Table of Contents

1. Laboratory on ESP 8266 on-site 2
1.1. Laboratory hardware in detail 2
1.1.1. Microcontroller development board 2
1.1.2. Temperature and humidity sensor 3
1.1.3. 2x16 LCD display shield 3
1.2. Software 4
1.2.1. VSCode, PlatformIO and project configuration 4
1.2.1.1. Running your code 5
1.2.1.2. Libraries 6
1.2.1.3. Hardware-software components and libraries 6
1.2.2. Software 9
1.2.3. Tracing 10
1.3. Tasks to do during labs 11
1.3.1. Task 1 11
1.3.1.1. Advanced 11
1.3.2. Task 2 11
1.3.3. Task 3 11

1. Laboratory on ESP 8266 on-site

While ESP 8266 (Espressiff) is, in general, a network processor with broad applications in IoT, here we focus on using it as an embedded-class microcontroller.

1.1. Laboratory hardware in detail

Each team is given the following configuration with a pre-built hardware stack.

1.1.1. Microcontroller development board

Wemos D1 R2 development board:

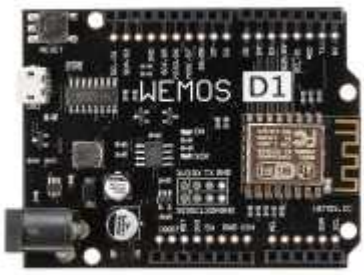


Fig. 1: WeMos D1 R2 development board

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:

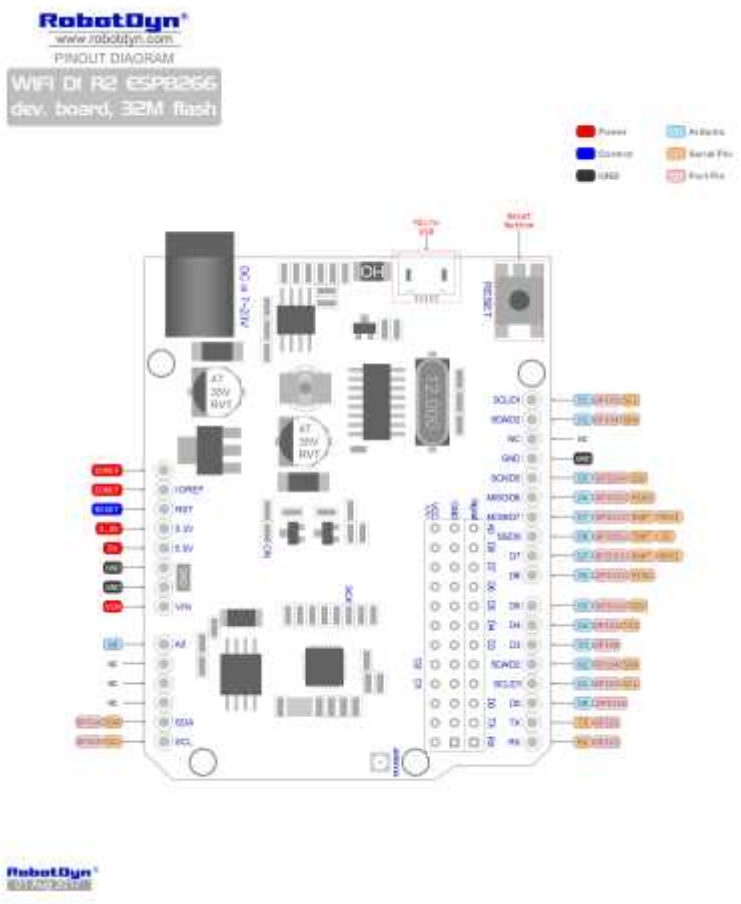


Fig. 2: WeMos D1 R2 pinout © RobotDyn

1.1.2. Temperature and humidity sensor

DHT 11 temperature and humidity sensor



Fig. 3: DHT 11 temperature and humidity sensors



Do not try to read temperature and humidity too frequently. Once every 5s is enough for information and safe enough not to let your readings race with the communication protocol. If you read too frequently, your sensor may not be able to deliver data on time, and you will obtain a 'nan' (not a number) instead of temperature and humidity.

1.1.3. 2x16 LCD display shield

2x16 characters LCD shield with keypad (buttons)



Fig. 4: LCD shield 2×16 characters

1.2. Software

You will be using Visual Studio Code with the PlatformIO plugin to author software for the ESP8266. The framework (software stack) is Arduino-compatible with the popular source code model:

```
#include <Arduino.h>

void setup() {
  // put your setup code here to run once:
}

void loop() {
  // put your main code here to run repeatedly:
}
```

Firmware uploading is done via a USB cable providing a COM (serial) port to the PC computer you're developing software on. You can also use the same port to trace (send communications) from your code to the console (so-called “serial monitor”).



Computers in laboratory rooms 320 and 321, and 336 use “freezing” software, meaning that computers are restored to their initial state for every startup. All changes done to the local storage are then lost! Keep your work in some other place between meetings i.e. AEI GIT, Google Drive and so on.

1.2.1. VSCode, PlatformIO and project configuration

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

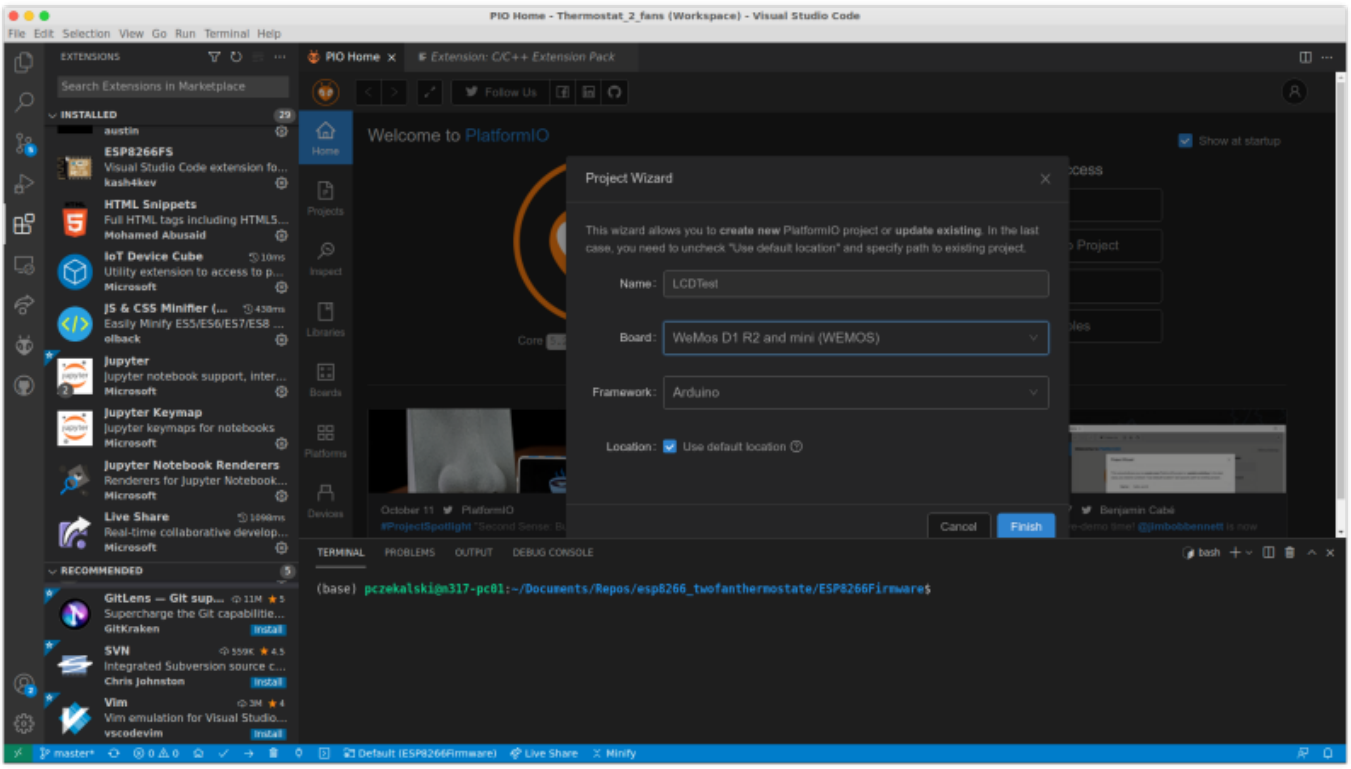


Fig. 5: Project configuration settings in VSCode/PlatformIO

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 monitoring. 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
...
```



Check your COM port in the Device Manager (Windows) or terminal (dev for Linux / macOS). It can be different than in the example above!

1.2.1.1. 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:

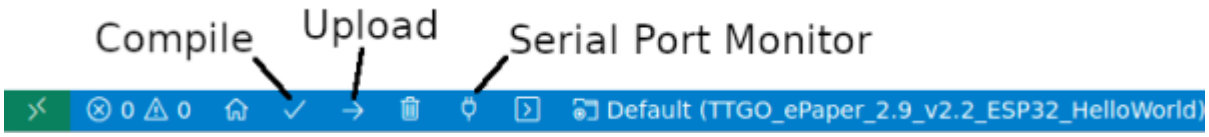


Fig. 6: PlatformIO ribbot to execute most common actions

1.2.1.2. Libraries

There is a number of built-in libraries that come with an already installed framework (ESP8266). The obligatory one is `Arduino.h` that brings a number of useful constants and routines like i.e. `delay(…)`. It is strongly advised to include it in the source code and the majority of implementations will require it:

```
#include <Arduino.h>
```

External code libraries can be added as per project or per system. They help to automate tasks and interface hardware. The preferred method is *per project*, dynamically downloaded during compilation. To add a library, find one in the “Libraries” section in the PlatformIO Home tab, then click the library and then click the button: “Add to Project” (7):

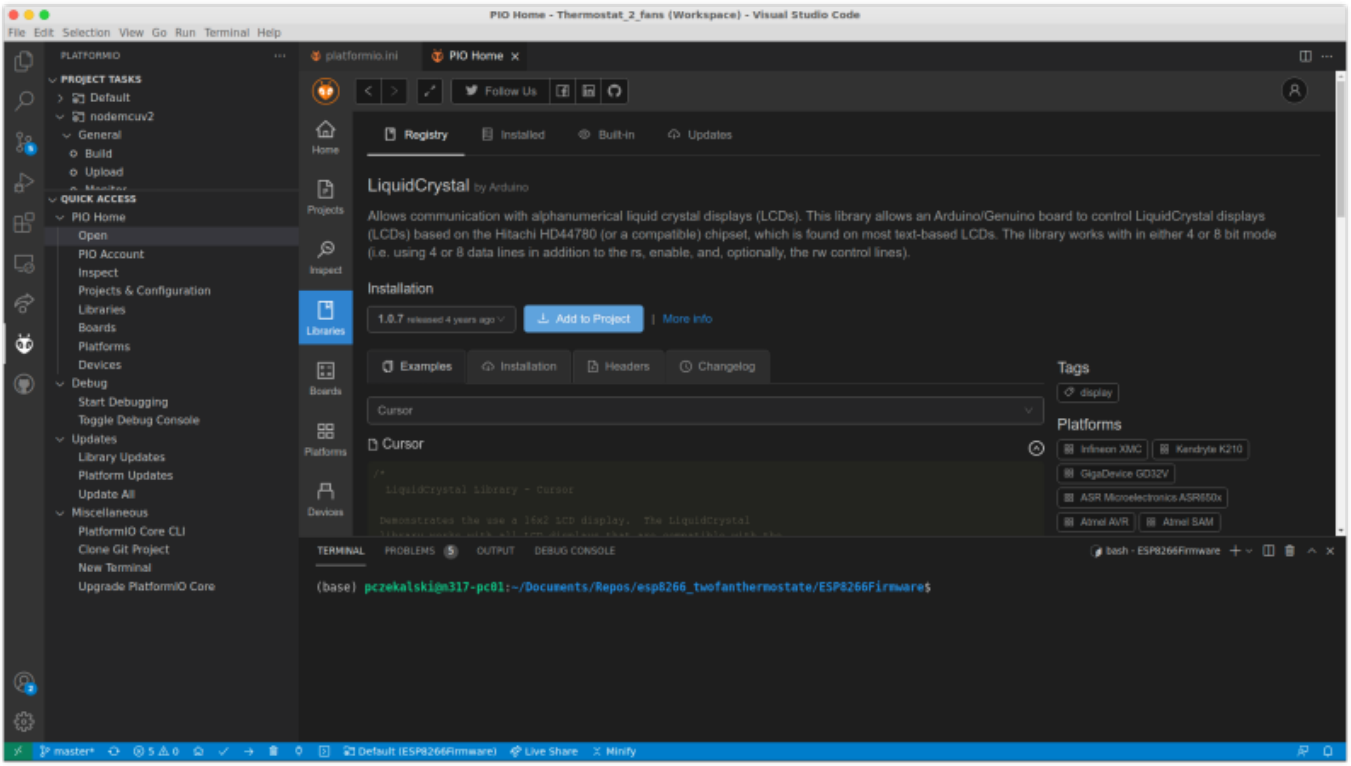


Fig. 7: Adding a code library to the project

Adding libraries to the project causes the following entry to appear in the `platformio.ini` file:

```
lib_deps = ...
```

External libraries will help you to interface i.e. LCD display and DHT sensors, as presented below.

1.2.1.3. 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.

1.2.1.3.1. 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);
...
```

Some useful methods of the `LiquidCrystal` class include (among others):

```
...
lcd.setCursor(x,y);
...
lcd.print(...);
...
```



LCD module has a standard ASCII (US) set of characters embedded. It is possible to re-program it, i.e. for Polish-specific letters like ąęółśćżź and virtually any other symbol. If you do not re-program it, pleas use English alphabet only for messaging.

1.2.1.3.2. Keypad

The LCD shield has six 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 8 for connection diagram). To estimate readings, you can calculate the voltage appearing on the AD0 input using Ohm's law.

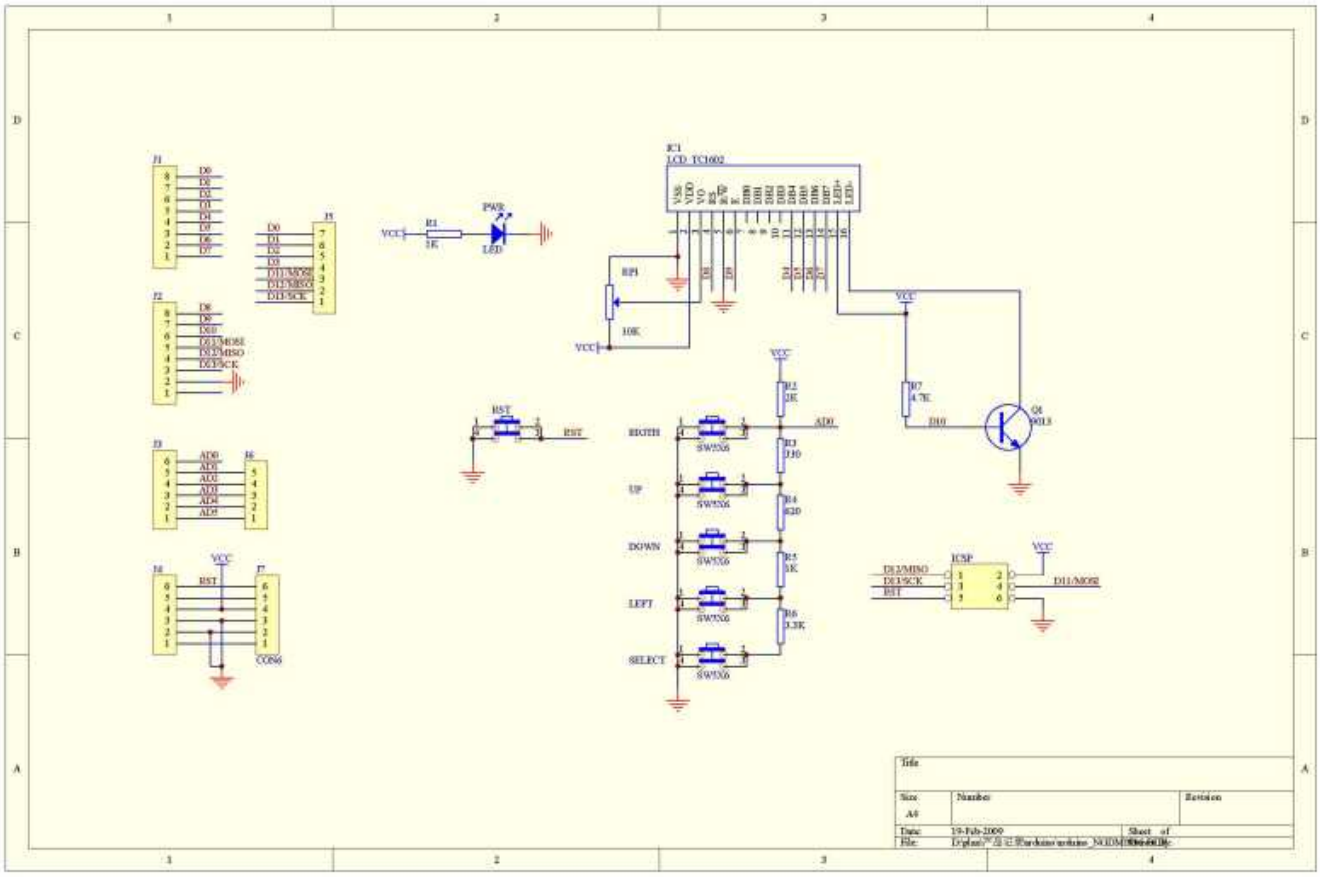


Fig. 8: 2×16 LCD shield connection diagram including buttons

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 8 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;
...
  nAdcReading = analogRead(ADCPIN);
...
```

1.2.1.3.3. 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();
...
```

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

```
%2.1f
```

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

```
%%
```

is an escape character to print a

```
%
```

(percent) symbol.

```
delay(time)
```

is measured in milliseconds.

1.2.3. 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 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 with no need to convert to string explicitly.

1.3. Tasks to do during labs

1.3.1. Task 1

Write on the LCD screen your name (in the first line) and the city name you live in (in the second line).

1.3.1.1. Advanced

Make it animated (each line) in the loop, towards the left or right.

1.3.2. Task 2

Present on the LCD screen set of current environmental parameters: temperature and humidity, as read from the DHT sensor. In the first line, there should be headers stating: `Temp.: ''` and then `temperature that rounds to 1 decimal place, and then the unit ''C` (in Celsius). Use string formatting functions like `sprintf` and so on. The second line of the LCD screen should present relative humidity, in the following form `Hum.:`, then relative humidity in per cent (rounded to the integer) and then unit as %Rh. Update information at most every 1s, but on the other hand, short enough to notify changes when keeping the sensor in your hand or blowing into it. 5s seems to be a reasonable delay between consecutive readings.

1.3.3. Task 3

Present on the LCD screen the absolute value of the ADC 0 (analogue to digital converter) value and the button name that has been pressed. The Upper LCD screen line should present current AD0 readings, and the lower should present decoded button name. Mind, every LCD shield is slightly different, so you may need to run an absolute AD0 value reader to grab values reported when particular buttons are pressed, then update your solution with a decoder function that will return the button name, as detected by the absolute AD0 readings. Note that ESP8266's default AD0 resolution is 10 bits, so AD0 reading can range from 0 to 1024.



One of the buttons won't work. It is natural to ask the supervisor for details if you're interested in why.



Bonus question, why is there 1024 reading, while the lowest is 0 and it is 10 bits range (so 0..1023 is valid)?

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

Permanent link:
https://dokuwiki.zmitac.aei.polsl.pl/en:edu:zmitac:macro:embedeed:lab:esp-on-site

Last update: **2023/04/05 12:23**