Table of Contents

1. Laboratory on the Internet of Things 2
1.1. Hardware 3
1.2. VSCode, PlatformIO, and project configuration 4
1.2.1. Running your code 5
1.3. Software 5
1.4. Networking 5
1.5. Network services 5
1.5.1. MQTT 6
1.6. Tracing 6
1.7. Laboratory scenarios 7
1.7.1. Scenario Lab1 8
1.7.2. Scenario Lab2 9
1.7.3. Scenario Lab3 12
1.7.4. Scenario Lab4 14
1.7.5. Scenario Lab5 15

1. Laboratory on the Internet of Things

Welcome to the laboratory of the Internet of Things. Students are about to perform a number of laboratory tasks. Each task is to be finished with a report summarizing on what and how it was done. The report should be prepared in PDF form and eventually ZIPped with extra files like source code, resources, etc.

1.1. Hardware

Each team is given the following IoT microcontroller:

- Wemos D1 R2 development board:

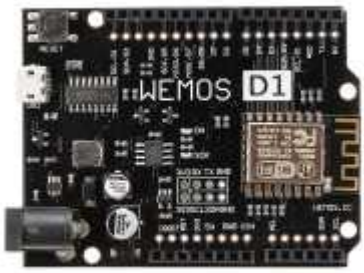


Fig. 1: WeMos D1 R2 development board

Note, this microcontroller, even if its board reminds Arduino Uno has a different logic voltage (here 0..3.3V while Arduino Uno uses classical 0..5V logic) and has different pinout numbering. Detail on the following figure:



Fig. 2: WeMos D1 R2 pinout

- DHT 11 or DHT 22 temperature and humidity sensor



Fig. 3: DHT 11 (left) and DHT 22 (right) temperature and humidity sensors

- 2x16 characters LCD shield with keypad



Fig. 4: LCD shield 2×16 characters

1.2. VSCode, PlatformIO, and project configuration

In any case, creating a new project under PlatformIO should follow this general setting, regarding Board and Framework (see figure 5):

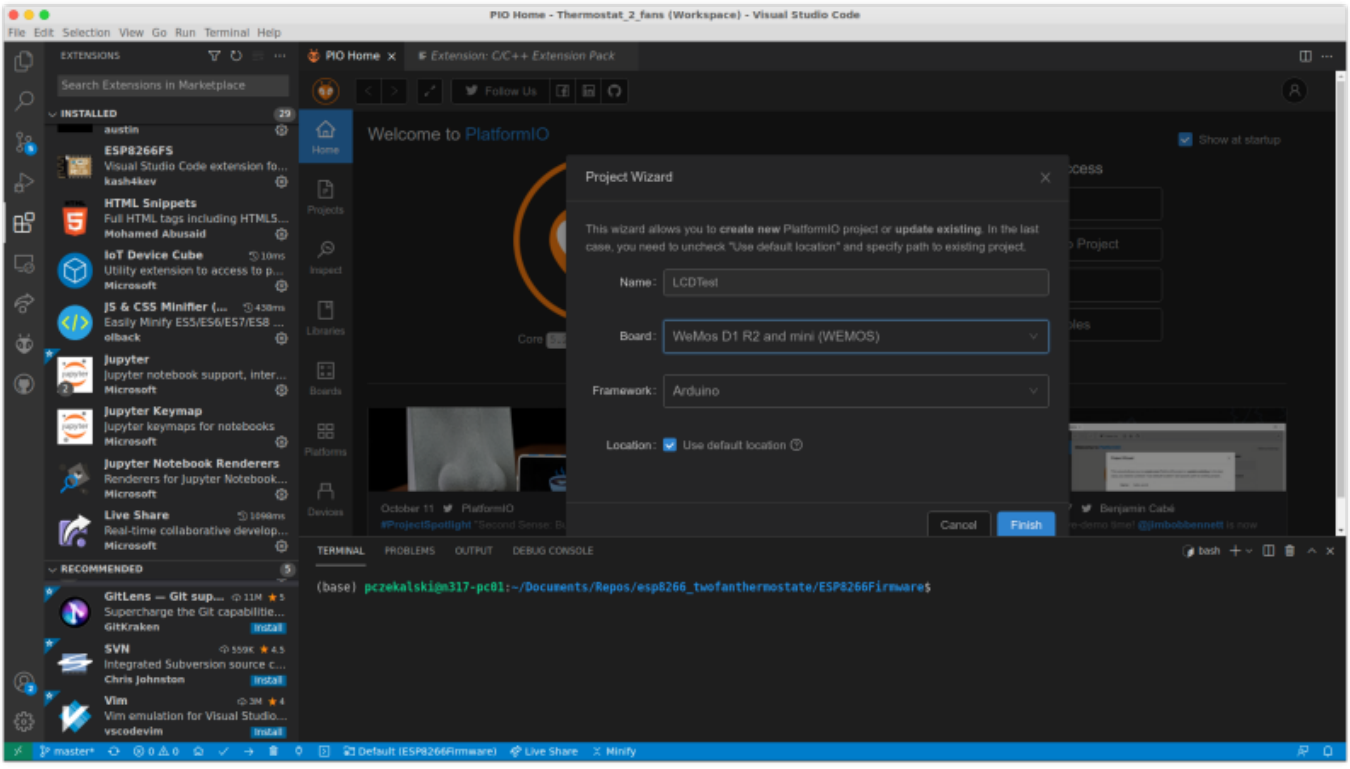


Fig. 5: Project configuration settings in VSCode/PlatformIO

This configuration brings the following project configuration file `platformio.ini`:

```
[env:d1_mini]
platform = espressif8266
board = d1_mini
framework = arduino
```

It is also necessary to configure the serial port name used for uploading your firmware and monitoring (if you use it). In the case of this microcontroller, it is the same port, so the `platformio.ini` file should be modified with the following (assuming the serial port is, i.e., `COM3`):

```
...
upload_port = COM3
monitor_port = COM3
...
```

1.2.1. Running your code

To execute your code, you need to compile it and upload it to the microcontroller.
The easiest way to interact with PlatformIO is to use its blue ribbon bar, located at the bottom of the VSCode (image 6:

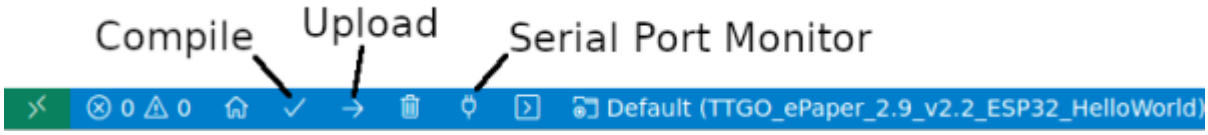


Fig. 6: PlatformIO ribbon to execute most common actions

1.3. Software

You will be writing an application for end nodes (ESP8266) in C++ using VS Code.
Additionally, you may need to use an MQTT client, i.e., MQTT Explorer: MQTT Explorer.


Computers in laboratory rooms 320, 321, and 336 use “freezing” software, which means that for every startup, computers are restored to their initial state.

1.4. Networking

There is a WiFi 2.4GHz, IPv4 network available to the students. This is a PRIVATE network. This network is separated from the Internet, but all connected devices can talk to one another. There is a DHCP server with networking addressing scope 192.168.90.10-192.168.90.254.


Do not use static IP! Some services in the network have fixed IP addresses; overlapping causes the unavailability of your solution and this service to other teams! Use only DHCP for obtaining a network address.

To connect to the wireless network, use the following credentials:

- SSID: internal.IOT
- Passkey: IoTlab32768
- IP address: DHCP, 192.168.90.10-192.168.90.254

1.5. Network services

1.5.1. MQTT

There is an MQTT Broker, a binding WiFi PRIVATE network, and the PUBLIC Campus network. This way, you can route MQTT messages from your end node devices connected to the `internal.IOT` network to the external tools, i.e., cloud-based dashboard or storage via, i.e., publishing from your end node IoT device through the PRIVATE network to MQTT Broker and subscribing to those messages from the cloud using PUBLIC Campus network. Please note the context (thus, the IP address of the MQTT broker) changes depending on which device and network you're using. The following network schema should be of help.

You can access the MQTT broker from the Campus PUBLIC network using the following configuration and credentials:

- Address: 157.158.56.54
- Port: 1883 for plain text, 8883 for secure communication
- User: vrel
- Password: vrel2018

You can also access the MQTT broker from the `internal.IOT` PRIVATE network using the following configuration and credentials:

- Address: 192.168.90.5
- Port: 1883 for plain text, 8883 for secure communication
- User: vrel
- Password: vrel2018

1.6. Tracing

There is no direct debugging capability in this environment. Yet, it is possible to send log/trace messages to the PC console over a serial (serial over USB) port, which is used for flashing. To open serial communication, one must invoke begin with the appropriate bps setting on the `Serial` singleton object (usually in `Setup()` function:

```
Serial.begin(9600);
```

9600 is the default speed, but you can change it in the `platformio.ini` file. It is possible to send virtually any message to the console (keep in mind, you need explicitly open Serial Monitor in the VS code) using:

```
...
Serial.print(...);
//or
Serial.println(...);
...
```

Both functions accept a wide range of argument types without needing to convert to string explicitly.



Note, there is a limit on the character set that is ASCII only. We suggest avoiding



“special” characters, language-specific.

1.7. Laboratory scenarios

All teams share the same scenario yet; please note that MQTT messages exchanged between devices shall be coordinated among team members not to overlay one another.



Mind keeping your code between meetings. It is regular that at every next laboratory meeting, you will extend your code by something extra.

1.7.1. Scenario Lab1

Getting ready with your development environment
In this scenario, you will write your first “Hello World” application for ESP8266. Using the selected IDE, write code that:

1. Flashes built-in LED every 1 second (1s on; 1s off);
2. Sends a message via serial port to the serial port monitor: `Hello World`



The GPIO the built-in LED is connected to is number 14. Physically it is located almost in the middle of the mainboard, under the display, and its color is blue.

```
#define LED_BUILTIN 14
```

Note, to use tracing with the `Serial.print(…)` singleton object, you need to tell VS Code which port to use to receive tracing messages you send from your code. Mind including in the `platformio.ini` file the appropriate configuration (here COM3 is a sample, check yours in PC configuration):

```
...
monitor_port = COM3
...
```

Advanced:

- Use a timer/interrupt to flash the LED instead of an infinite loop.

Report:

- Describe your approach and include source code in the report

1.7.2. Scenario Lab2

** Using LCD **

In this scenario, you will write an application that reports on the LCD Display.
Your code shall implement a message `Hello IoT` in the upper line and a counter ticking every second in the lower one.
Additionally, present a label `LEFT/RIGHT/UP/DOWN` when a key is pressed on the 6 keys keypad, located down to the LCD display module. The label should appear in a lower line of the LCD, along with a ticking counter that is still working. The label should disappear when the key is released.

1.7.2.1. The LCD Display

To control LCD, it is easiest to use a software library (add to the `platformio.ini` file:

```
...
lib_deps = arduino-libraries/LiquidCrystal @ ^1.0.7
...
```

Your LCD shield is 2×16 characters. To initialize it use the following GPIO definition:

```
const int rs = 12, en = 13, d4 = 4, d5 = 0, d6 = 2, d7 = 14;
LiquidCrystal lcd(rs, en, d4, d5, d6, d7);
```

Remember to initialize it in your code; you need to tell the library the physical organization of the display:

```
...
  lcd.begin(16,2); //2 lines, 16 characters each
...
```

1.7.2.2. Keypad

The LCD shield has 6 buttons. One is hard-wired to the reset function, but other buttons (marked SELECT, UP, DOWN, RIGHT, LEFT) can be read through voltage analysis delivered into the AD0 (Analogue to Digital Converter, Channel 0) existing in the ESP8266 chip. Buttons are connected the way that they short the connection, and every button creates a separate voltage divider between pull-up and respectable resistance, different for each button (see figure 7 for connection diagram). To estimate readings, you can calculate the voltage appearing on the AD0 input using Ohm's law.

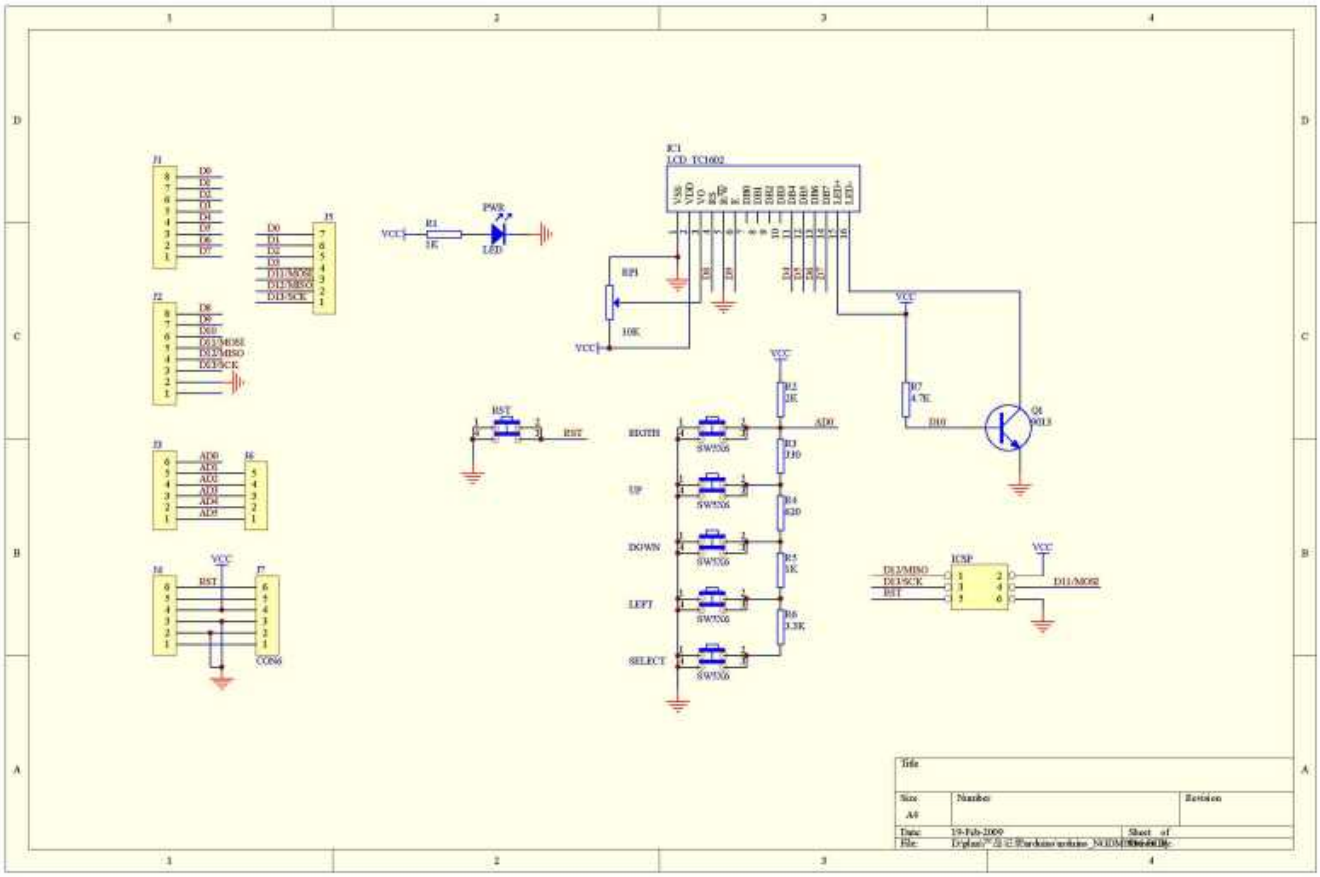


Fig. 7: 2×16 LCD shield connection diagram including buttons

Note there is an assumption that you press only one button at a time. If no button is pressed, Vcc goes to the AD0, and you get maximum reading (see schematics on figure 7 above and warning note below, on readings AD0).



Because of the LCD shield construction and the fact that ESP8266 can accept out of the box on their AD0 input voltage range of 0..3.3V only (actually 0..1V but there is a voltage divider in the middle), you're unable to grab button SELECT pressed, because shorting it causes input voltage to on the AD0 to be ~3.7V, while when no button is pressed, the input voltage on AD0 is 5V. Both are far beyond the measurable range, so you will always get maximum reading (1024) in both cases, and there is no way to distinguish them without electrical modification.

Reading an AD0 is as simple as:

```
#define ADCPIN 0
...
  int nAdcReading = 0;
...
  nAD0 = analogRead(ADCPIN);
...
```

Report:

- Describe how you solved the problem of the asynchronous key presses/releases processing.
- Include source code.

1.7.3. Scenario Lab3

** Presenting temperature and humidity on the local LCD screen **
In this scenario, you will measure temperature and relative humidity using the attached DHT 11 and present it on the screen. In the first row, there should appear a label `Temperature:` or `Humidity:` while in the lower, measured value, right-aligned along with the measurement unit. Switching between presenting temperature and humidity should be possible using the Up/Down keys from the keypad and changing the temperature between Celsius and Fahrenheit using the Left/Right keys. Measurements should include a unit: `C/F` (for temperature) and `%Rh` for humidity. The temperature should be present with one decimal point, while humidity as an integer.

1.7.3.1. Temperature and humidity sensor

There is a DHT11 (blue) sensor connected to the GPIO 5.
There are two libraries needed to handle reading from the sensor: a common sensor communication library and a specific one for DHT sensors, both by Adafruit, so two more entries are needed in the `platformio.ini` file, and you need to refer to it in the source code:

```
lib_deps =
    adafruit/Adafruit Unified Sensor@^1.1.4
    adafruit/DHT sensor library@^1.4.3
```

```
#include <Adafruit_Sensor.h>
#include <DHT.h>
```

To access sensor readings, it is necessary to instantiate a communication class that binds the hardware to the code:

```
#define DHTTYPE DHT11   // DHT 11
#define DHTPIN 5
DHT dht(DHTPIN, DHTTYPE,50);
```

It is also necessary to initialize the sensor class and give it some time to set up:

```
...
dht.begin();
delay(1000);
...
```

The sensor class returns data (among others) in a float containing temperature (in C) and relative humidity (in percent):

```
...
  float hum = dht.readHumidity();
  float temp = dht.readTemperature();
...
```

Report:

- Present a circuit with connections (LCD, keypad, and sensor).
- Present an algorithm for both overall program flow and for obtaining measurements.
- Include your code with appropriate comments.

1.7.4. Scenario Lab4

** Connecting to the network **
Connect your ESP8266 (as a client, STA mode) to the PRIVATE *internal.IOT* network with an address obtained from the DHCP server. Present the connection status (`connected/disconnected`) in the upper row of the LCD and your `IP address` in the second line.

Report:

- When using ESP in STA mode, observe if the consequent running of the solution brings new or the same IP address attached via the DHCP server.
- Include source code.

1.7.5. Scenario Lab5

Sending data to the network
In this scenario, you will implement a fully functional IoT end-node, sending measurements to the MQTT Broker. Use your solution from the last step of Lab4 and also Lab3. Measure temperature and humidity, present them on the screen as in the case of the Lab3 scenario, and parallel, publish them as MQTT messages to the Broker.



Mind agreeing with other sections to let you do not overlap with MQTT Message Topics. They should be unique for all teams!

Using any software-based MQTT client, subscribe to the messages using a PRIVATE or PUBLIC network and observe results. You can use a PC computer you're developing your solution for this task.

The simplest starting point is to use the pub-sub MQTT client library:

```
#include <ESP8266WiFi.h>
#include <PubSubClient.h>
```

You may also want to define MQTT topic(s) and constant payloads (i.e., those related to the last-will flag) using `#define` statement. It is also a good idea to put connection constants here:

```
//WiFi and MQTT server credentials
#define wifi_ssid "internal.IOT"
#define wifi_password "IoTlab32768"
#define mqtt_server "192.168.90.5"
#define mqtt_user "vrel"
#define mqtt_password "vrel2018"
//MQTT last will
#define lastWillTopic "/sut/aeii/iot-lab/321/node1/state"
#define lastWillMessage "off"
#define mqttWelcomeMessage "on"
```

Mind, `192.168.90.5` is your MQTT server address.

To use the pub-sub client, you must instantiate a global object using a network client, i.e., `WiFiClient`:

```
...
// WiFi & MQTT
WiFiClient espClient;
PubSubClient client(espClient);
...
```

Then connect your MQTT pub-sub client to the server. You can define a separate function in case you need to repeat this process when the connection is down (for any reason) to simplify reconnect procedure:

```
void reconnect() {
  // Loop until we're reconnected
  while (!client.connected()) {
    Serial.print("Attempting MQTT connection...");
    if (client.connect(<clientname>, mqtt_user, mqtt_password)) {
      Serial.println("connected to MQTT");
    } else {
      Serial.print("failed, rc=");
      Serial.print(client.state());
      Serial.println(" try again in 5 seconds");
      // Wait 5 seconds before retrying
      delay(5000);
    }
  }
}
```

Advanced:

- Implement a “last will” informing that device has lost connection (suddenly, not via disconnect)—present the last available measurements as persistent. Present the device state by changing a persistent message on connect/disconnect to let the subscribers know the state of your device. To implement controlled disconnect, use *select* key from the keypad (you can assume that you may need to restart the device to get it back on the network, but you also may implement a bistable logic: the first click of the *select* triggers MQTT disconnect, the second click connects it back to the Broker).

To publish a last-will message, you may use an extended version of the `PubSubClient.connect()` function (i.e., in your `reconnect()` function:

```
...
client.connect(<clientname>, mqtt_user, mqtt_password, lastWillTopic, 0,
true, lastWillMessage)
...
```

And once connected, you may notify the dashboard (if any) that you're back online:

```
...
client.publish(lastWillTopic, mqttWelcomeMessage, true);
...
```

Report:

- Provide your MQTT message Topics and Payload formats (semantics).
- Include source code.

From:
https://dokuwiki.zmitac.aei.polsl.pl/ - **RAU6 Dokuwiki**

Permanent link:
https://dokuwiki.zmitac.aei.polsl.pl/en:edu:zmitac:macro:iot:lab?rev=1682087420

Last update: **2023/04/21 16:30**