Technical University of Denmark

34373 - Microcontroller Development for Advanced IoT using Embedded C

Custom BMS Interface for DanSTAR Using BQ76942 and STM32 Nucleo-F302R8

Students:
Apolonia, Tomás (s242888)
Elkjær, Marius Mainz (s203836)

Professor:
Martin Nordal Petersen

Work distribution:

s203836	s242888
1, 2, 3.1, 3.4, 3.6, 3.7, 4, 5, 6	1, 3.2, 3.3, 3.5, 3.7, 4, 5, 6

Department of Electrical Engineering

May, 2025

Contents

List of Figures

List of Tables

1 Introduction

This project aims to develop a software interface for a Battery Management System (BMS) using an STM32 microcontroller to communicate with a dedicated BMS chip from Texas Instruments via SPI. The primary focus is on implementing core functionalities such as register configuration, real-time data monitoring through UART, fault detection and response mechanisms, and power optimization features. The BMS hardware has been developed by student s242888 for DanSTAR, DTU's student rocket team. The objective of this project is to complete the software development required to operate and validate the system.

The implemented features in this project are:

- **Register Configuration via SPI with CRC Handling**: Setting thresholds for protection, enabling/disabling charging and discharging, and controlling cell balancing, all with CRC verification as per chip requirements.
- **Real-Time Monitoring via UART**: Logging data for debugging and validation as the system runs.
- **Interrupt Handling with LED Indication**: Using interrupts to handle faults or warnings and give visual feedback.
- **Temperature-Based Protection**: Using an external temperature sensor to shut things down safely if the system overheats.
- **Power Rail Monitoring via High-Resolution ADC**: Using the STM32F3's 16-bit ADC to keep an eye on the logic supply for any voltage spikes, drift, or unexpected ripple.

As an extra, we are also looking into integrating low-power modes to make the system more energy-efficient and compliant with EU power standards for embedded devices.

2 System Overview

2.1 System Wiring and Component Integration

Figure 1 illustrates the hardware integration of the BMS test system using the STM32 Nucleo-F303RE development board. The key components include the BQ769x2 battery management IC, an LM35 analog temperature sensor, a voltage divider for battery voltage sensing, and a red LED used for visual fault indication.

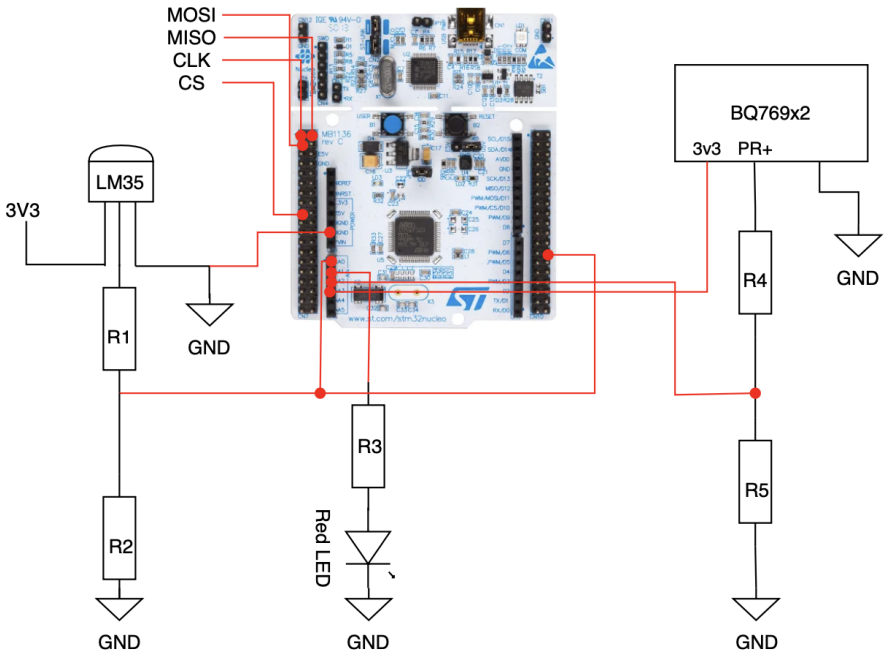

Figure 1: Diagram of the system setup.

The LM35 sensor is powered from the 3.3 V rail and outputs a voltage linearly proportional to temperature, which is read by the STM32 via pin A0 (ADC1_IN1). A resistive voltage divider composed of R4 and R5 is used to step down the battery voltage from the BMS's positive power rail output to a safe level that can be read via pin A1 (ADC1_IN5). The BQ769x2 communicates with the STM32 over SPI, connected to standard pins for MOSI, MISO, SCLK, and CS. The red LED is controlled via a GPIO pin through a series resistor (R3), and serves as a basic fault indicator that can be toggled by firmware during temperature overrun events or undervoltage conditions.

The corresponding CubeMX configuration can be seen in Figure 2.

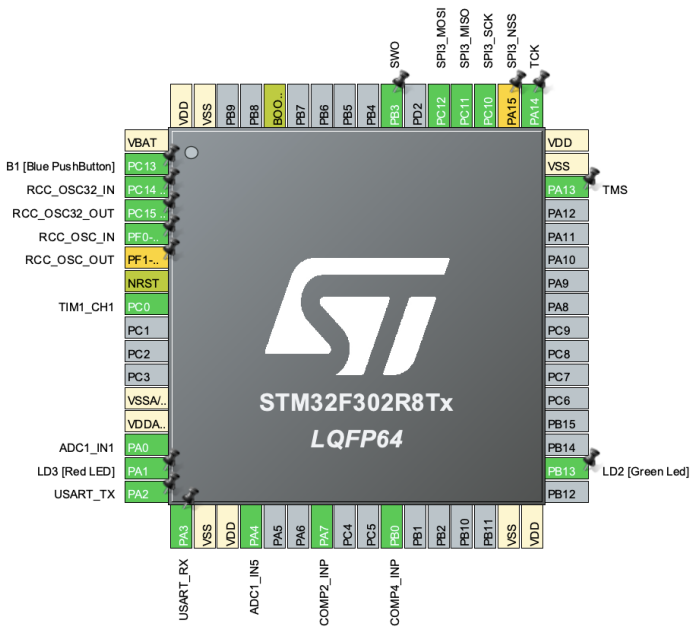

Figure 2: CubeMX pinout.

A picture of the implemented hardware setup can be seen in Figure 3, showing the STM32 Nucleo board connected to the LM35 temperature sensor, voltage divider, and BQ769x2. SPI is not connected in this setup, which we will discuss later in the report.

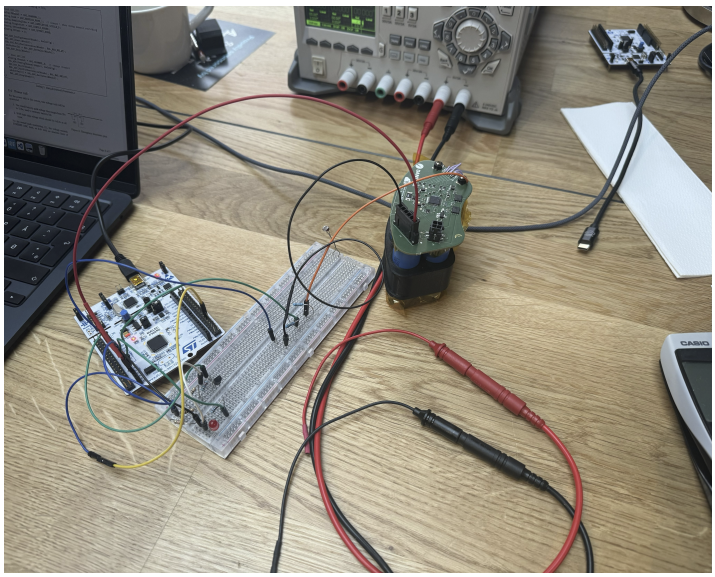

Figure 3: Overall setup including variable power supply to validate voltage variations

3 Implementation

3.1 Communication Protocol

The BQ769x2 battery management IC used in this project is configured for communication via the SPI protocol. This chip requires a valid CRC-8 checksum appended to each transaction, even when simply reading data. All SPI messages must follow a strict frame structure, and incorrect or missing CRC values will cause the BMS chip to silently ignore the command. Therefore, correct implementation of the communication protocol, including the CRC-8 calculation and timing constraints, is critical for successful operation. CRC is often used in communication protocols to reduce the chance of miscommunication or bit errors using an error correction check (CRC). For the BMS chip, CRC was enabled by default therefore CRC implementation and validation is needed.

3.2 Validating CRC

To validate the CRC response, an example from the BQ769x2 Software Development Guide (p.14) [2] was reproduced. The example is a read operation where the command sequence consists of the bytes `0xE6` (command) and `0x82` (address). According to the documentation, the expected CRC byte for this sequence is `0xBA`.

Rather than implementing CRC calculation from scratch, we noticed that the STM32 has an internal CRC peripheral that can be configured with the SPI settings. Following the datasheet we configured the CRC in 8-bit mode using the polynomial $x^8 + x^2 + x^1 + x^0$ (0x07), which matches the BQ769x2's CRC standard. This means that when two bytes are transmitted, there will automatically be transmitted a third byte calculated on the previous transmitted data.

The CRC was validated using a Saleae logic analyzer. As shown in Figure 4, the transmitted SPI message includes the correct CRC byte (`0xBA`) as the final byte on the MOSI line, confirming that the STM32's internal CRC engine is functioning as intended.

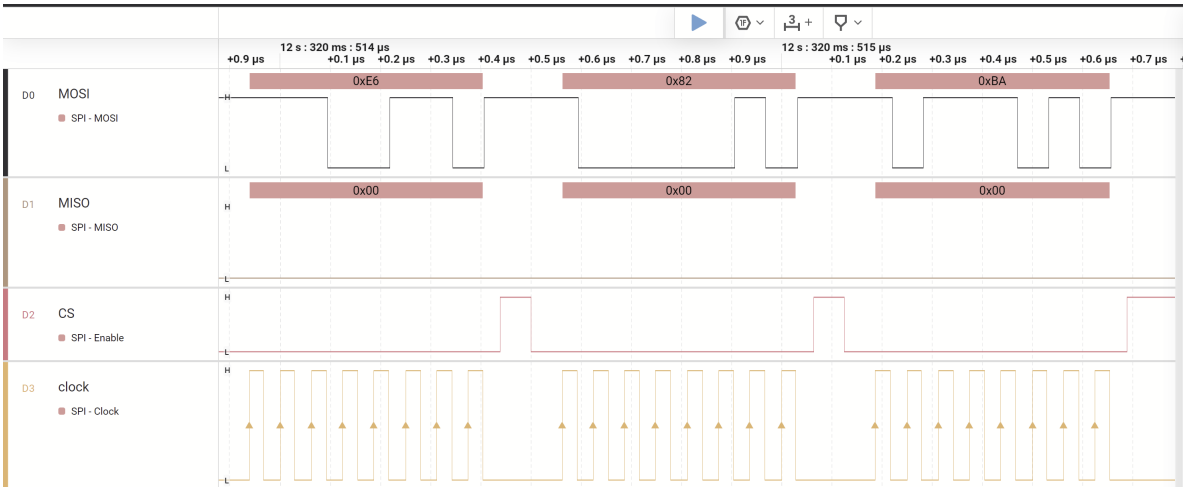

Figure 4: CRC validation

3.3 Issues programming with SPI

During programming with the BMS connected, no response was observed on the MISO line, initial hypothesis was a hardware issue, as both the code and hardware are experimental. The lack of response on Miso was found due to the SPI clock running 10x faster than the datasheet limit. According to Section 7.30 of the BQ76942 datasheet [1].

$$t_{SCK} > 500\,\text{nS} \tag{3.1}$$

$$f_{sck} < 2\,\text{MHz} \tag{3.2}$$

By prescaling the APB1 clock of STM32F3 allowed an SPI clock of 0.5MHz, well within the requirement. Initial response shown in Figure 5.

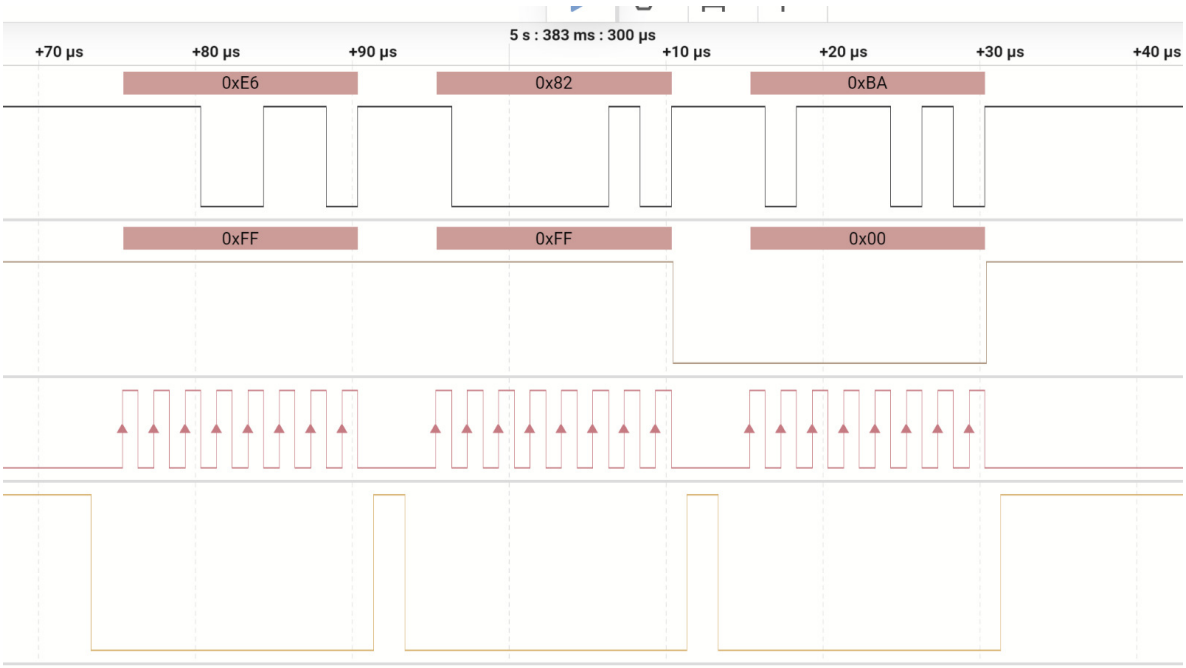

Figure 5: BMS initial response.

Unfortunately for the next few attempts, the MISO line always returned with 0xFFFF00. Which according to the datasheet [1]: *"if the device returns 0xFFFF00, then the previous transaction had not completed when the present transaction was sent, which may mean the previous transaction should be re-tried, or at least needs more time to complete"*. No matter how long we waited between communication transfers, the return was always the same.

We are unsure on the underlying issue but due to lack of time, we revised our project scope and how to achieve the goals without using the chip directly. We agreed that it was still possible to demonstrate the remaining peripherals using external components and circuitry. This meant using an LM35 temperature sensor and measuring power lines directly from the batteries, using a voltage divider circuit.

3.4 Dual ADC readings

Since we want to use the internal ADC for both the battery voltage monitoring and temperature monitoring with the LM35 sensor. To read the values from two different inputs using the same internal ADC, we made the helper function `scan_adc_channels()` that sets up the ADC for the first channel and then polls for conversion and stores the value into a buffer array of size 2 named `adcBuffer[2]`. The value from the first ADC is the temperature reading and it is stored at the first index of the buffer. The ADC channel is then configured to read the input from channel 5 of the ADC1, before polling and storing the value as for the previous channel. The function can be seen in Listing 1.

```c
void scan_adc_channels(void) {
    ADC_ChannelConfTypeDef sConfig = {0};

    // Read ADC_IN1
    sConfig.Channel = ADC_CHANNEL_1;
    sConfig.Rank = ADC_REGULAR_RANK_1;  // Always 1 when doing manual switching
    sConfig.SamplingTime = ADC_SAMPLETIME_1CYCLE_5;
    sConfig.SingleDiff = ADC_SINGLE_ENDED;
    sConfig.OffsetNumber = ADC_OFFSET_NONE;
    sConfig.Offset = 0;

    HAL_ADC_ConfigChannel(&hadc1, &sConfig);
    HAL_ADC_Start(&hadc1);
    HAL_ADC_PollForConversion(&hadc1, HAL_MAX_DELAY);
    adcBuffer[0] = HAL_ADC_GetValue(&hadc1);
    HAL_ADC_Stop(&hadc1);

    // Read ADC_IN5
    sConfig.Channel = ADC_CHANNEL_5;  // change channel
    HAL_ADC_ConfigChannel(&hadc1, &sConfig);
    HAL_ADC_Start(&hadc1);
    HAL_ADC_PollForConversion(&hadc1, HAL_MAX_DELAY);
    adcBuffer[1] = HAL_ADC_GetValue(&hadc1);
    HAL_ADC_Stop(&hadc1);
}
```

Listing 1: scan_adc_channels from main.c

3.5 Power rail

For the power rails in the system, two voltage rails will be monitored.

1. The overall battery pack voltage that ranges from 25v to 17.5v, used to estimate battery life.
2. BMS logic chip voltage which should be ≈3.3v at all times.

For the overall pack monitoring, (1), the voltage exceeds STM32F3 ADC limit, at 3.3v with an absolute limit of 3.8v ref 6.3.18 [5]. Therefore a voltage divider divides using $100k\Omega$ and $10k\Omega$. This steps down the voltage to (2.27 - 1.59)v.

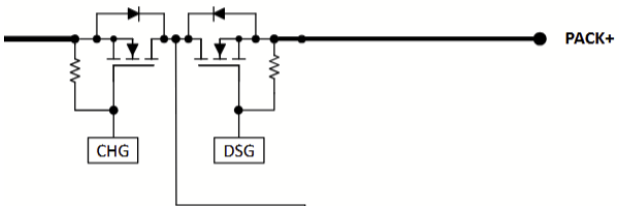

Figure 6: Emergency shutdown pins

By applying a simplified linear relationship between battery voltage and battery %, using the limits explained above. Equation was derived as:

$$y = mx + c \tag{3.3}$$

$$y = \frac{100}{2270 - 1590} \cdot x - \frac{100}{2270 - 1590} \cdot 1590 \tag{3.4}$$

$$y = 0.147x - 233.9 \tag{3.5}$$

Where y is battery % and x is voltage in mV. For this overall functionality check appendix function "float read_temperature(void)" for the full code.

For BMS chip logic voltage, (2), will be monitored encase it collapses, indicating a serious fault with the BMS chip. This is done with COM4 comparator, to allow for Interrupt functions. In the current code this simply indicates to the user a serious fault has occurred over UART2. For the DanSTAR rocket, this will pull CFETOFF, DFETOFF from the BMS chip high, causing PACK+ to be floating, Fig. 6. Significantly reducing the risk of safety hazard.

```
void COMP4_6_IRQHandler(void)
{
  char buf[32];
  snprintf(buf, sizeof(buf), "COMP4: Power Alert!\r\n");
  HAL_UART_Transmit(&huart2, (uint8_t*)buf, strlen(buf), HAL_MAX_DELAY);
  HAL_COMP_IRQHandler(&hcomp4);
}
```

Listing 2: COMP4 interrupt stm32f3xx_it.c

3.6 Temperature Fail Safe

As part of the system's thermal safety simulation, low-power functionality is used to represent a shutdown condition triggered by high temperature. While real-world BMS systems may disconnect loads or disable charging in such events, this project simulates that behavior by placing the microcontroller into STOP mode when the measured temperature exceeds a critical threshold. STOP mode halts the CPU and most peripheral clocks, significantly reducing power consumption while preserving the contents of RAM and peripheral registers [4]. In our implementation, this mode acts as a placeholder for a full system shutdown, effectively freezing system activity in response to a overheated condition.

The transition to STOP mode is handled by the function `enter_stop_mode()` using commands from the STM32 HAL library in Listing 3

```
void enter_stop_mode(void) {
    HAL_SuspendTick();
    HAL_PWR_EnterSTOPMode(PWR_LOWPOWERREGULATOR_ON, PWR_STOPENTRY_WFI);
}
```

Listing 3: Entering STOP mode in STM32 HAL

Wake-up is triggered by either a user button press or a comparator interrupt that detects when the temperature has returned to a safe level, both triggering `exit_stop_mode()` as seen in Listing 4.

```
void exit_stop_mode(void){
  HAL_ResumeTick();
  SystemClock_Config();  // Reinitialize clocks
}
```

Listing 4: Leaving STOP mode in STM32 HAL

3.6.1 ADC Readings

The function `read_adc_values()` works by first reading the raw adc value from ADC1_IN1 by the previous mentioned `scan_adc_channels()`, before processing the data to a compatible format. The output voltage of the LM35 scales linearly with temperature by 10mV per °C from (-40 to 125) °C [3]. The output voltage is 500mV at 0 deg C and thus the temperature can be calculated as:

$$temp°C = \frac{Vout - 500\,\text{mV}}{10\,\text{mV/C}} \tag{3.6}$$

Furthermore, we calculate the power rail monitoring in this function, as mentioned in the Power Rail section, to be able to output the calculated value to UART simultaneously. Lastly, the function returns the temperature in degrees.

3.6.2 Comparator and temperature detection

To enable automatic wake-up from STOP mode when the temperature drops back below the safety threshold, a comparator (COMP2) was configured to monitor the LM35 temperature sensor output. However, since the LM35 outputs an analog voltage directly proportional to temperature, a reference voltage was needed to define the comparison threshold.

To achieve this, a voltage divider circuit was used to reduce the output voltage, allowing comparison with the reference voltage. Specifically, the comparators built-in `VREFINT/2` reference option was used. This sets the comparison threshold at approximately 610 mV, corresponding to a temperature of around 10 °C when using the LM35 (which outputs 10 mV/°C with a 500 mV offset). Since this was too low for testing we added the resistor values so the trigger value will match with 22 degrees.

$$V_{22°\text{C}} = \frac{10\,\text{k}\Omega}{10\,\text{k}\Omega + 1.8\,\text{k}\Omega} \cdot 0.720\,\text{V} = 0.610\,\text{V} \tag{3.7}$$

The output of the LM35 is connected to the comparators inverting input, and `VREFINT/2` is internally routed to the non-inverting input. This configuration causes the comparator output to go high, triggering an interrupt when the temperature drops below the threshold, when the LM35 output falls below 610 mV at 22 degrees since it's configured for falling edge.

The key reason of this setup is that the comparator operates using a separate low-speed internal clock that remains active during STOP mode. This allows the STM32 to wake up from STOP mode via the comparator even when the main system clocks are disabled. The corresponding interrupt is shown in Listing 5 and is used to notify the system of a safe temperature condition.

```
void HAL_COMP_TriggerCallback(COMP_HandleTypeDef *hcomp)
{
    if (hcomp->Instance == COMP2)
    {
      exit_stop_mode();
      wake_from_temp = 1;
        wake_flag = 1;
    }
}
```

Listing 5: COMP2 interrupt main.c

The wake flags are used for the main forever loop.

3.7 The main loop

The main loop works by first reading the temperature and battery voltage values using the `read_adc_values()` function. It then checks whether the system has recently woken up from STOP mode. If so, it identifies the wake-up source, either a comparator interrupt triggered by a safe temperature condition or a manual button press and logs the event over UART.

```
if (wake_flag)
    {
      if (wake_from_temp)
      {
        wake_from_temp = 0;
        HAL_COMP_Stop_IT(&hcomp2);
        comp_flag = 0; // Mark comparator as stopped

        const char *msg = "Woke up: temperature dropped below threshold.\r\n";
        HAL_UART_Transmit(&huart2, (uint8_t *)msg, strlen(msg), HAL_MAX_DELAY);

        sleep_allowed = 0;
      }

      if (wake_from_button)
      {
        wake_from_button = 0;
        const char *msg = "Woke up: manual button press.\r\n";
        HAL_UART_Transmit(&huart2, (uint8_t *)msg, strlen(msg), HAL_MAX_DELAY);
```

```
        sleep_timer = HAL_GetTick() + SLEEP_TIME_MS;
      }

      wake_flag = 0;
    }
```

Listing 6: Main loop wake statements

Next, the system evaluates whether the measured temperature exceeds a defined safety threshold. If it does, a warning message is transmitted, indicator LEDs are activated, and the comparator is started (if not already running) to detect when the temperature drops back below the threshold. A timer is used to ensure that STOP mode is only entered after a sustained high-temperature period.

```
if (temp > TEMP_HIGH_THRESHOLD)
    {
      const char *msg = "Temp too high!\r\n";
      HAL_UART_Transmit(&huart2, (uint8_t *)msg, strlen(msg), HAL_MAX_DELAY);

      HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_SET);
      HAL_GPIO_WritePin(LD3_GPIO_Port, LD3_Pin, GPIO_PIN_SET);

      if (!comp_flag) // Only start if it's currently off
      {
        HAL_COMP_Start_IT(&hcomp2);
        const char *start_msg = "Starting COMP2..\r\n";
        HAL_UART_Transmit(&huart2, (uint8_t *)start_msg, strlen(start_msg),
            HAL_MAX_DELAY);
        comp_flag = 1;
      }

      if (HAL_GetTick() >= sleep_timer)
      {
        sleep_allowed = 1;
      }
      else
      {
        sleep_allowed = 0;
      }
    }
    else
    {
      HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_RESET);
      HAL_GPIO_WritePin(LD3_GPIO_Port, LD3_Pin, GPIO_PIN_RESET);

      if (comp_flag) // Only stop if it's currently on
      {
        HAL_COMP_Stop_IT(&hcomp2);
        const char *stop_msg = "Stopping COMP2..\r\n";
        HAL_UART_Transmit(&huart2, (uint8_t *)stop_msg, strlen(stop_msg),
            HAL_MAX_DELAY);
```

```
        comp_flag = 0;
      }

      sleep_allowed = 0;
    }
```

Listing 7: Main loop temperature handling

If the temperature is within a safe range, the comparator is stopped and the LEDs are turned off. When all conditions for sleep are met, the system prints a message and enters STOP mode using `enter_stop_mode()`. The loop includes a short delay at the end to ensure periodic sampling and to limit the system's polling rate.

```
if (sleep_allowed)
    {
      const char *msg = "Going back to sleep...\r\n";
      HAL_UART_Transmit(&huart2, (uint8_t *)msg, strlen(msg), HAL_MAX_DELAY);
      enter_stop_mode();
    }

    HAL_Delay(500); // Delay before next loop
```

Listing 8: Main loop sleep mode handling

4 Testing and verification

To verify the system's behavior during thermal fault conditions, UART output was logged while manually increasing the LM35 sensor temperature using a lighter. The system continuously monitored both temperature (via ADC1_IN1) and battery voltage (via ADC1_IN5), outputting values to the serial terminal.

Figure 7 shows the captured UART log. Each line includes:

- Raw ADC reading for temperature and battery input
- Scaled voltage values in millivolts.
- Computed temperature in degrees celsius
- Computed battery voltage and estimated battery percentage

```
ADC_IN1: 885, Temp_mV: 713 mV, Temp: 21 C --- ADC_IN5: 2263, Batt_mV: 1823 mV, Battery: 46.9%
ADC_IN1: 887, Temp_mV: 714 mV, Temp: 21 C --- ADC_IN5: 2261, Batt_mV: 1822 mV, Battery: 46.7%
ADC_IN1: 889, Temp_mV: 716 mV, Temp: 21 C --- ADC_IN5: 2254, Batt_mV: 1816 mV, Battery: 46.1%
ADC_IN1: 886, Temp_mV: 713 mV, Temp: 21 C --- ADC_IN5: 2239, Batt_mV: 1804 mV, Battery: 44.7%
ADC_IN1: 889, Temp_mV: 716 mV, Temp: 21 C --- ADC_IN5: 2259, Batt_mV: 1820 mV, Battery: 46.5%
ADC_IN1: 883, Temp_mV: 711 mV, Temp: 21 C --- ADC_IN5: 2238, Batt_mV: 1803 mV, Battery: 44.6%
ADC_IN1: 947, Temp_mV: 763 mV, Temp: 26 C --- ADC_IN5: 2245, Batt_mV: 1809 mV, Battery: 45.3%
Temp too high!
Starting COMP2..
Going back to sleep...
ADC_IN1: 889, Temp_mV: 716 mV, Temp: 21 C --- ADC_IN5: 2426, Batt_mV: 1955 mV, Battery: 61.5%
Woke up: temperature dropped below threshold.
ADC_IN1: 888, Temp_mV: 715 mV, Temp: 21 C --- ADC_IN5: 2245, Batt_mV: 1809 mV, Battery: 45.3%
ADC_IN1: 900, Temp_mV: 725 mV, Temp: 22 C --- ADC_IN5: 2338, Batt_mV: 1884 mV, Battery: 53.6%
ADC_IN1: 891, Temp_mV: 718 mV, Temp: 21 C --- ADC_IN5: 2248, Batt_mV: 1811 mV, Battery: 45.5%
ADC_IN1: 882, Temp_mV: 710 mV, Temp: 21 C --- ADC_IN5: 2259, Batt_mV: 1820 mV, Battery: 46.5%
ADC_IN1: 889, Temp_mV: 716 mV, Temp: 21 C --- ADC_IN5: 2259, Batt_mV: 1820 mV, Battery: 46.5%
ADC_IN1: 894, Temp_mV: 720 mV, Temp: 22 C --- ADC_IN5: 2254, Batt_mV: 1816 mV, Battery: 46.1%
ADC_IN1: 887, Temp_mV: 714 mV, Temp: 21 C --- ADC_IN5: 2274, Batt_mV: 1832 mV, Battery: 47.9%
ADC_IN1: 892, Temp_mV: 718 mV, Temp: 21 C --- ADC_IN5: 2277, Batt_mV: 1834 mV, Battery: 48.1%
ADC_IN1: 889, Temp_mV: 716 mV, Temp: 21 C --- ADC_IN5: 2244, Batt_mV: 1808 mV, Battery: 45.2%
ADC_IN1: 884, Temp_mV: 712 mV, Temp: 21 C --- ADC_IN5: 2214, Batt_mV: 1784 mV, Battery: 42.5%
ADC_IN1: 885, Temp_mV: 713 mV, Temp: 21 C --- ADC_IN5: 2224, Batt_mV: 1792 mV, Battery: 43.4%
ADC_IN1: 930, Temp_mV: 749 mV, Temp: 24 C --- ADC_IN5: 2211, Batt_mV: 1781 mV, Battery: 42.2%
ADC_IN1: 882, Temp_mV: 710 mV, Temp: 21 C --- ADC_IN5: 2214, Batt_mV: 1784 mV, Battery: 42.5%
ADC_IN1: 885, Temp_mV: 713 mV, Temp: 21 C --- ADC_IN5: 2209, Batt_mV: 1780 mV, Battery: 42.0%
ADC_IN1: 921, Temp_mV: 742 mV, Temp: 24 C --- ADC_IN5: 2196, Batt_mV: 1769 mV, Battery: 40.9%
ADC_IN1: 930, Temp_mV: 749 mV, Temp: 24 C --- ADC_IN5: 2206, Batt_mV: 1777 mV, Battery: 41.8%
ADC_IN1: 923, Temp_mV: 743 mV, Temp: 24 C --- ADC_IN5: 2213, Batt_mV: 1783 mV, Battery: 42.4%
ADC_IN1: 882, Temp_mV: 710 mV, Temp: 21 C --- ADC_IN5: 2205, Batt_mV: 1776 mV, Battery: 41.7%
ADC_IN1: 933, Temp_mV: 751 mV, Temp: 25 C --- ADC_IN5: 2210, Batt_mV: 1780 mV, Battery: 42.1%
Temp too high!
Starting COMP2..
Going back to sleep...
```

Figure 7: UART output when forcing the temperature above 24℃ with a lighter.

When the temperature exceeds approximately 24 °C, the system prints "`Temp too high!`", initializes the comparator, and enters STOP mode. This simulates a thermal shutdown. Upon cooling below 22° , an interrupt is triggered and the system resumes operation, indicated by "`Woke up: temperature dropped below threshold`".

During testing, adding multiple channels to the same ADC occasionally introduced voltage fluctuations that appeared to affect both measurements. This led to unreliable spikes in the temperature readings, causing the system to enter and exit STOP mode more frequently than intended. One likely contributing factor was the inclusion of a voltage divider on the sensor output, which also introduced a scaling factor of approximately 1.18 in the temperature conversion formula.

In hindsight, the ADC input should have been connected directly to the LM35 sensor output to preserve the direct output voltage for accurate temperature measurement. The voltage divider could instead have been only connected to the comparator input, since the comparator only checks against a fixed threshold and is not affected by scaling in the same way as the ADC calculations.

The detection of overheating worked as it should as well as entry and exit from STOP mode confirms that the fail safe mechanism is functioning, however the system was more stable when having a single ADC channel active. We also tried using DMA and different rankings on the ADC channels without luck, since the ADC channels measured the same value.

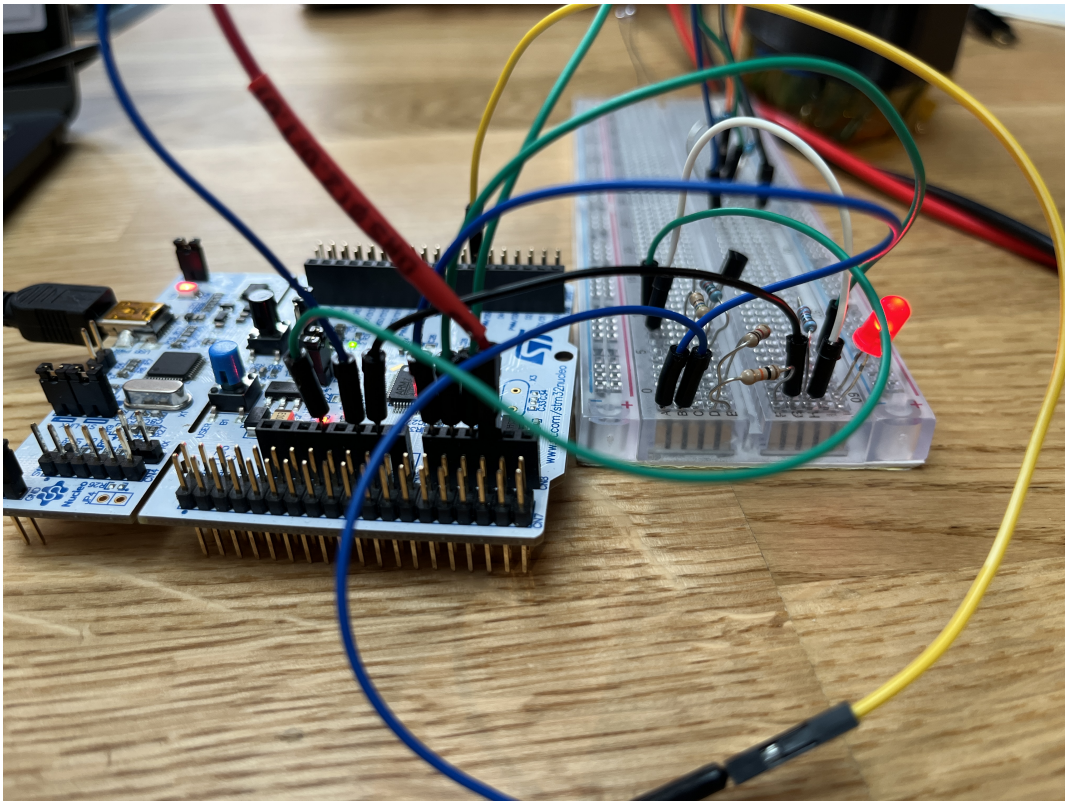

Figure 8: Alarm LED is indicating that the system is in stop mode.

Figure 8 shows the setup of the system where the red external and green internal led both light up, indicating that the temperature is too high.

5 Conclusion

This project partially demonstrated key features of a Battery Management System using the STM32 microcontroller peripherals, including temperature monitoring, power rail sensing, low-power STOP mode, and interrupt-based wake-up. Although full integration with the BQ769x2 BMS chip was not achieved due to persistent SPI communication issues, the core safety functionality was replicated using external components such as the LM35 temperature sensor and voltage divider circuits.

The system was able to detect overheating conditions, simulate shutdown behavior via STOP mode, and resume operation based on comparator and user input. These results confirm the correct configuration of ADC channels, low-power modes, and interrupt handling, providing a solid foundation for future work involving full BMS chip integration.

6 Future Work

While the core peripheral functionality was successfully demonstrated using external components, full integration with the BQ769x2 chip remains an important objective for future development. Resolving the SPI communication issue will require deeper investigation into SPI timing and chip initialization sequences.

Future work could involve:

- Implementing a complete startup sequence as recommended by TI's application notes
- Verifying CS line behavior and SPI frame alignment with a higher-resolution logic analyzer
- Testing alternative communication protocols, such as I^2C, if supported by the BMS configuration
- Integrating additional BMS features such as cell balancing, fault detection, and logging
- Use internal temperature sensor of the BMS.
- Monitor the output power consumption in STOP mode.

A Git Repository

The full project can be found in the linked git repository: `https://github.com/Mariusmainz/BMS2.git`

References

[1] Texas Instruments. *BQ76942 Datasheet*. URL: `https://www.ti.com/lit/ds/symlink/bq76942.pdf?ts=1746729356321&ref_url=https%253A%252F%252Fwww.ti.com%252Fproduct%252FBQ76942`.

[2] Texas Instruments. *BQ769x2 Software Development Guide*. URL: `https://www.ti.com/lit/an/sluaa11b/sluaa11b.pdf?ts=1746765225014`.

[3] Texas Instruments. *LM35 Datasheet*. URL: `https://www.ti.com/lit/ds/symlink/lm35.pdf`.

[4] STMicroelectronics. *Getting started with PWR*. 2025. URL: `https://wiki.st.com/stm32mcu/wiki/Getting_started_with_PWR#Stop0--Stop1--and_Stop2_modes`.

[5] STMicroelectronics. *stm32f303vc Datasheet*.