Hochschule Konstanz
Department of Computer Science

UC

AIN 6

Ubiquitous Computing

Wintersemester 24/25

Name: Tobias Sautter

Matr.-Nr.: 304790

Lab 1: Introduction

The exercises we completed in the first laboratory were generally meant to help us get used to the Arduino environment and some basic coding. The code for every exercise can be found on github.com/Sauttets/Ubiquitous-Computing

Exercise 0: Blink

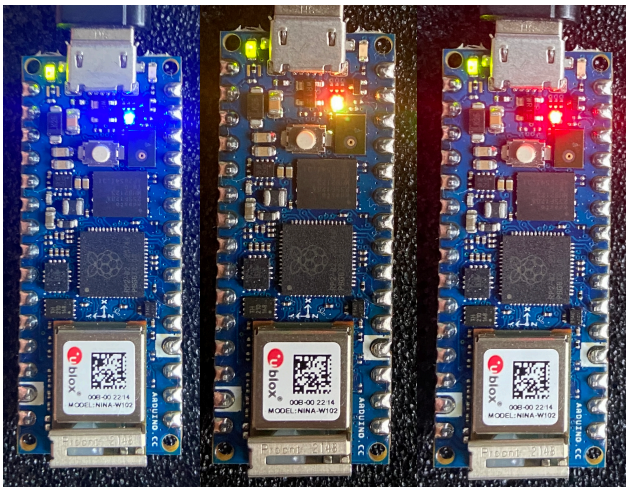
The introductory exercise was straightforward since the task was extremely detailed. After installing the Arduino IDE and setting up the board, it was as easy as opening the example sketch that came with the Arduino-IDE and uploading it to the Arduino.

Exercise 1: RGB

Since the first exercise required only making the RGB LED blink every half second, we decided to use the delay function to ensure the LED color changes every 500ms. If the code had additional features, we would have chosen an interrupt-based timer, as the delay would otherwise block other code from executing.

Exercise 2: Temperature

In the second exercise, we used the RGB LED from before in combination with the temperature sensor, that was already mounted on the controller. The LED was supposed to change color depending on the current temperature. To implement this function, we used the Arduino_LSM6DSOX library, as suggested in the description of the task.



Temp < 25°C *32°C> temp < 36°C* *temp > 32°C*

Whilst testing our code, we encountered some issues with the temperature sensor. Since it is so closely mounted to a heat generating component, the sensor always displayed values well above the actual temperature.
We overcame this issue by using a radiator to heat the Arduino above 36°C and a fridge to cool it below 20°C.

Exercise 3: Microphone

The objective of this task was to get used to and understand the microphone on the controller. Since the code for this exercise was already given, we just uploaded the code to the Arduino and tested it. As described in the instructions, the code would turn the LED on and off if the noise level exceeded a given threshold.

Exercise 4: Sitting posture

In this exercise, we had to modify existing code to use a different IMU library than the one previously used in the code example. This was pretty straightforward since the two libraries are quite similar.
However, we encountered an issue with one of the tasks that we couldn’t resolve. Setting the filter frequency to 104 Hz always resulted in either faulty or extremely delayed values. Thus, we decided to stick with 25 Hz, using the `millis()` function to read 25 times per second instead of 104 times.
Since only the sitting posture was of interest to us, we stripped down the code to use only the pitch value, as this corresponds with leaning over and not sitting straight anymore.
In the end we also implemented a LED, that would light up if the posture exceeded an acceptable tolerance. To ensure we wouldn't loose any functionality we implemented this LED like described in exercise 1.

Task 1: Lifestyle/Health monitor

After completing the initial exercises, I worked on improving the device's functionality by integrating a posture sensor with a noise sensor to enhance monitoring in study environments. The goal was to detect inappropriate noise levels and poor posture, triggering alerts through LEDs based on threshold values. The device uses a red LED to indicate noise levels that exceed an acceptable threshold, a blue LED to indicate poor posture, and a green LED to signal normal conditions after a 10-second delay from the last alert.

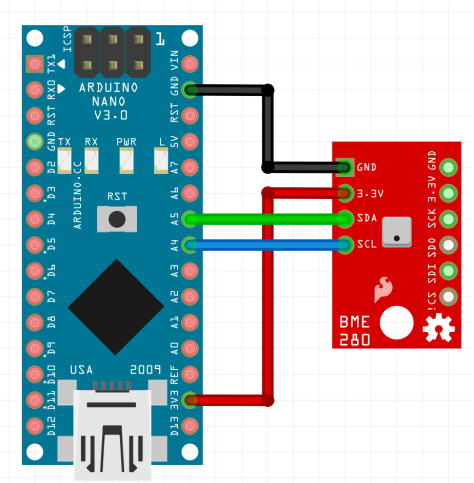
The process of implementing both sensors together required careful adjustments to ensure synchronization, particularly in handling timing issues. Initially, I encountered challenges in making both sensors work simultaneously due to overlapping conditions and timing conflicts. By restructuring the if statements and consolidating the LED control logic, I managed to create smooth, coordinated operation between the two sensors.

After achieving smooth operation with both sensors, I attempted to implement the temperature sensor as well. Ultimately, I decided to remove this implementation tough since I could not get accurate readings from the sensor. This is likely due to its mounting position, which appears to be too close to heat-generating components of the board (probably the CPU).

The Code starts off by defining global variables and initialising the necessary sensors. Within the main loop, the pitch value of the gyroscope is read continuously.if an improper posture is detected, the blink interval of the blue LED is updated and the code is sending notifications over the serial bus. The microphone code behaves similarly, but uses the red LED to notify that the sound threshold has been exceeded. The final segment contains the blinking logic, that reads the current time and compares it with the blink time, to determine If the LED should blink or not.

Further Improvements

To achieve more accurate readings, the temperature sensor could be replaced with an external BME280 sensor, which additional to the temperature sensor also provides humidity, and barometric pressure readings. Using the BME280 would allow for accurate measurements thus enabling us to use even more Sensors to ensure a good working environment.



Example wiring of a BME280 via I2C

Another potential improvement for the future could involve a monitoring system that checks whether the user is moving enough. Since sitting in the same posture for extended periods is unhealthy, the tilt sensors could monitor if the user is moving sufficiently. Additionally, since the BME280 I proposed earlier has quite a high resolution, this sensor could even detect if the user is standing up, making it easy to track activity as well.

One last improvement that could be pretty useful, would be the power efficiency. Depending on the area of application of the controller, having to be plugged in to a power supply, is rather inconvenient. To improve the power consumption, the Arduino should check the value of the different sensors periodically rather than constantly. To implement this change though, the code needs a complete overhaul, since the gyroscope works of relative values instead of absolute values. This means, that we need to find a way to get the correct posture with just one call to the sensor.

Task 2: Arduino WiFi

There are two main approaches to using the WiFi module on the RP2040. For the first approach, the controller can connect to an existing network and set up a web server that other devices on the network can access. This is especially useful in smart home use cases where the user can access the controls of their house via a simple website within their home network.
The second approach involves running the controller in AP mode. This makes the controller itself an access point, allowing other devices to connect directly to it. Within this network, the Arduino can then set up a simple web server accessible to all devices connected to the Arduino. This approach is often used in rural areas that don't already have an existing WiFi network.
To get started, we only need the Arduino board itself, the Arduino IDE, and the WiFiNINA.h library. This library is responsible for essential WiFi functions, such as handling network protocols and managing data transfer over WiFi.

Setting up the Arduino in Access Point (AP) Mode:

After connecting the Arduino to the PC and including the necessary libraries, we define two global variables for the SSID (the name of the WiFi) and the password. Within the setup function, we initialize the Serial Monitor with the `Serial.begin()` function to allow for debugging on the Serial Monitor. We then use the `WiFi.status()` function to check if the WiFi module is ready before calling `WiFi.beginAP(ssid, pwd)` to start the access point. The Arduino gets the default IP address 192.168.4.1, which can be changed with the WiFi.config function if desired. A new WiFi network should now be visible in the WiFi settings of your device.

Setting up the Arduino in WiFi (Client) Mode

Similar to the setup in AP mode, we need to import the WiFiNINA library and define global variables for the SSID and password. However, this time we set the variables to the values of an existing network we want to connect to. In the setup function, we again check if the WiFi module is ready with the `WiFi.status()` function, but this time use `WiFi.begin(ssid, pwd)` to connect to the network rather than creating one ourselves. The `WiFi.localIP()` function provides the IP of the Arduino within our network. It should be noted that the IP address can

change after the Arduino reconnects to the WiFi. This can be fixed by either using the `WiFi.config()` function or by assigning a static IP in your DHCP server.

Setting up a Web Server on the Arduino:

After setting up the Arduino in Access Point mode or Client mode, we can implement a simple HTTP server on the Arduino. We start by creating a global WiFiServer object using the `server(80)` function to specify the port. Within the setup function, we call `server.begin()` to start the HTTP server. Within the main loop, we use `server.available()`, which will return a WiFiClient object representing the connection if anything connects to the web server. Using this client object allows us to read messages from the client and send messages to it.

Order of Events:

When a user accesses the web server, the client will send an HTTP request to the Arduino. The Arduino will then respond to the request with its HTTP version, a status code such as 200, which means OK, and a content type like text/html to signal to the client how to interpret the data it is receiving. After this, the server can send the body content, for example, consisting of standard HTML statements. Once this is done, the connection is closed, and the server waits for another request.

```
client.println("HTTP/1.1 200 OK");
client.println("Content-type:text/html");
```

HTTP Server response

Controlling the Arduino with the HTTP Server:

To control the Arduino with the HTTP server, we can create simple HTML buttons that link to specific URLs. Depending on the URL called, the Arduino can handle the requests accordingly. In the example provided, we control the RGB LED on the Arduino with six endpoints. Each LED is assigned two endpoints to turn it on and off. By adding listeners to each endpoint, we can execute the code to enable or disable an LED.

```
if (currentLine.endsWith("GET /RH")) {
  digitalWrite(LEDR, HIGH);
}
```

HTTP endpoint listener example