBle.dormantMode();
    bleSerial.end();


    // set FLASH to deep sleep & reset SPI pins for min. energy consumption
    DFlashUltraDeepSleep();


    SCB->SCR |= SCB_SCR_SLEEPDEEP_Msk;            // set sleep mode -> deep sleep
    USB->DEVICE.CTRLA.reg &= ~USB_CTRLA_ENABLE;   // Disable USB
}

void loop()
{
  __WFI();     // call again for repeat if interrupted
```

```
}


// FLASH chip sleep functions
void DFlashUltraDeepSleep()
{
  static const uint8_t SS_DFLASH  = 44 ;
  SPI.begin();
  pinMode(SS_DFLASH, OUTPUT);
  digitalWrite(SS_DFLASH, HIGH);
  transmit(0xB9);
  SPI.end();
  resetSPIPins();
}

void transmit(uint8_t val)
{
  SPISettings settings;
  digitalWrite(SS_DFLASH, LOW);
  SPI.beginTransaction(settings);
  SPI.transfer(val);
  SPI.endTransaction();
  digitalWrite(SS_DFLASH, HIGH);
  delayMicroseconds(1000);
}

void resetSPIPins()
{
  resetPin(MISO);
  resetPin(MOSI);
  resetPin(SCK);
  resetPin(SS_DFLASH);
}

void resetPin(uint8_t pin)
{
  PORT->Group[g_APinDescription[pin].ulPort].
  PINCFG[g_APinDescription[pin].ulPin].reg=(uint8_t)(0);
  PORT->Group[g_APinDescription[pin].ulPort].
  DIRCLR.reg = (uint32_t)(1<<g_APinDescription[pin].ulPin);
  PORT->Group[g_APinDescription[pin].ulPort].
  OUTCLR.reg = (uint32_t) (1<<g_APinDescription[pin].ulPin);
}
```

Using the program above, the current draw achieved can be recorded, and is visualized in Figure 2.

Sending over LoRa and Sleeping

Now that we have a basis, it is time to collect some data over a period of time, and send this over LoRa once a set is collected. Since an analog temperature sensor is included on the ExpLoRer, we will use that for this example. The data sheet of the thermistor (MCP9700AT) provides the required conversion formula.

In order to record this temperature, the following initialization and conversion come to mind:

```
uint8_t message[2];

void setup()
{
    pinMode(TEMP_SENSOR, INPUT);
    getTemperature();

    // now do something with contents of message
}

void getTemperature()
{
/*
```

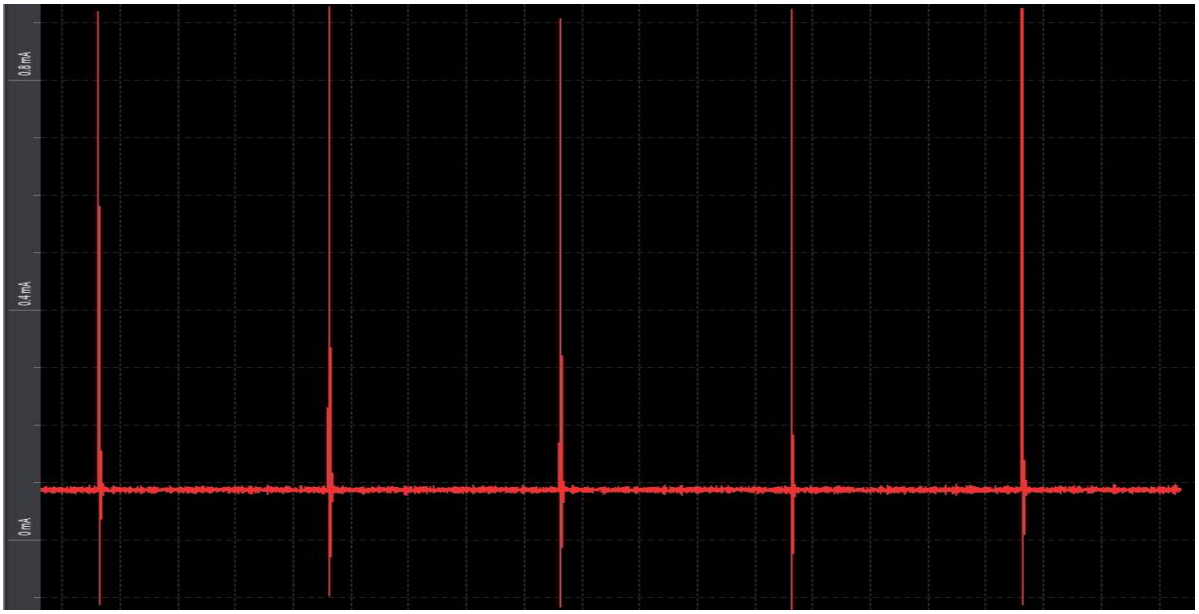

Figure 2: ExpLoRer Low Power. Post start-up 10 second snapshot, 88.5μA, 3.75V.

```
 *  Note: even though it's supposedly calibrated to have
 *        output 500mV at 0 deg C, the temperature is
 *        found to be off-set. A calibration is recommended
 */
    int int_temp;
    uint8_t negativeFlag;

    float mVolts = (float)analogRead(TEMP_SENSOR) * 3300.0 / 1024.0;
    float temp = (mVolts - 500.0) / 10.0;

    temp *= 100;

    if (temp < 0) negativeFlag = 0x80;
    else negativeFlag = 0x00;

    int_temp = abs((int) temp);

    message[0] = (int_temp >> 8) | negativeFlag;
    message[1] = int_temp & 0xFF;
}
```

This implementation returns a big-endian 2-byte signed integer, stored in the external array named *message*. The limits of this temperature range are $\pm$ 65.5 degrees Celsius.

The LoRa protocol specifies different communication settings that allow for increased communication link stability, that may be required due to a longer range for transmission, or obstacles. Typically used are Spreading Factor (SF) 7 and SF 12, for devices close to the gateway and those further away from the gateway, respectively.

With each step in Spreading Factor, the time-on-air is doubled, making the difference between SF7 and SF12 is a factor 32. The energy required for this transmission is then expected to scale similarly.

While the LoRa module is turned on, however, not only data is transmitted: before data transmission, a 'blank' signal is transmitted to prepare the gateway and provide timing of the signal. Additionally, the LoRa protocol has two receive windows, spread over 2 seconds, to receive data and acknowledgement.

This means that the on-time of the LoRa module is largely determined by the receive window. This can be seen in Table 1, which shows the total power consumption of an ExpLoRer when the LoRa module is awake and transmitting. Interesting to note are the differences between SF 7 and SF 12: SF 12 is only twice the period, and consumes only four times the energy: nowhere near the expected factor 32!

The measured power consumption in Table 1 is shown in Figure 3. Note the high peak, which is

the effective transmission period: the trailing high power draw is the module active and waiting for its receive windows.

Spreading Factor	4 Bytes sent	51 Bytes sent
Base deep sleep power	55.5μW	55.5μW
7	70mW, 2.4 sec	74mW, 2.5 sec
9	76mW, 2.6 sec	87mW, 2.8 sec
12	118mW, 3.7 sec	144mW, 5.2 sec

Table 1: ExpLoRer Power Consumption during LoRa transmission sequence.

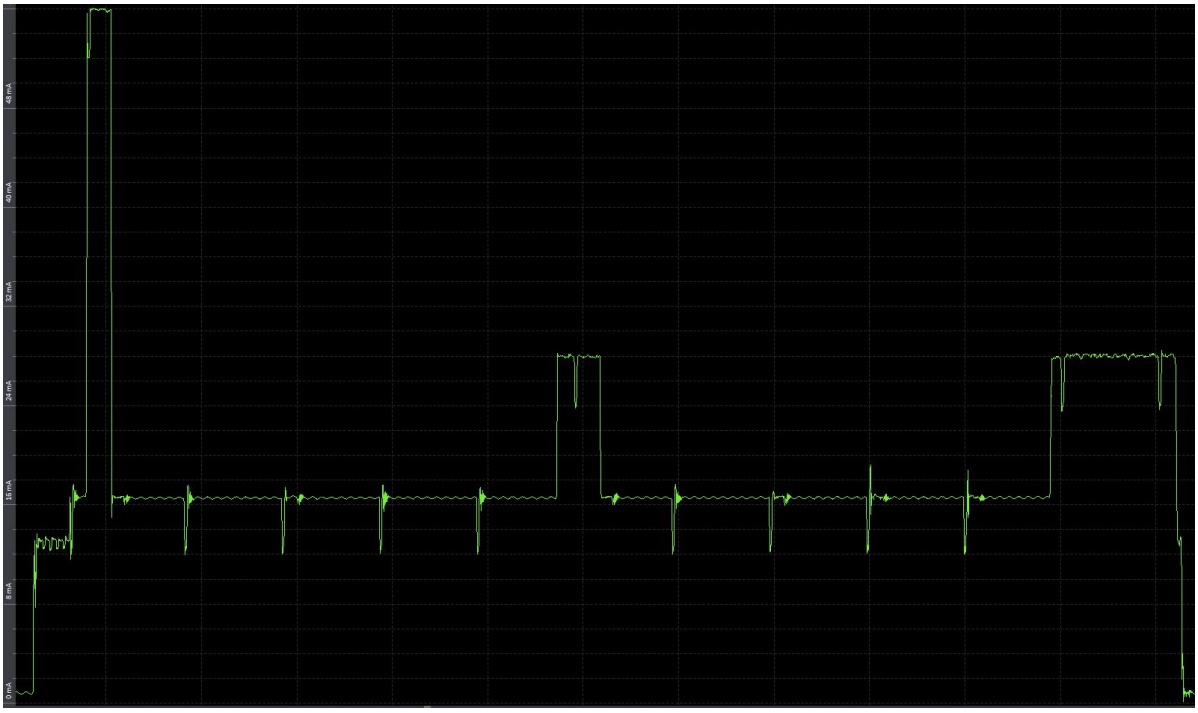

Figure 3: 4-byte transmission sequence at Spreading Factor 7.

Implementing everything through the following code:

```
#include <Arduino.h>
#include <Sodaq_RN2483.h>
#include <Sodaq_wdt.h>
#include <SPI.h>
#include <RN487x_BLE.h>
#include <RTCTimer.h>
#include <RTCZero.h>

#define CONSOLE_STREAM   SERIAL_PORT_MONITOR

#define debugSerial      SerialUSB       // debug on USB
#define bleSerial        Serial1         // Bluetooth module Serial
#define loraSerial       Serial2         // LoRa module Serial

#define LORA_BAUD        57600
#define DEBUG_BAUD       57600

#define NIBBLE_TO_HEX_CHAR(i) ((i <= 9) ? ('0' + i) : ('A' - 10 + i))
#define HIGH_NIBBLE(i) ((i >> 4) & 0x0F)
#define LOW_NIBBLE(i) (i & 0x0F)

RTCZero rtc;
RTCTimer timer;
```

```
volatile bool minuteFlag;

//
//                          setup your constants here!!
//
const uint8_t records_to_send = 2;           // set this to change the amount of records to send
const uint8_t record_every_x_minutes = 1;     // set this to the desired interval in minutes
const uint8_t spreading_factor = 7;           // set this to the desired LoRa spreading factor


// ***************************************************************************************
// LoRa communication setup!

//  true: use OTAA
// false: use ABP
bool OTAA = false;

// ABP setup (device address)
// USE YOUR OWN KEYS!
const uint8_t devAddr[4] =
{
  0x00, 0x00, 0x00, 0x00
};

// application session key
// USE YOUR OWN KEYS!
const uint8_t appSKey[16] =
{
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
};

// network session key
// USE YOUR OWN KEYS!
const uint8_t nwkSKey[16] =
{
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
};

// OTAA (device EUI)
// With using the GetHWEUI() function the HWEUI will be used
static uint8_t DevEUI[8]
{
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
};

const uint8_t AppEUI[8] =
{
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
};

const uint8_t AppKey[16] =
{
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
};

// ***************************************************************************************
// setup

bool LoRa_sleeps = false;
uint8_t message[records_to_send*2];

void setup()
{
  sodaq_wdt_enable(WDT_PERIOD_8X);       // Enable the wdt at maximum interval
  sodaq_wdt_reset();
  sodaq_wdt_safe_delay(5000);

  pinMode(TEMP_SENSOR, INPUT);

  initRtc();
```

```
  SCB->SCR |= SCB_SCR_SLEEPDEEP_Msk;      // sets SAMD sleep mode to deep sleep

  // networks
  loraSerial.begin(LoRaBee.getDefaultBaudRate());
  LoRaBee.init(loraSerial, LORA_RESET);
  setupLoRa();                                        // set the gears in motion

  initRtcTimer();                                     // timer interrupt > 1 minute interval

  sodaq_wdt_reset();

  SerialUSB.flush();
  SerialUSB.end();
  USBDevice.detach();
  USB->DEVICE.CTRLA.reg &= ~USB_CTRLA_ENABLE;         // Disable USB

  sleep_setup();
}

// ********************************************************************************************
// loop

// array for temperature we wish to send;
// here we're taking a 6-value (12 bytes) array to send at once
// intended to measure every n*60 seconds, send every x messages.

const uint8_t measurements_to_send = 6;
int temperature_array[measurements_to_send];
uint8_t list_iter = 0;     // variable to keep track of array index and when to send

void loop()
{
  if (sodaq_wdt_flag) {
    sodaq_wdt_reset();
    sodaq_wdt_flag = false;
  }

  if (minuteFlag) {
    timer.update();
    minuteFlag = false;
  }

  systemSleep();
}

// ********************************************************************************************
// Sleep commands

void BT_powerdown()
{
  rn487xBle.hwInit();
  bleSerial.begin(rn487xBle.getDefaultBaudRate());
  rn487xBle.initBleStream(&bleSerial);
  rn487xBle.enterCommandMode();
  rn487xBle.dormantMode();
  bleSerial.end();
}

void sleep_setup()
{
  // set FLASH to deep sleep & reset SPI pins for min. energy consumption
  DFlashUltraDeepSleep();

  sleep_LoRa();

  // RN4871 BT/BLE module sleep
  BT_powerdown();
}

void systemSleep()                    // Since only LoRa and MCU awake, only set those to sleep
```

```
{
  if (!LoRa_sleeps)                    // Skip if LoRa is asleep
  {
    sleep_LoRa();
  }

  noInterrupts();
  if (!(sodaq_wdt_flag || minuteFlag)) {
    interrupts();
    debugSerial.println("Sleeping");
    __WFI();              // SAMD sleep
  }
  interrupts();
}

void sleep_LoRa()
{
    loraSerial.flush();
    LoRaBee.sleep();
    LoRa_sleeps = true;
    sodaq_wdt_safe_delay(5);        // without this, it doesn't sleep.. don't know why
}

void wake_LoRa()
{
  LoRa_sleeps = false;
  LoRaBee.wakeUp();
}


// ********************************************************************************************
// SST25PF040C Flash functions (SPI)

void DFlashUltraDeepSleep()
{
  static const uint8_t SS_DFLASH  = 44 ;
  // SPI initialisation
  SPI.begin();

  // Initialise the CS pin for the data flash
  pinMode(SS_DFLASH, OUTPUT);
  digitalWrite(SS_DFLASH, HIGH);

  transmit(0xB9);

  SPI.end();

  // Resets the pins used
  resetSPIPins();
}

void transmit(uint8_t val)
{
  SPISettings settings;
  digitalWrite(SS_DFLASH, LOW);
  SPI.beginTransaction(settings);

  SPI.transfer(val);

  SPI.endTransaction();
  digitalWrite(SS_DFLASH, HIGH);

  delayMicroseconds(1000);
}

void resetSPIPins()
{
  resetPin(MISO);
  resetPin(MOSI);
  resetPin(SCK);
  resetPin(SS_DFLASH);
```

```
}

void resetPin(uint8_t pin)
{
  PORT->Group[g_APinDescription[pin].ulPort].
  PINCFG[g_APinDescription[pin].ulPin].reg=(uint8_t)(0);
  PORT->Group[g_APinDescription[pin].ulPort].
  DIRCLR.reg = (uint32_t)(1<<g_APinDescription[pin].ulPin);
  PORT->Group[g_APinDescription[pin].ulPort].
  OUTCLR.reg = (uint32_t) (1<<g_APinDescription[pin].ulPin);
}

// ************************************************************************************
// RN2483/RN2903 LoRa commands
// for RN2903: uncomment the setFsbChannels line.

void setupLoRa()
{
  getHWEUI();

  if(!OTAA){
    setupLoRaABP();    // ABP setup
  } else {
    setupLoRaOTAA();   // OTAA setup
  }
  // Uncomment the following line to for the RN2903 with the Actility Network.
  // For OTAA, update the DEFAULT_FSB in the library
  // LoRaBee.setFsbChannels(1);

  LoRaBee.setSpreadingFactor(spreading_factor);

}

void setupLoRaABP(){
  if (LoRaBee.initABP(loraSerial, devAddr, appSKey, nwkSKey, true))
  {
    debugSerial.println("Communication␣to␣LoRaBEE␣successful.");
  }
  else
  {
    debugSerial.println("Communication␣to␣LoRaBEE␣failed!");
  }
}

void setupLoRaOTAA(){

  if (LoRaBee.initOTA(loraSerial, DevEUI, AppEUI, AppKey, true))
  {
    debugSerial.println("Network␣connection␣successful.");
  }
  else
  {
    debugSerial.println("Network␣connection␣failed!");
  }
}

static void getHWEUI() // gets + stores HWEUI
{
  uint8_t len = LoRaBee.getHWEUI(DevEUI, sizeof(DevEUI));
}

void send_message(uint8_t*val, size_t val_size) {
  wake_LoRa();

  // since the debug port is not enabled in this example, the debug message is not printed
  switch (LoRaBee.send(1, (uint8_t*)val, val_size))     // send(port, payload, length)
  {
  case NoError:
    debugSerial.println("Successful");
    break;
  case NoResponse:
```

```
    debugSerial.println("No␣Response");
    break;
  case Timeout:
    debugSerial.println("Timeout.␣starting␣20␣second␣delay");
    delay(20000);
    break;
  case PayloadSizeError:
    debugSerial.println("Payload␣too␣large!");
    break;
  case InternalError:
    debugSerial.println("Internal␣Error;␣resetting␣module");
    setupLoRa();
    break;
  case Busy:
    debugSerial.println("LoRa␣module␣active");
    delay(10000);
    break;
  case NetworkFatalError:
    debugSerial.println("Network␣connection␣error;␣resetting␣module");
    setupLoRa();
    break;
  case NotConnected:
    debugSerial.println("Not␣connected;␣resetting␣module");
    setupLoRa();
    break;
  case NoAcknowledgment:
    debugSerial.println("No␣acknowledgement␣received");
    break;
  default:
    break;
  }
  sleep_LoRa();
}

// ********************************************************************************************
// Temperature Sensor functions
// output: 2-byte int centi-celcius (divide by 100 to type float for correct value).

void getTemperature()
{
  int int_temp;
  uint8_t negativeFlag;

  float mVolts = (float)analogRead(TEMP_SENSOR) * 3300.0 / 1024.0;
  float temp = (mVolts - 500.0) / 10.0;

  temp *= 100;

  if (temp < 0) negativeFlag = 0x80;
  else negativeFlag = 0x00;

  int_temp = abs((int) temp);

  message[list_iter*2]     = (int_temp >> 8) | negativeFlag;
  message[list_iter*2 + 1] = int_temp & 0xFF;
}



// ********************************************************************************************
// RTC functions

// Initializes the RTC
void initRtc()
{
    rtc.begin();

    // Schedule the wakeup interrupt for every minute
    // Alarm is triggered 1 cycle after match
    rtc.setAlarmSeconds(59);
    rtc.enableAlarm(RTCZero::MATCH_SS); // alarm every minute
```

```
    // Attach handler
    rtc.attachInterrupt(rtcAlarmHandler);

    // This sets it to 2000-01-01
    rtc.setEpoch(0);
}

// Runs every minute by the rtc alarm.
void rtcAlarmHandler()
{
    minuteFlag = true;
}

// Initializes the RTC Timer
void initRtcTimer()
{
  debugSerial.println("init rtc timer");
    timer.setNowCallback(getNow); // set how to get the current time
    timer.allowMultipleEvents();

    resetRtcTimerEvents();
}


void resetRtcTimerEvents()
{
    // Schedule the default fix event (if applicable)
    timer.every(record_every_x_minutes * 60, measureTemperature);
    debugSerial.println("event set");
}

// Returns current datetime in seconds since epoch
uint32_t getNow()
{
    return rtc.getEpoch();
}

// Default event parameter
void measureTemperature(uint32_t now)
{
  getTemperature();

  list_iter++;

  if (!(list_iter < records_to_send)) {
    send_message((uint8_t*) &message, sizeof(message));
    list_iter = 0;
  }
}
```