Hochschule Konstanz
Department of Computer Science

UC

AIN 6

Ubiquitous Computing

Wintersemester 24/25

Name: Tobias Sautter

Matr.-Nr.: 304790

Lab 3: IoT-cloud-integration

The goal of this project was to develop a complete IoT system using Arduino, Node-Red and the cloud platforms HiveMQ and Datacake. The tasks included synchronization of physical and virtual LEDs, development of a dashboard for real-time monitoring of temperature data, and integration of the MQTT protocol with HiveMQ and Datacake for data visualization.

Exercise 1: Synchronization of physical and virtual LEDs

This exercise consists of 2 tasks, where in the first task the objective was to cross connects the virtual and physical LEDs and Buttons, and the objective of the second Task was to synchronise the virtual and physical LEDs and Buttons

Task 1: Physical button controls the virtual LED

In the first subtask, an Arduino was programmed to send the state of a physical button in real time to a virtual LED on a Node-Red dashboard. When the button was pressed, the virtual LED lit up. The other way around, the physical LED on the Arduino could be controlled by activating the virtual LED in the dashboard. This required a serial communication between Arduino and Node-Red.

Issues:

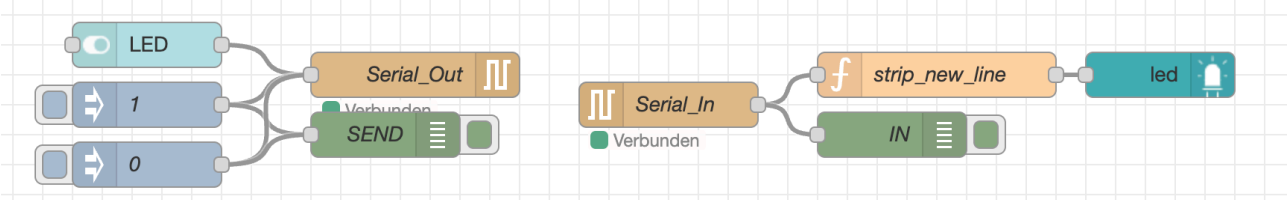
At the beginning of the implementation, controlling the LED via serial input did not work as expected. The reason for this was that the Arduino code terminates the sent messages with a line. This resulted in the received data not being processed correctly in the Node-Red flow.

Attempted solutions

The first solution was to remove the line break (\n) directly in the Node-Red serial input node. Although the corresponding setting was made, this did not work. For unknown reasons this method was not working.
The Arduino code was then modified by replacing the Serial.println() function with Serial.print(). In addition, the Serial Input node in Node-Red was configured to split messages into strings with a length of one character. This approach also failed. The most likely reason for this is that the Serial Input node waits for a terminator by default before processing a message.

Successful solution

Since the previous approaches did not produce the desired result, an additional function node in Node-Red was used to remove the line break (\n) from the received messages with a .trim() function. This method eventually was successful and the serial communication worked as intended.



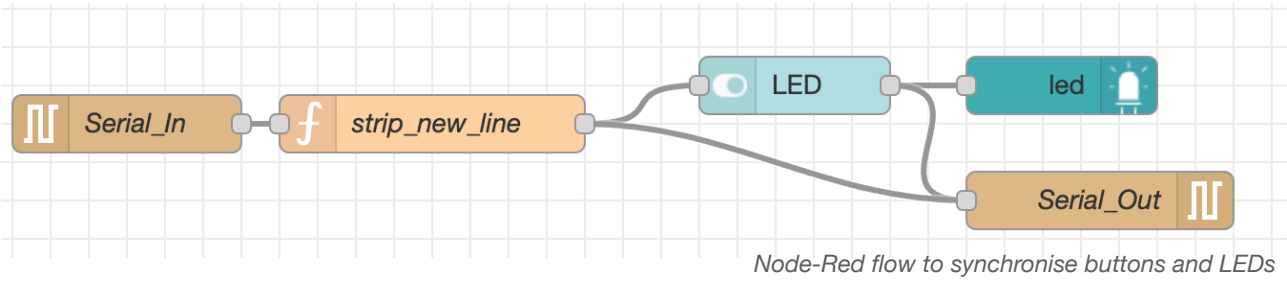
Node-Red flow to cross connect buttons and LEDs

Task 2: Synchronization of the physical and virtual LEDs

The focus of the second subtask was to achieve synchronization between the physical and virtual LEDs. Regardless of whether the LED was activated by the physical button or by the dashboard, both LEDs should always show the same state. The challenge was to ensure that changes in one system were adopted reliably and without delay in the other system.

Procedure:

The Node-Red flow was adjusted so that state changes of both LEDs were synchronized with each other. This required the integration of a feedback mechanism that transmitted the state of the LED from the Arduino back to Node-Red and vice versa. To accomplish this, a serial out node, connected to the virtual button was used to notify the Arduino if the virtual button was pressed. With this mechanism it was possible to only change the node-red flow, instead of also changing the Arduino code.



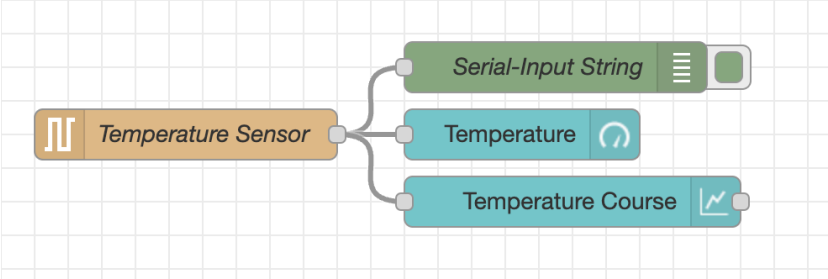
Node-Red flow to synchronise buttons and LEDs

```
const int BUTTON_PIN = 2;
const int LED_PIN = 13;
int oldButtonState;

void setup() {
  pinMode(LED_PIN, OUTPUT);
  pinMode(BUTTON_PIN , INPUT_PULLUP);
  Serial.begin(9600);
}

void loop() {
  int buttonState = !digitalRead(BUTTON_PIN);

  if(buttonState != oldButtonState) {
    oldButtonState = buttonState;
    Serial.println(buttonState);
  }


  if (Serial.available() > 0) {
    char command = Serial.read();
    if (command == '1') {
      digitalWrite(LED_PIN, HIGH);
    } else if (command == '0') {
      digitalWrite(LED_PIN, LOW);
    }
  }

  delay(100);
}
```

Arduino code to turn the LEDs on or off

Exercise 2: Temperature monitoring dashboard

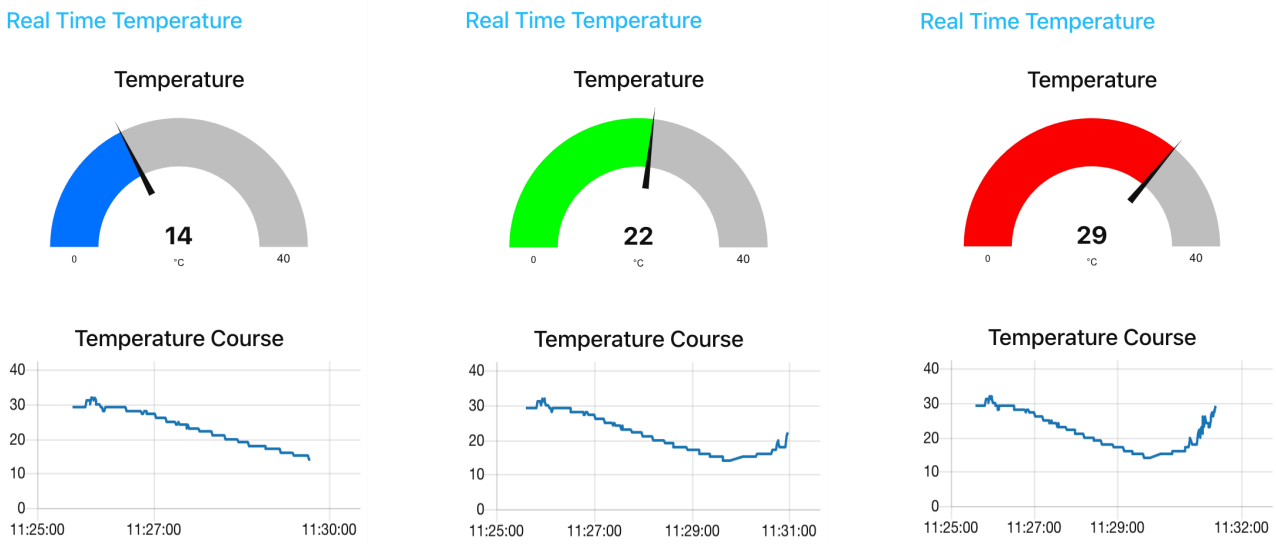
The second task was to integrate a temperature sensor into Arduino and display the collected data in real time on a Node Red dashboard. The dashboard used color coding to clearly show the temperature ranges: red for temperatures above 25 °C, blue for temperatures below 15 °C, and green for values in between.
This task did not present any challenges, since it was similar to previous exercises, which made the implementation of the dashboard and color coding smooth.



Node-Red flow Temperature monitoring Dashboard

```
#include <WiFiNINA.h>
#include <Arduino_LSM6DSOX.h>

void setup() {
  Serial.begin(9600);
  if(!IMU.begin())
    Serial.println("IMU error");
}

void loop() {
  int temp_deg = 0;
  IMU.readTemperature(temp_deg);
  Serial.println(temp_deg);
  delay(180000);
}
```

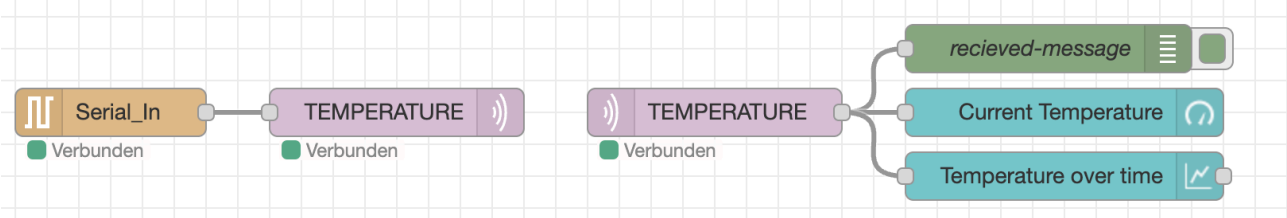
Arduino code to send temperature values over a serial port



Dashboard with different temperature values

Exercise 3: Cloud integration with MQTT

In this task, temperature data was sent from the Arduino via Node-Red to an MQTT broker (HiveMQ) and then visualized in the IoT platform Datacake. The focus was on setting up a secure and reliable data flow using the MQTT protocol.



Node-Red flow to send/receive temperature data over MQTT

Challenges and solutions

The integration of the MQTT protocol required several steps and brought with it some challenges, which were, however, successfully solved:

Use of MQTTS:

When trying to establish the connection between Node-Red and Datacake, an unsecured MQTT connection was initially used, which led to connection errors. After switching to MQTTS (an encrypted version of the protocol), the connection could be established without any problems. This ensured that the transmitted data was sent securely and in accordance with current security standards.

Uplink decoder:

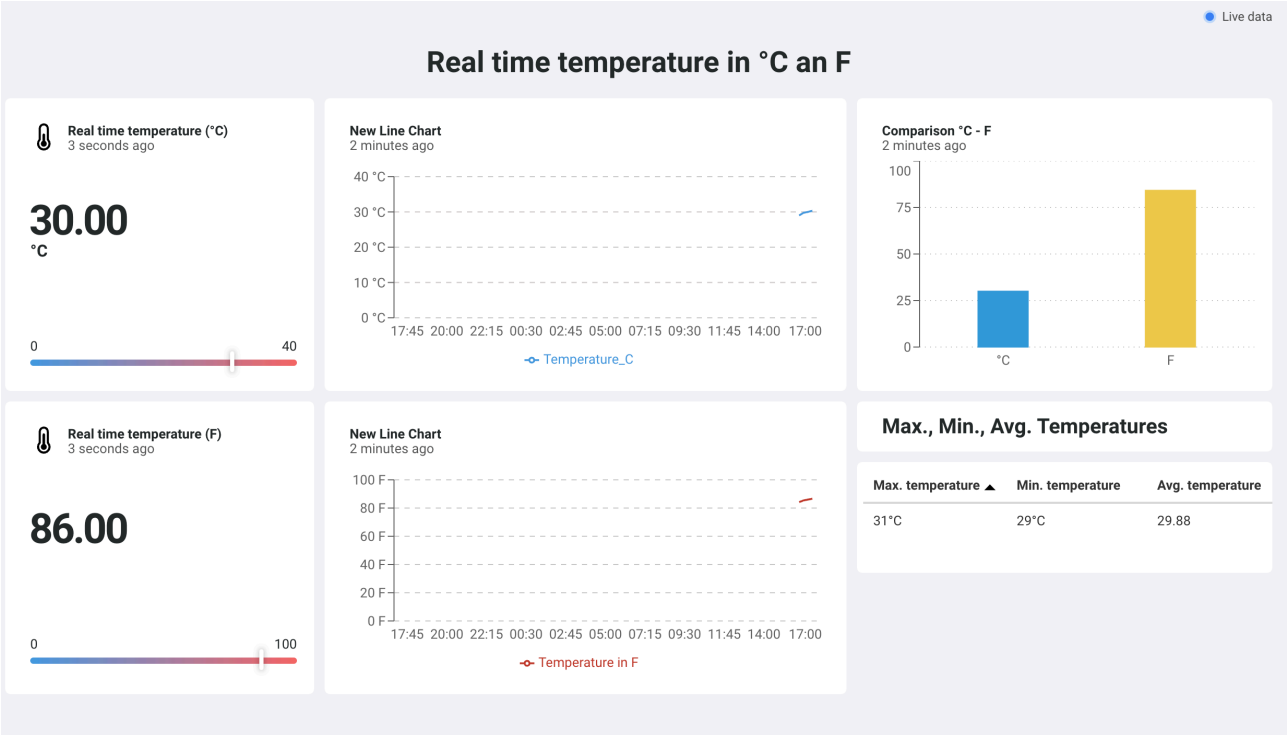
Building the uplink decoder correctly in Datacake was a bit challenging at first, but was successfully implemented after carefully analyzing the documentation. This ensured that the received MQTT messages were correctly interpreted and assigned to the defined data fields.

Datacake data limit:

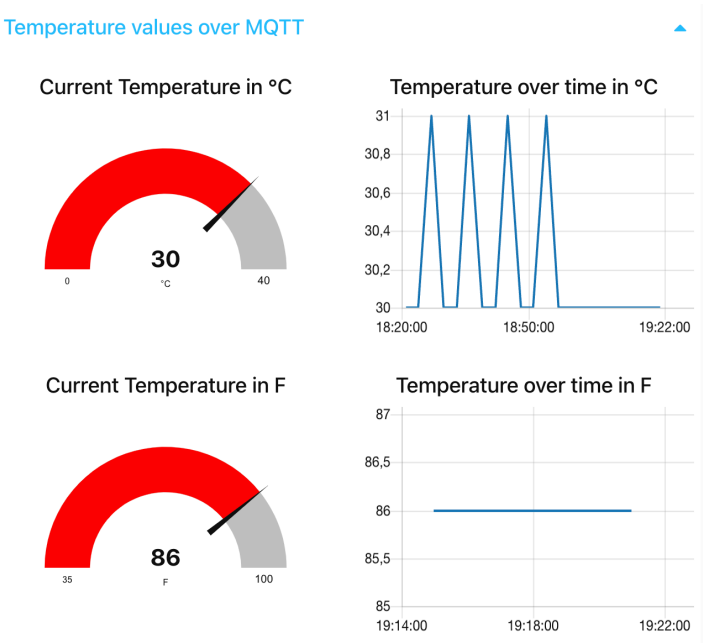
Datacake offers a limit of 500 messages per device per day in the free plan. This limit was quickly reached as data was originally sent every second. To solve this problem, the temperature data sending interval was increased from one second to three minutes. This drastically reduced the total number of messages transmitted daily and ensured that the limit would no longer be exceeded during normal operation.
To immediately work around the problem and continue operations, another device was created within the same Datacake account. Since the data limit is per device and not per account, the same configurations as the original device could be used.

```
function Decoder(topic, payload) {
    var temperature_c = payload
    var temperature_f = (payload * 9/5) + 32
    return [
        {
            device: "2bf0ffa7-2161-485e-92d1-2805c6a679fb",
            field: "TEMPERATURE_C",
            value: temperature_c
        },
        {
            device: "2bf0ffa7-2161-485e-92d1-2805c6a679fb",
            field: "TEMPERATURE_F",
            value: temperature_f
        }
    ];
}
```

Datacake upstream decoder for °C and F



Datacake Dashboard with values received over MQTT



Node-Red Dashboard with values received over MQTT