IoT WiFi Cars Speedometer

Microprocessor and Embedded Systems

Szymon Pluta

Informatics

Instructor: Bartłomiej Zieliński, PhD, DSc

Submit date: 06.02.2021

Faculty of Automatic Control, Electronics and Computer Science
Silesian University of Technology

Contents

1 Introduction

The goal of my project was to design and implement a prototype of the cars speedometer device that will save the registered cars speeds into the remote database using WiFi and HTTP.

Such a device can be deployed on-site in a real environment. The device then collects the data into the database.

1.1 Data analysis possibilities

Countless data analysis possibilities arise due to having collected the data in the form of a tuple: `cars speeds values` paired with `registered timestamp`.

Actual applications of the collected data are up to the data consumers (e.g., a team of data scientists working for the government), however I leave some exemplary usages below.

1.1.1 Safety analysis

Consider a scenario where this device is deployed on-site in a fixed place for a period of 2 years.

Having logged that much data, the data scientists could draw the conclusion **whether the car drivers *tend to* break the speed limit**, or not, relatively to other places.

Chances are that the speed limit is indeed being broken too often in that place of interest. In such a case, it would have been a strategic decision to **mount a real speed camera** or other **safety measures** such as road signs, road railings, etc.

1.1.2 Traffic density analysis

It is *easily* possible to compute the **road bandwidth**, i.e., the number of travelled cars within some time period t. Such road bandwidth parameter finds usages in environmental engineering area.

Peek hours could be derived and further used in other analysis (e.g. as a theoretical background for increasing public transport resources during peek hours or peek weeks).

When building a new **road intersection** or modernizing one, the computed bandwidth could help make a strategic decision to the following design problem:

Which intersection would meet the requirement of the road's bandwidth?

- a). intersection costing \$50 mln with a bandwidth of 300 $cars/hour$
- b). intersection costing \$65 mln with a bandwidth of 370 $cars/hour$

1.1.3 Multivariate analysis

The device also collects measured temperatures into the database, using the two distinct temperature sensors.

Therefore, it is entirely possible to perform a multivariate analysis, i.e., analyze how does the temperature (weather component) affect the traffic or cars speed.

Furthermore, coupling the data with some external data, such as the car accidents data, leads to a yet another analysis scenario.

2 Hardware

The device is built out of the following components:

- ESP8266 NodeMCU v3 (ESP12E)
- OLED 0.96" SSD1306 (128x64, I2C)
- LCD 2x16 + I2C LCM1602 converter
- 2x: Ultrasonic sensor HY-SRF05
- 2x: Temperature sensor DS18B20
- Bi-directional 8-channel logic level shifter
- Buzzer with generator
- Blue LED
- R10k, R330

2.1 ESP8266

Figure 1: ESP8266 NodeMCU v3

NodeMCU V3 is an open-source firmware and development kit with 10 GPIO pins, and is capable of generating PWM, I2C, SPI and UART serial communications.

The firmware is based on Lua programming language.

There is ESP12E network module, which is a WiFi module that operates using `802.11 b/g/n` standard using 2.4 GHz.

`USB-UART CH340G` converter is embedded within, which is used for both programming and powering the microcontroller, however, microUSB is used instead of a typical USB port.

The GPIO pinouts operate in 3.3 V.

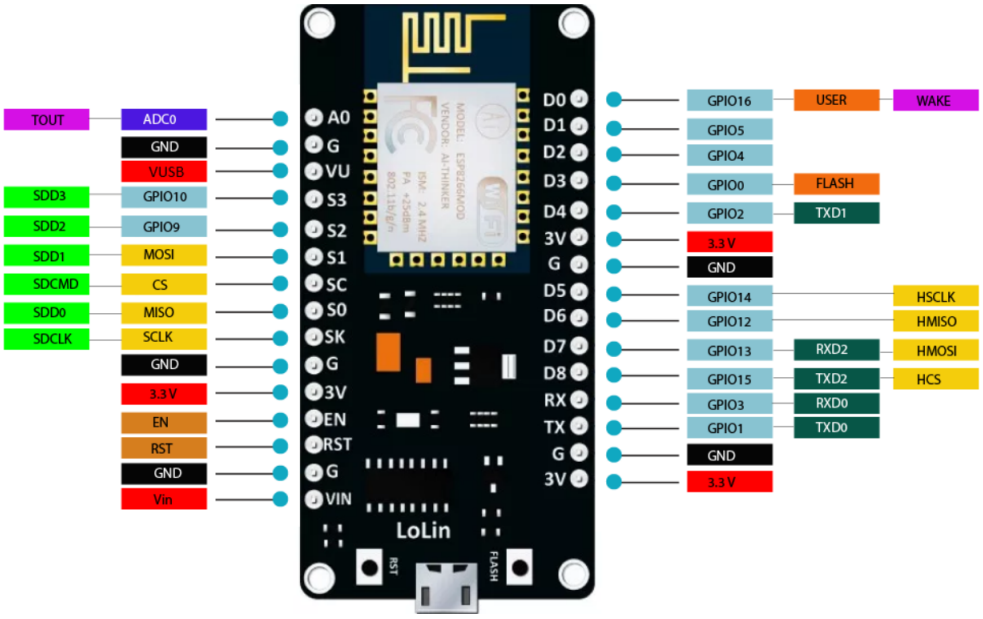

Figure 2: ESP8266 NodeMCU v3 pinout

The microcontroller can be either powered by USB (5V) or `VIN` pin.

If and only if the device is powered with USB, then the pin `VU` can serve as a 5 V voltage source for powering other components.

There are three 3.3 V sources, and 5 `GND` pins.

2.2 OLED 0.96”

This OLED display works as a **main display** for the car drivers. Information relevant to the drivers is shown on the screen.

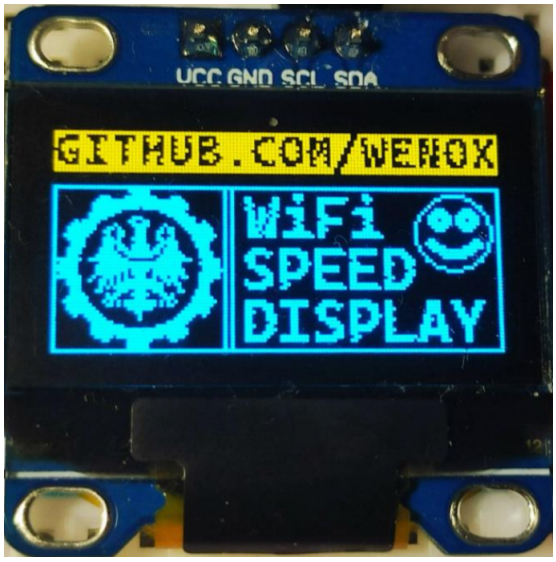

Figure 3: OLED 0.96” SSD1306

This OLED can be supplied with $3 \div 5\ V$ and uses I2C for communcation with the ESP8266. I used $3.3\ V$ for V_{cc}.

The height of this device is $64\ px$: first 16 rows of pixels are yellow, and the remaining are blue.

2.3 LCD 16x2 + I2C Converter

This LCD display works as an **admin display**, that is: it must not be visible for the drivers. Instead, the display is used for showing information relevant to the on-site engineers, such as error messages.

The LCD needs $5\ V$ supply, and also uses I2C for communication with the ESP8266.

Typically, this LCD is troublesome to connect due to large number of pins it requires, however, I used the version with `I2C LCM 1602 converter` soldered-in. This decreased the required number of pins to only 4.

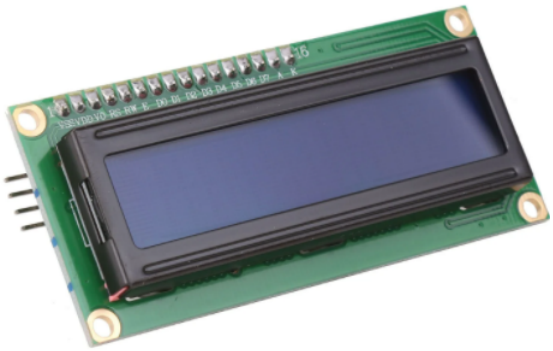

Figure 4: LCD 16x2 + I2C LCM1602 converter

2.4 Ultrasonic sensors

Essentially, this device repeatedly sends chunks of the ultrasonic waves, and awaits for the reflection and return of thereof. Echolocation works similarly.

The declared range of detection is up to 400 cm.

I have used two of those sensors, and they operate in 5 V.

`TRIG` is the input pin, whereas `ECHO` is the output pin. The output signal is PWM proportional in length to the detected range.

The pin `OUT` is not used.

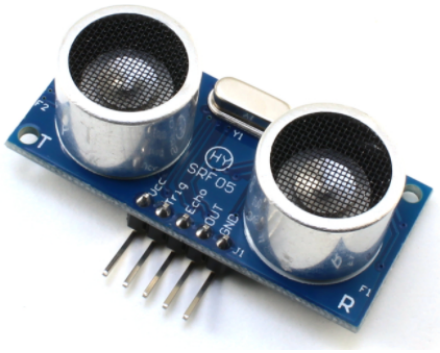

Figure 5: Ultrasonic sensor HY-SRF05

2.5 Temperature sensors

I use two of those termometers. The idea is that the first measures the temperature of the air, and the second measures the temperature of the ground.

Both termometers are easily connected together, in a serial fashion.

Parasitic connection could also be used, however I decided not to, as it tends to be somewhat unstable.

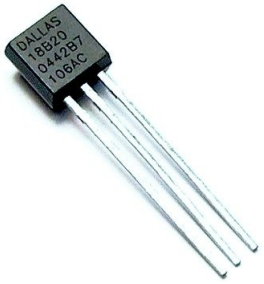

Figure 6: Temperature sensor DS18B20

This digital sensor detects the temperatures from $-55\ ^{\circ}C$ to $125\ ^{\circ}C$ and uses **1-Wire** interface.

The measurement uncertainty is $\pm 0.5\ ^{\circ}C$ in the range from $-10\ ^{\circ}C$ to $85\ ^{\circ}C$, and the resolution is from 9 up to 12 bits.

The V_{cc} is compatible with $3 \div 5.5\ V$, and I decided to power it with $3.3\ V$.

2.6 Blue LED

The diode works as an indicator that is lit everytime HTTP request is sent. I paired the diode with $330\ \Omega$ resistor.

Figure 7: Blue LED and $330\ \Omega$ resistor

2.7 Logic level shifter

This converter can handle conversions from the 1.8 V up to 6 V.

Given the fact that ESP8266 GPIO operate in 3.3 V, and some of the components work in 5 V, and some in 3.3 V, I needed to use the logic level shifter to make it compatible.

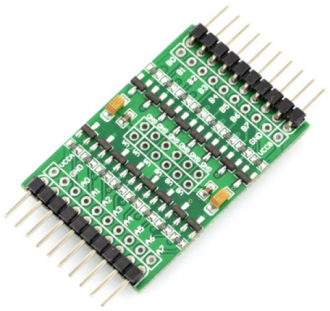

Figure 8: Logic level shifter with 8 channels

In fact, most of my components operate in 5 V, making this logic level shifter obligatory component.

`VU` pin which is 5 V source is connected to the V_{CC_A} of this logic level shifter, whereas `3V3` pin is connected to the V_{CC_B}.

2.8 Buzzer

This component is responsible for producing short sound signals that inform the user about boot-up success, or some failure, according to the predefined sound error codes.

Figure 9: Buzzer with generator

It can work with any voltage in the range $[3, 18\ V]$. I have used 5 V to power it and pin $D0$ to steer it.

3 Software

The software consists of ESP8266 module and HTTP server.

Proper HTTP client could also be implemented as a simple extension to this project.

3.1 Device

The device was programmed using Arduino IDE and USB port.

3.1.1 Overview

Here is the high-level overview of the embedded device software:

```
void setup() {
        delay(INITIAL_STARTUP_DELAY);

        init_ESP8266();
        init_WiFi();
        init_LCD();
        init_OLED();
        init_sensors();

        buzzer_system_success();
        draw_splash_screen(DISPLAY_SPLASH_SCREEN_DURATION);
}

void loop() {
        if (!is_WiFi_connected()) {
                handle_WiFi_not_connected();
        } else {
                process_ultrasonic_sensors();
                process_temperature_sensors();
                process_LED();
                update_OLED_mode();
                draw_OLED();
        }
}
```

3.1.2 Libraries

The following libraries were used:

```
#include <SPI.h>
#include <Wire.h>
#include <OneWire.h>
#include <DallasTemperature.h>
#include <Adafruit_GFX.h>
#include <Adafruit_SSD1306.h>
#include <Arduino.h>
#include <ESP8266WiFi.h>
#include <ESP8266WiFiMulti.h>
#include <ESP8266HTTPClient.h>
#include <WiFiClient.h>
#include <LiquidCrystal_I2C.h>
```

3.1.3 Configuration

The device is configurable.

Server configuration:

```
/** Server configuration. */
const String SERVER_ADDRESS   = "http://192.168.43.252:8080";


/** REST endpoints. */
const String API_SPEED        = "/api/speed";
const String API_TEMPERATURES = "/api/temperatures";
const String API_INFO         = "/api/info/next";
const String API_COVID        = "/api/covid-cases";
```

Customizable configuration:

```
/** Device configuration. */
#define HOW_OFTEN_TEMPERATURES_REQUEST 15000 // Temperature measurements are
                                     // sent to the database every 15 seconds.

#define DISPLAY_SPEED_DURATION         4500  // Show speed for 4.5 seconds
#define DISPLAY_MESSAGE_DURATION       3000
#define DISPLAY_COVID_CASES_DURATION   4000
```

```
#define DISPLAY_TEMPERATURE_DURATION   4000
#define DISPLAY_SPLASH_SCREEN_DURATION 6000
#define SENSOR_1_INVALIDATE_AFTER      12000
#define LED_DURATION                   150

#define SENSORS_GAP_DISTANCE 1000 // Two cases to distinguish here:
// 1. For an nearly-instantaneous speed, the gap between the sensors
//    should be as small as possible in a real environment (e.g. 100 cm).
// 2. For an average speed on some distance,
//    the gap between the sensors could be of an arbitrary length (e.g. 5 km).

#define SENSOR_DETECT_RANGE    20   // The detection range should be as high
                // as possible in a real environment (up to 400 centimeters).
#define SENSOR_MAX_RANGE       400
#define SERIAL_PORT            115200
#define INITIAL_STARTUP_DELAY  5000   // Initial 5 seconds start-up delay.
```

Pinout configuration:

```
// GPIO pins values
#define D0  16
#define D1  5
#define D2  4
#define D3  0
#define D4  2
#define D5  14
#define D6  12
#define D7  13
#define D8  15
#define D9  3

// Buzzer
#define PIN_BUZZER D0
#define BUZZER_SHORT_BEEP_DELAY 250
#define BUZZER_LONG_BEEP_DELAY  750

// Ultrasonic sensors
#define PIN_TRIG1 D6
#define PIN_TRIG2 D4
#define PIN_ECHO1 D5
#define PIN_ECHO2 D3

// Temperature sensors
#define PIN_TEMPERATURE D7

#define PIN_LED D8
```

3.2 Server

The HTTP server was implemented using Spring Boot (Java 11) and the implementation follows a simple layered architecture.

3.2.1 REST

The server exposes the following REST endpoints:

- **POST:** `/api/speed`
- **POST:** `/api/temperatures`
- **GET:** `/api/info/next`
- **GET:** `/api/covid-cases`

The `POST` endpoints persist the registered speeds and temperatures into the database:

- Everytime the car is detected, a request with registered speed payload is sent to `POST: /api/speed`.
- Request with registered temperatures is periodically sent to `POST: /api/temperatures`, e.g., every 15 seconds, as configured in:

```
1 #define HOW_OFTEN_TEMPERATURES_REQUEST 15000
```

The endpoint `GET: /api/info/next` fetches the next message that is to be displayed for the drivers on the OLED display, e.g., *"Drive safe"* or *"Remont drogi za 3 km"*, whereas the endpoint `GET: /api/covid-cases` fetches the daily coronavirus cases from the `https://gov.pl/` website.

3.2.2 Database

Exemplary records saved by the device into the **PostgreSQL** database:

id	registered_time	speed_value
8,608	2021-01-20 09:05:48	6.37
8,611	2021-01-20 09:06:24	21.42
8,612	2021-01-20 09:06:26	58.63
8,613	2021-01-20 09:06:27	58.73
8,614	2021-01-20 09:06:29	96.26
8,615	2021-01-20 09:06:30	145.75

Figure 10: Registered cars speeds

id	registered_time	temperature_sensor_1	temperature_sensor_2
49	2021-01-20 00:58:40	23	23
50	2021-01-20 00:59:04	23.06	23.13
51	2021-01-20 00:59:28	21.5	21.75
52	2021-01-20 00:59:52	20	20.31
53	2021-01-20 01:00:16	19.13	19.56
54	2021-01-20 01:00:40	18.69	19.19

Figure 11: Temperatures registered right after opening the window in the room

id	registered_time	message
1	2021-01-19 22:12:48	ZAPNIJ PASY
2	2021-01-19 22:12:48	DRIVE SAFE
3	2021-01-19 22:12:48	TRZYMAJ SIE PRAWEGO PASA
4	2021-01-19 22:12:48	PAMIETAJ O KORYTARZU ZYCIA
5	2021-01-19 22:12:48	UWAGA! WYPADKI
6	2021-01-19 22:12:48	KONTROLA PREDKOSCI
7	2021-01-19 22:12:48	REMONT 3 KM
8	2021-01-19 22:12:48	OBJAZD AUTOSTRADA A4
9	2021-01-19 22:12:48	INFORMACJA DROGOWA 19-111
10	2021-01-19 22:12:48	KONTROLA GRANICZNA
11	2021-01-19 22:12:48	STOSUJ JAZDE NA SUWAK

Figure 12: The messages to be displayed for the drivers

3.3 Client

The device works as a HTTP client.

However, *another* HTTP client could be implemented (working as a remote client), whose responsibility would be the maintenance of the `Figure 12` database table (effectively performing CRUD operations).

Such a client could have proper user-friendly user interface.

Alternatively, `curl` or similar software could be used:

```
curl --header "Content-Type: application/json" \
     --request POST \
     --data   '{"message":"WARNING! Wet road!"}' \
              http://localhost:8080/api/info
```

4 Circuit design

4.1 Logical schematic

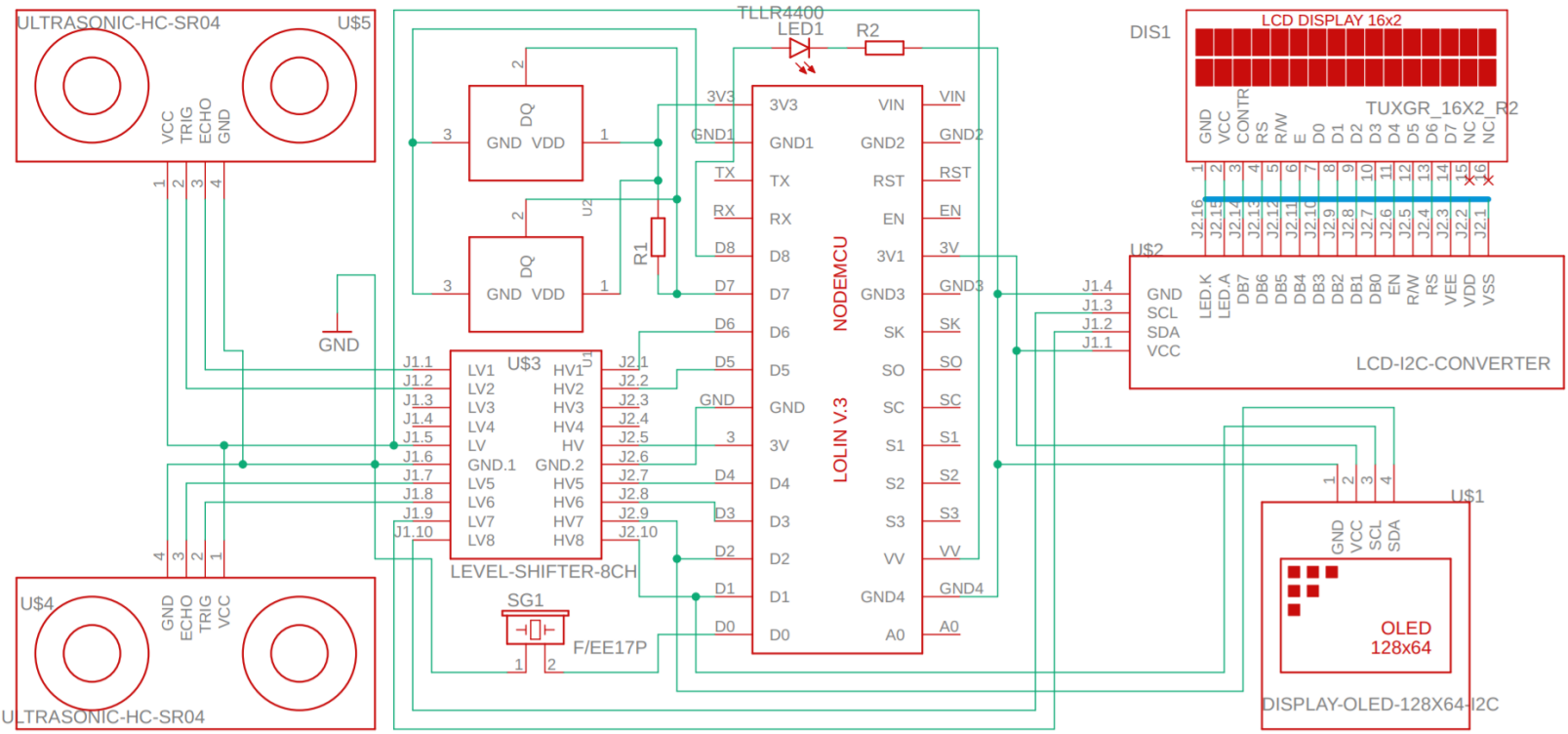

Figure 13: Logical schematic

4.2 Printed circuit board

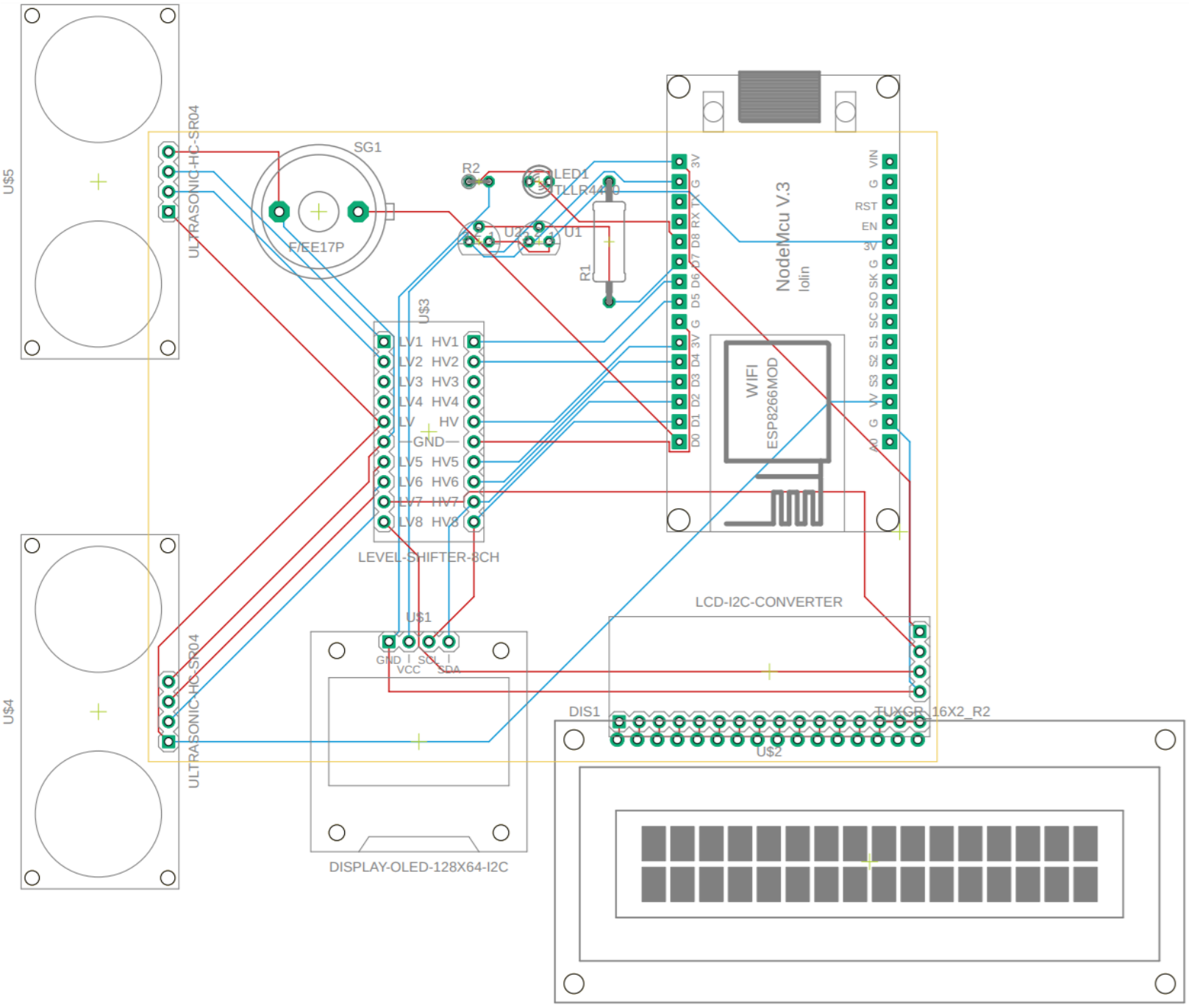

Figure 14: Printed circuit board

5 Results

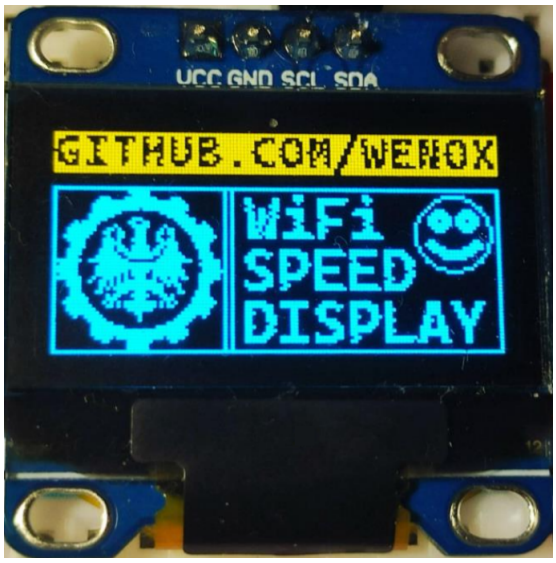

Figure 15: Startup splash screen

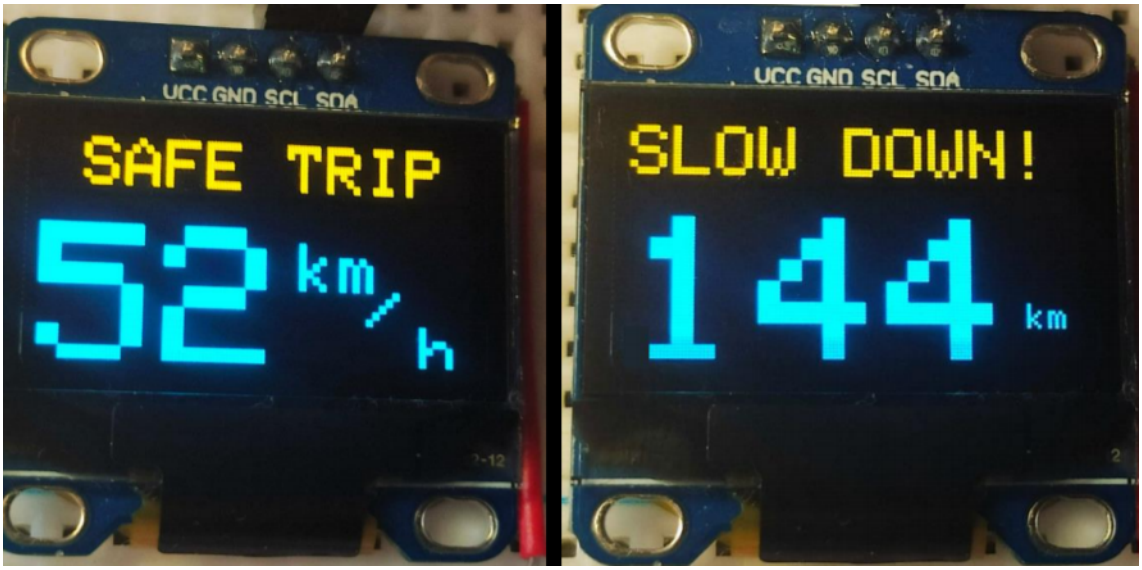

Figure 16: Displaying the speed

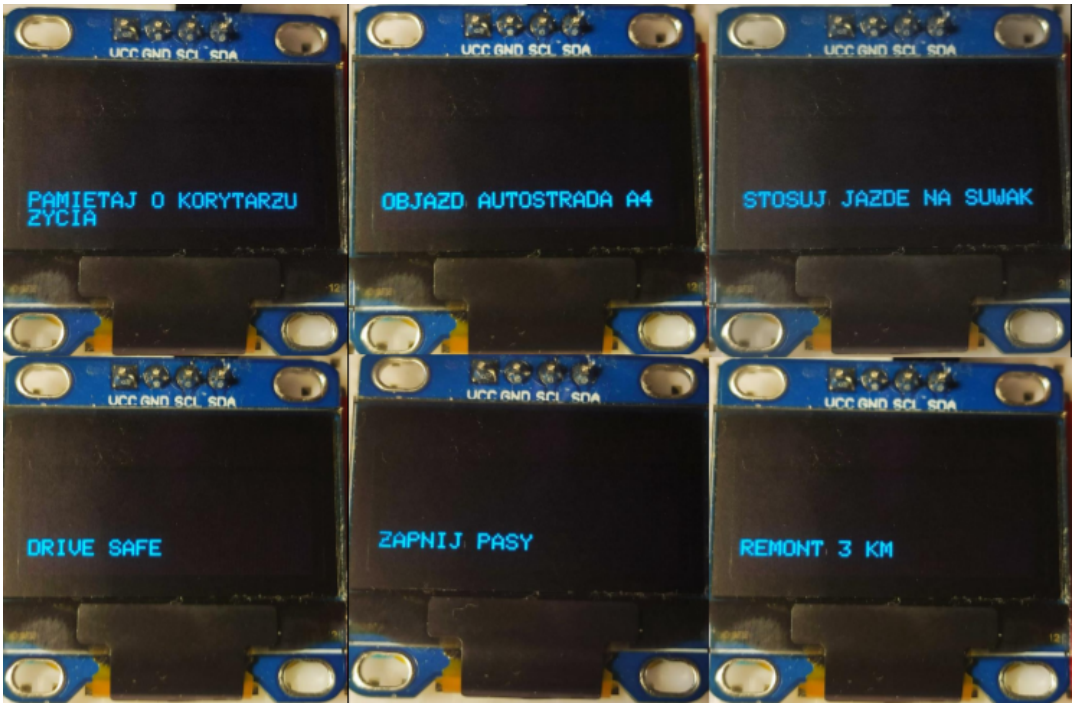

Figure 17: Messages for the drivers

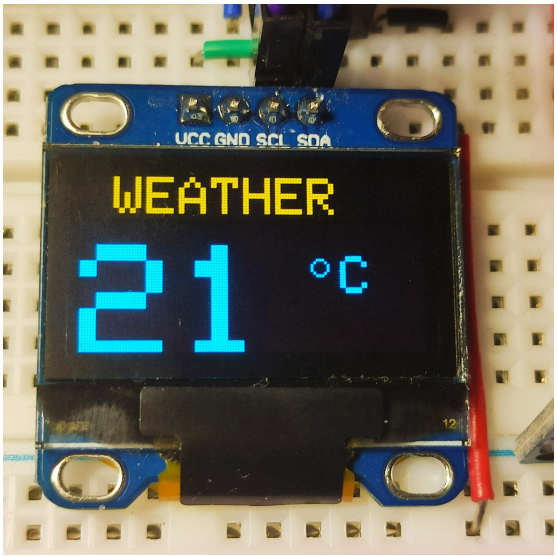

Figure 18: Displaying the temperature

Daily new cases of the coronavirus:

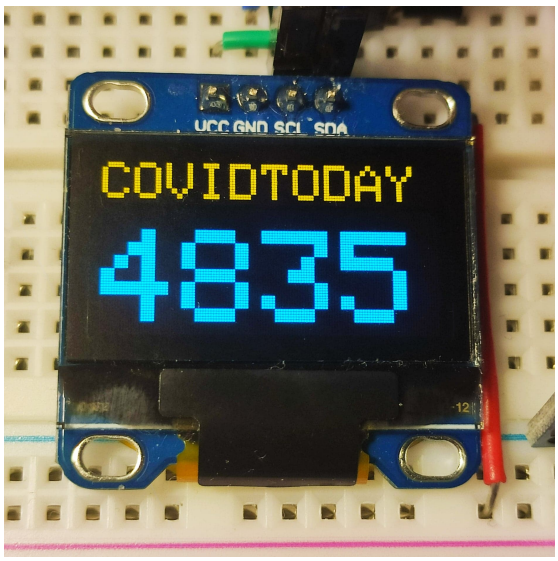

Figure 19: Any information could be displayed

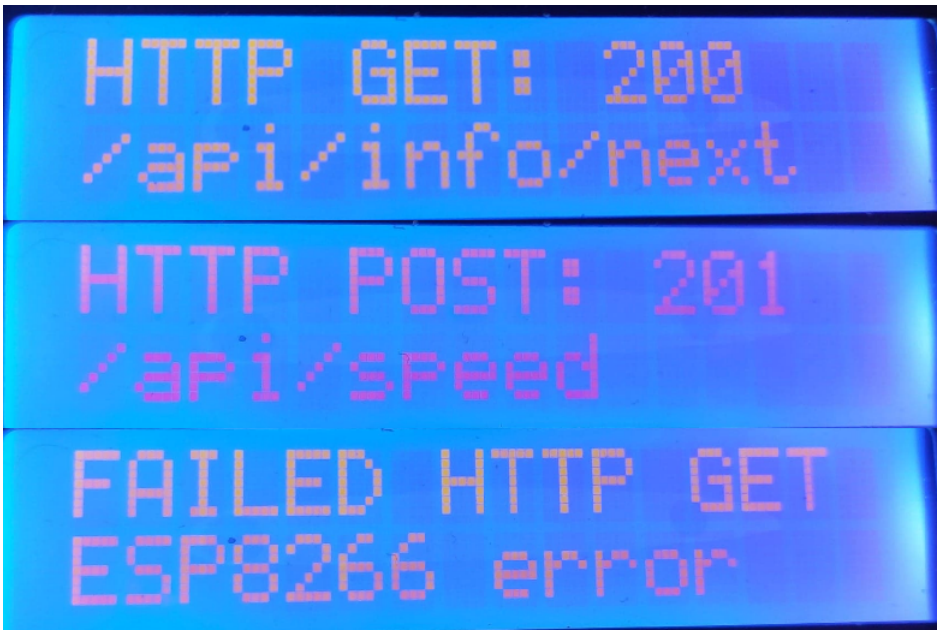

Figure 20: Side display for the on-site engineer

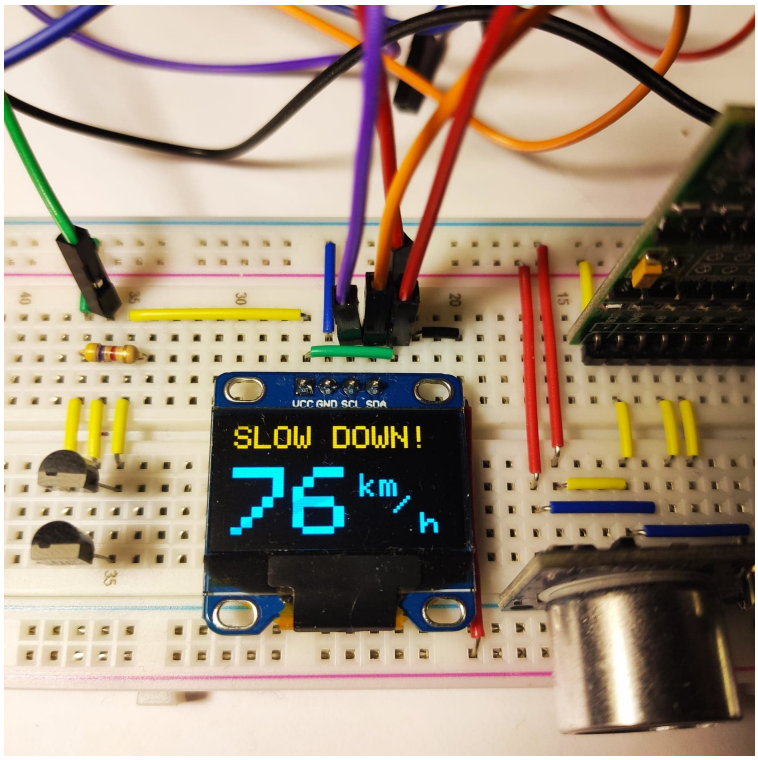

Figure 21: Breadboard core

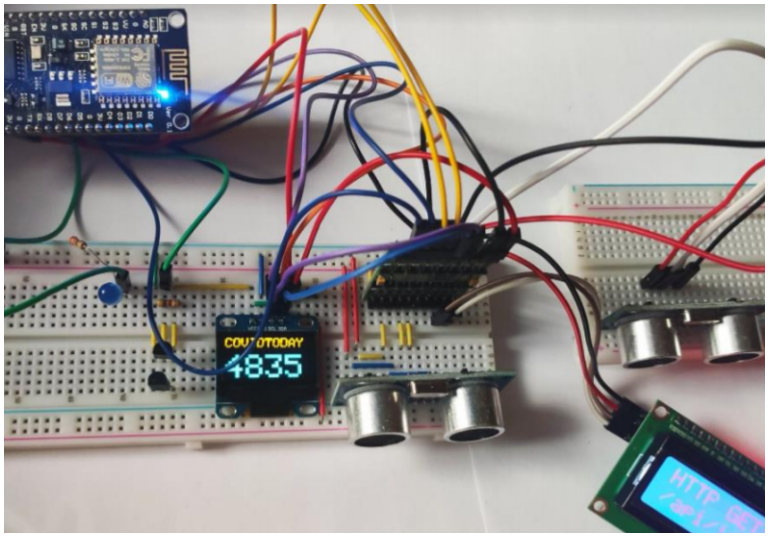

Figure 22: Components mounted on the breadboard

6 Conclusion

The assembled circuit is working properlyp.