Table of Contents

1. Laboratory 1 - Introduction 2
1.1. Hardware 2
1.2. VSCode, PlatformIO and project configuration 3
1.2.1. Running your code 4
1.2.2. Hardware-software components and libraries 5
1.3. Software 7
1.4. Tracing 8
1.5. Networking 9
1.6. Network services 9
1.6.1. MQTT 9
1.6.2. Node Red and Node Red Dashboard 9
1.7. Laboratory scenarios 10
1.7.1. Scenario 1 11
1.7.2. Scenario 2 12
1.7.3. Scenario 3 13

1. Laboratory 1 - Introduction

Introduction to the IoT programming.

During labs, you will likely use Visual Studio Code with PlatformIO plugin, to author code for the IoT device. Devices are connected to the PC via USB cable but they expose a serial port to the operating system. You need to check the COM port number as it varies from PC to PC, and configure your project accordingly.



You can use default COM port transfer speed but in any case, when using COM port as a trace, you need to configure the transmission speed on the code side. Default is 9600 (bps).



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

1.1. Hardware

Each team is given the following IoT microcontroller:

- Wemos D1 R2 development board:

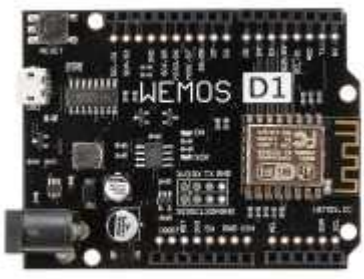


Fig. 1: WeMos D1 R2 development board

Note, this microcontroller even if its board reminds Arduino Uno has 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:

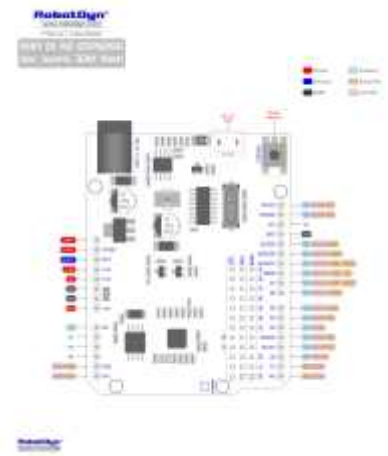


Fig. 2: WeMos D1 R2 pinout

- DHT 11 temperature and humidity sensor



Fig. 3: DHT 11 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 settings, regarding Board and Framework (see figure 5):

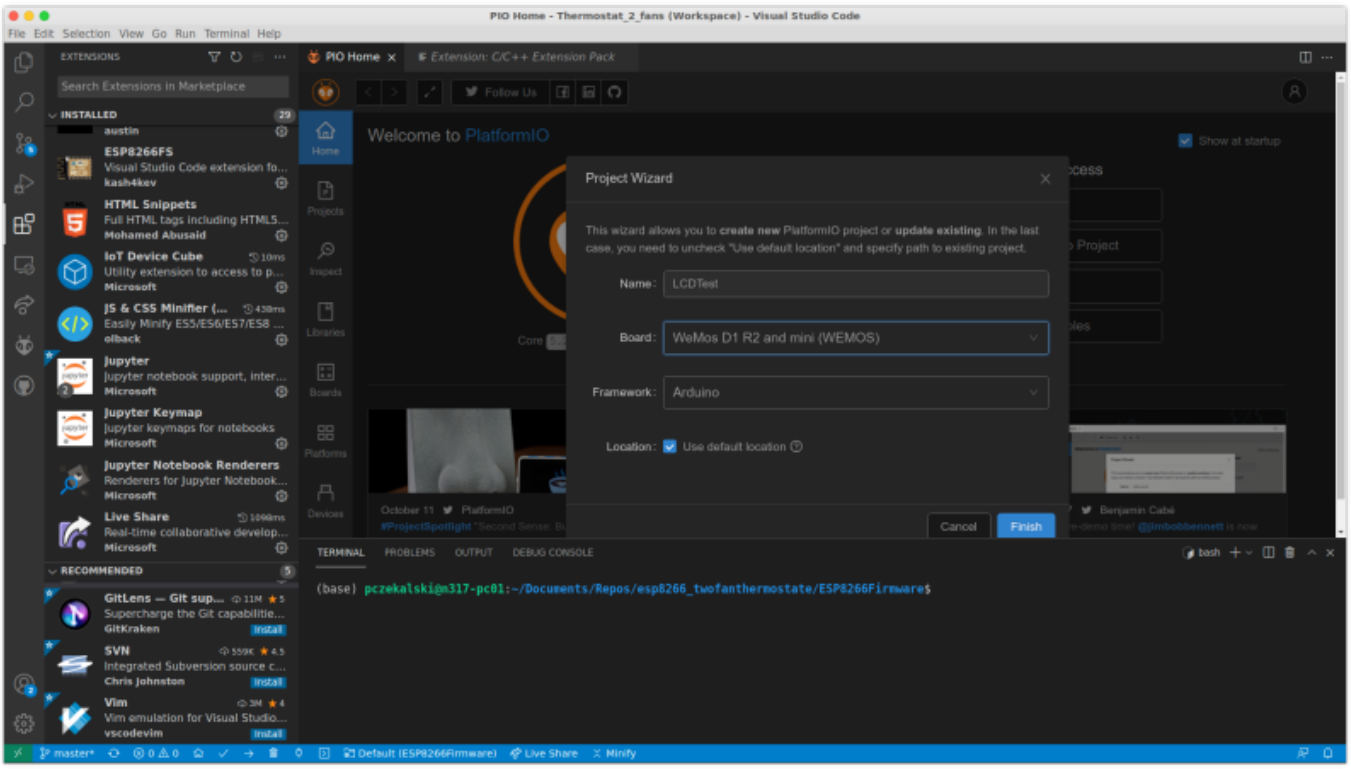


Fig. 5: Project configuration settings in VSCode/PlatformIO

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 that is used for uploading your firmware and to monitor. In the case of this microcontroller, it is the same port, so `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:

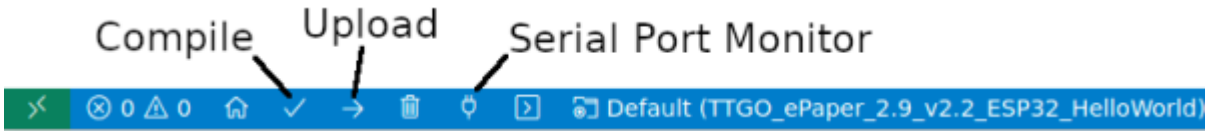


Fig. 6: PlatformIO ribbot to execute most common actions

1.2.2. Hardware-software components and libraries

Interfacing hardware requires the use of particular GPIOs. Laboratory nodes and hardware used needs specific configuration. Following chunks of code will help to set up proper hardware-software configuration.

1.2.2.1. LCD display

Built-in LED on the microcontroller development board can help to manage simple tracing of the code execution:

```
#define LED_BUILTIN 14
```

Your LCD shield is a 2×16 character one, driven by a Hitachi HD44780 controller. You can use a library to use it easily. The PlatformIO will pull the library automatically if defined in *platformio.ini* file:

```
lib_deps = arduino-libraries/LiquidCrystal
```

```
#include <LiquidCrystal.h>
```

To initialise it and instantiate controller object, use following GPIO definition and initialisation:

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

It is also necessary to tell the library about the display organisation as there is a variety of configurations possible. Here, the organisation is 2 lines, 16 columns each, so in the code you need to explicitly initialise it:

```
...
lcd.begin(16,2);
...
```

1.2.2.2. Keypad

The LCD shield has 6 buttons. One of them 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 voltage appearing on the AD0 input using Ohm's law.

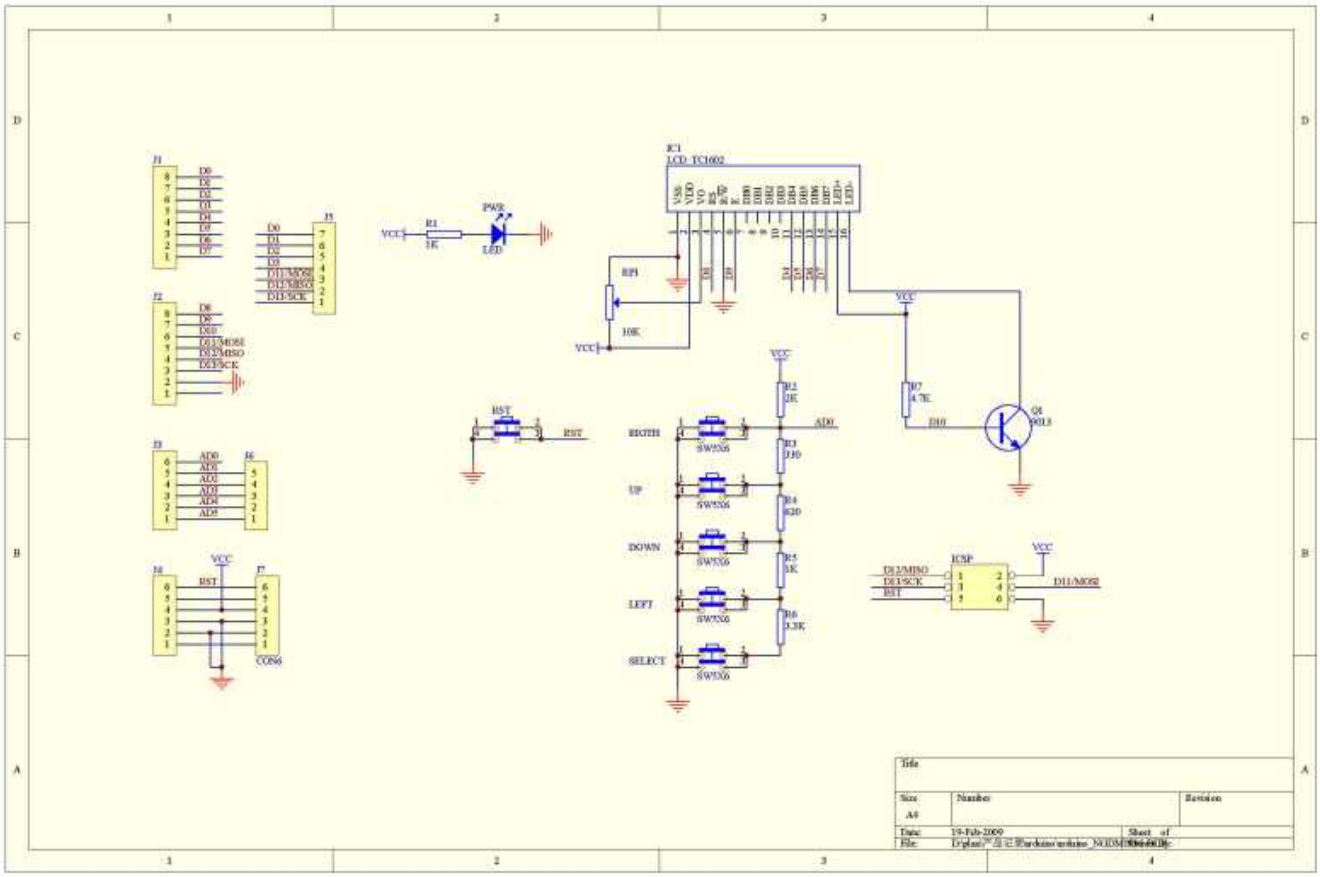


Fig. 7: 2×16 LCD shield connection diagram including buttons

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);
...
```

1.2.2.3. 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 for DHT sensors, both by Adafruit, so two more entries are needed in the *platformio.ini* file, and obviously 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 initialise 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 per cent):

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

1.3. Software

String formatting may be a challenge. An easy approach is to allocate a character buffer of the size of the LCD or just one display's line, or any other size, large enough to handle your text.

```
...
char buffer [50];
...
```

To convert float into the string with formatting use `sprintf` function:

```
...
```

```
  sprintf(buffer,"%2.1f C",temp);
...
  sprintf(buffer,"%2.1f%%Rh",hum);
...
```



`sprintf` uses number of wildcards that are rendered with data. Refer to the c/c++ documentation on `sprintf`. Here

```
%2.1f
```

means: having float number render it to string using two digits before decimal point and one after. Temperature in our lab is always above 10C.

```
%%
```

is an escape character to print a

```
%
```

(percent) symbol.

```
delay(time)
```

is measured in milliseconds.

1.4. 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, the same that is used for flashing. To open serial communication, one must invoke begin with 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 with no need to explicitly convert to string.

1.5. 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 to it can talk to 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 present in the network have fixed IP addresses, overlapping it causes unavailability of your solution and this service, also to other teams! Use the 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

1.6. Network services

1.6.1. MQTT

There is an MQTT Broker, binding WiFi PRIVATE network and the PUBLIC Internet. 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 Internet.

You can access it from the Internet 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 it from the aforementioned 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

1.6.2. Node Red and Node Red Dashboard

A Node RED server is working on same Raspberry Pi 2 as MQTT and in private address pool only, available through *internal.IoT* on 192.168.90.5 address, port 1880:

- http://192.168.90.5:1880 - main editor

- http://192.168.90.5:1880/ui - Node RED dashboard

Login using the following credentials:

- User: *contact supervisor*
- Password: *contact supervisor*

1.7. Laboratory scenarios

All teams share the same scenario yet please note that in particular, 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 every next laboratory meeting you will extend your code by something extra.

1.7.1. Scenario 1

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 colour is blue.

```
...
#define LED_BUILTIN 14
...
```

The standard serial port speed is 9600 (bps). Connect COM port monitor (a plug icon in the PlatformIO toolbar), once you compile and upload your code, to see the “Hello World” message.

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

Advanced:

- Use 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 2

** Using LCD **
In this scenario, you will write an application that writes 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.

Report:

- Present algorithm.
- Include source code.

1.7.3. Scenario 3

** Using Keypad **
Extend Scenario 2 with a counter that increments on the keypress of the Keypad Shield (any but SELECT!).

Report:

- Describe how did you solve the problem of the asynchronous processing of the keypresses/releases.
- Include source code.

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

Permanent link:
https://dokuwiki.zmitac.aei.polsl.pl/en:edu:zmitac:s2:iotn:lab1?rev=1647551831

Last update: **2022/03/17 22:17**