University Côte d'Azur

Polytech Nice Sophia

Department
Engineering of Electronic Systems

Labs
IoT Architecture

Fabrice MULLER

🖂 : Fabrice.Muller@univ-cotedazur.fr

- 2023/2024 -

Contents

Part I

Espressif Framework

LAB N° 1

Framework

Lab Objectives

- Understand the Espressif IoT Development Framework.
- Run a first program.
- Create an GitHub repository.
- Use the Microsoft Visual Studio Code with a dedicated ESP32 project template.

1.1 Espressif IoT Development Framework

We will start by understanding the structure of the Espressif IDF framework using an example provided by Espressif.

1.1.1 Quick install of Espressif IoT Development Framework 4.3

We are going to install version 4.3 of Espressif IoT Development Framework. The complete documentation is available online from this link. if you have difficulty to install it, go to the **Get Started** section otherwise, open a terminal and follow the script below :

- Install prerequisites

```
esp32:~$ sudo apt-get install git wget flex bison gperf python3 python3-pip
    python3-setuptools cmake ninja-build ccache libffi-dev libssl-dev dfu-
    util libusb-1.0-0
```

- Get ESP-IDF

```
esp32:~$ mkdir -p ~/esp
esp32:~$ cd ~/esp
esp32:~$ git clone -b v4.3 --recursive https://github.com/espressif/esp-idf.
    git
```

- Set up the tools

```
esp32:~$ cd ~/esp/esp-idf
esp32:~$ ./install.sh
```

- Set the rights for USB driver and debug tools

```
esp32:~$ sudo usermod -a -G dialout $USER
esp32:~$ sudo usermod -a -G uucp $USER
esp32:~$ sudo usermod -a -G plugdev $USER
esp32:~$ sudo cp ~/.espressif/tools/openocd-esp32/v0.10.0-esp32-20210401/
    openocd-esp32/share/openocd/contrib/60-openocd.rules /etc/udev/rules.d/
```

- Add the environment
  - Open *.bashrc* file.

    ```
    esp32:~$ gedit ~/.bashrc
    ```

  - Copy this line below at the end of the file.

    ```
    . $HOME/esp/esp-idf/export.sh
    ```

- Reboot the computer

```
esp32:~$ reboot
```

- Open a terminal. verify the environment. You should see the lines for the environment configuration of Espressif IoT Development Framework as below :

```
Setting IDF_PATH to '/home/esp32/esp/esp-idf'
Detecting the Python interpreter
Checking "python" ...
Python 3.6.9
"python" has been detected
Adding ESP-IDF tools to PATH...
Using Python interpreter in /home/esp32/.espressif/python_env/idf4.3_py3.6
    _env/bin/python
Checking if Python packages are up to date...
Python requirements from /home/esp32/esp/esp-idf/requirements.txt are
    satisfied.
Added the following directories to PATH:
  /home/esp32/esp/esp-idf/components/esptool_py/esptool
  ...
  /home/esp32/.espressif/python_env/idf4.3_py3.6_env/bin
  /home/esp32/esp/esp-idf/tools
Done! You can now compile ESP-IDF projects.
Go to the project directory and run:

  idf.py build
```

You are now ready to use the Espressif IoT Development Framework !

1.1.2 Creation of GitHub repository for Labs

We will have many Labs. Thus, we are going to create a GitHub repository *labs* in the *esp* folder that will contain all the Labs.
Firstly, in the WEB interface of GitHub (open with Google Chrome or another navigator), you have to create a new repositories in GitHub, for example « labs » (use the same name of your labs folder, normally « labs »). **Configure your GitHub in private access** with a *README.md* file as shown the figure 1.1

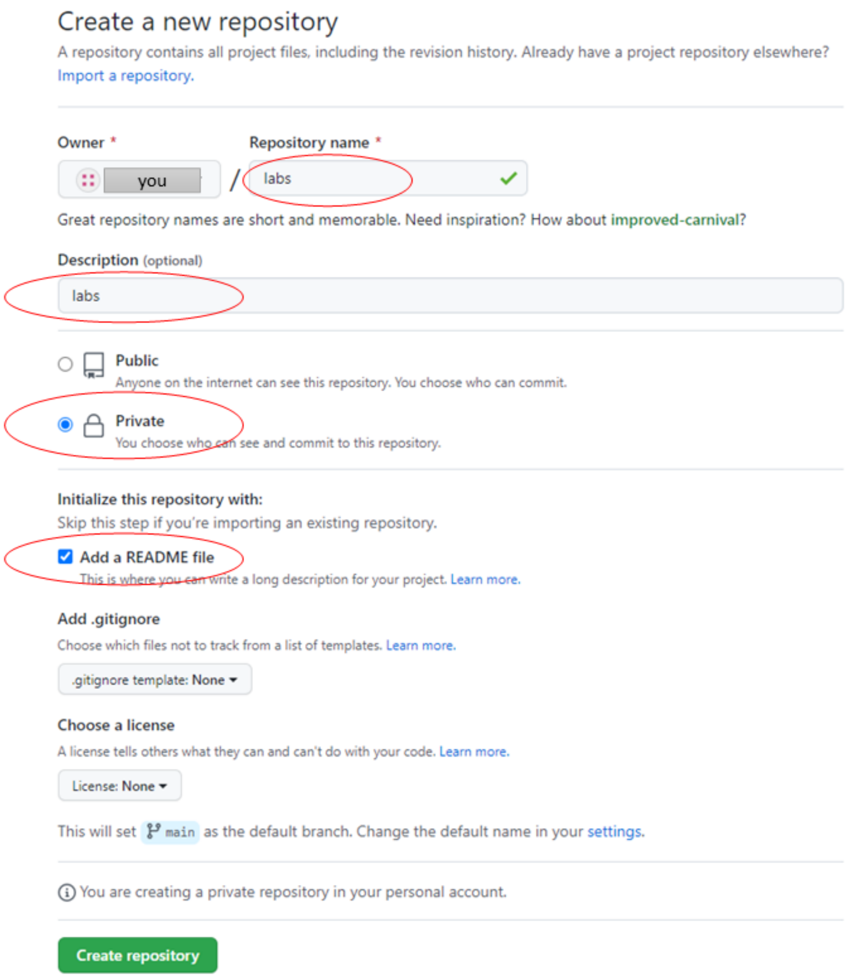


FIGURE 1.1 – *Create a GitHub repository.*

Secondly, you should create a personal access token to use in place of a password to access of your GitHub folder. Follow the instructions to generate a token (select only *repo* and *user*).

Thirdly, in a terminal, follow the steps to clone your new GitHub repositories in your computer :

1. You start cloning your « labs » repository to the computer. To obtain the URL of your repository, copy the repository URL located on GitHub webpage. The example below shows you the principle when you have to replace <your owner> by your GitHub owner.

```
esp32:~/$ cd ~/esp
esp32:~/esp$ git clone https://github.com/<your owner>/labs
```

2. You can now configure your name and email address for GIT in the new « labs » repository.

```
esp32:~/esp$ cd labs
esp32:~/esp/labs$ git config --global user.name "your name"
esp32:~/esp/labs$ git config --global user.email "your email address"
```

3. You could enter this command to avoid typing your *username* and *token* each time in Visual Studio Code.

```
esp32:~/esp/labs$ git config credential.helper store
```

You have information for configuring GIT for your new project.

1.1.3 First look of the first example

Take the example « hello_world » which displays the string « hello world ! » and characteristics of the ESP32 board on the console. The example is located in the following directory :

```
esp32:~$ cp -R ~/esp/esp-idf/examples/get-started/hello_world ~/esp/labs/
    hello_world
esp32:~$ cd ~/esp/labs/hello_world
```

The compilation is done from a Python script called *idf.py*. This script is located in `~/esp/esp-idf/tools/` and added in the path.

```
esp32:~/esp/labs/hello_world$ which idf.py
/home/esp32/esp/esp-idf/tools/idf.py

esp32:~/labs/hello_world$ env | grep esp-idf
IDF_TOOLS_EXPORT_CMD=/home/esp32/esp/esp-idf/export.sh
PWD=/home/esp32/esp/labs/hello_world
IDF_TOOLS_INSTALL_CMD=/home/esp32/esp/esp-idf/install.sh
IDF_PATH=/home/esp32/esp/esp-idf
PATH=/home/esp32/esp/esp-idf/components/esptool_py/esptool:/home/esp32/esp/esp-idf/
    components/espcoredump:/home/esp32/esp/esp-idf/components/partition_table/:/
    home/esp32/.espressif/tools/xtensa-esp32-elf/esp-2019r2-8.2.0/xtensa-esp32-elf/
    bin:/home/esp32/.espressif/tools/esp32ulp-elf/2.28.51.20170517/esp32ulp-elf-
    binutils/bin:/home/esp32/.espressif/tools/openocd-esp32/v0.10.0-esp32-20190313/
    openocd-esp32/bin:/home/esp32/.espressif/python_env/idf4.0_py3.6_env/bin:/home/
    esp32/esp/esp-idf/tools:/home/esp32/.local/bin:/usr/local/sbin:/usr/local/bin:/
    usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin
```

To display the help of the Python script, just type the name of the command as below. We will mainly use the following commands : `build, flash, monitor, menuconfig, fullclean, size ...`

```
esp32:~/esp/labs/hello_world$ idf.py
Checking Python dependencies...
Python requirements from /home/esp32/esp/esp-idf/requirements.txt are satisfied.
Usage: /home/esp32/esp/esp-idf/tools/idf.py [OPTIONS] COMMAND1 [ARGS]... [COMMAND2
    [ARGS]...]...

  ESP-IDF build management

Options:

  -b, --baud INTEGER              Baud rate. This option can be used at most once
    either globally, or
                                  for one subcommand.
...
  -p, --port TEXT                 Serial port. This option can be used at most once
     either globally,
                                  or for one subcommand.
...
Commands:
  all                    Aliases: build. Build the project.
...
  clean                  Delete build output files from the build directory.
...
  flash                  Flash the project.
  fullclean              Delete the entire build directory contents.
  menuconfig             Run "menuconfig" project configuration tool.
  monitor                Display serial output.
...
  size                   Print basic size information about the app.
  size-files             Print per-source-file size information.
```

The C source files are usually located in the « main » folder. We see below the « hello_world_main.c » file. The other files will be studied later.

```
esp32:~/esp/labs/hello_world$ ll main

total 20
drwxr-xr-x 2 esp32 esp32 4096 avril  2 15:31 ./
drwxr-xr-x 4 esp32 esp32 4096 mai   26 10:07 ../
-rw-r--r-- 1 esp32 esp32   85 avril  2 15:31 CMakeLists.txt
-rw-r--r-- 1 esp32 esp32  146 avril  2 15:31 component.mk
-rw-r--r-- 1 esp32 esp32 1232 avril  2 15:31 hello_world_main.c
```

1.1.4 Building the first example

The generation of the executable in this specific case is called **cross-compilation** because the program will not be performed on the computer but on the ESP32 board. We build the executable from the following command.

```
esp32:~/esp/labs/hello_world$ idf.py build

[59/62] Linking C static library esp-idf/spi_flash/libspi_flash.a
[60/62] Linking C static library esp-idf/main/libmain.a
[61/62] Linking C executable bootloader.elf
[62/62] Generating binary image from built executable
esptool.py v2.8
Generated /home/esp32/esp/labs/hello_world/build/bootloader/bootloader.bin
[820/820] Generating binary image from built executable
esptool.py v2.8
Generated /home/esp32/esp/labs/hello_world/build/hello-world.bin

Project build complete. To flash, run this command:
/home/esp32/.espressif/python_env/idf4.0_py3.6_env/bin/python ../../../components/
   esptool_py/esptool/esptool.py -p (PORT) -b 460800 --before default_reset --
   after hard_reset write_flash --flash_mode dio --flash_size detect --flash_freq
   40m 0x1000 build/bootloader/bootloader.bin 0x8000 build/partition_table/
   partition-table.bin 0x10000 build/hello-world.bin
or run 'idf.py -p (PORT) flash'
```

A new « build » folder appears. In this folder, you can see the « hello-world.elf » which will be flashed in the ESP32 board.

```
esp32:~/esp/labs/hello_world$ ll
total 56
drwxr-xr-x  4 esp32 esp32  4096 mai   26 10:07 ./
drwxr-xr-x  4 esp32 esp32  4096 avril  2 15:31 ../
drwxr-xr-x 74 esp32 esp32  4096 mai   26 10:27 build/
-rw-r--r--  1 esp32 esp32   234 avril  2 15:31 CMakeLists.txt
drwxr-xr-x  2 esp32 esp32  4096 avril  2 15:31 main/
-rw-r--r--  1 esp32 esp32   183 avril  2 15:31 Makefile
-rw-r--r--  1 esp32 esp32   170 avril  2 15:31 README.md
-rw-r--r--  1 esp32 esp32 25463 mai   26 10:26 sdkconfig

esp32:~/esp/labs/hello_world$ cd build

esp32:~/esp/labs/hello_world/build$ ll hello_world*
-rw-r--r-- 1 esp32 esp32  147232 mai   26 10:27 hello_world.bin
-rwxr-xr-x 1 esp32 esp32 2451528 mai   26 10:27 hello_world.elf*
-rw-r--r-- 1 esp32 esp32 1541555 mai   26 10:27 hello_world.map

esp32:~/esp/labs/hello_world/build$ cd ..
```

1.1.5 Running the first example on ESP32 board

You find details of the ESP32-PICO-KIT board in the Getting Started Guide. To run the program on the board, follow the procedure below :

- Connect the ESP32 card to the computer via USB (cf. figure 1.2)

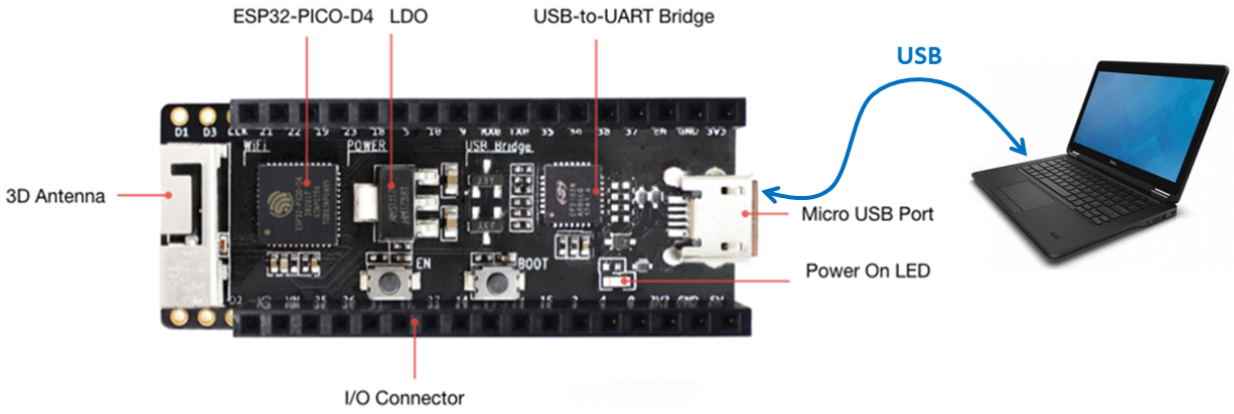


FIGURE 1.2 – *ESP32-PICO-KIT board connections.*

- Identify the USB serial port (usually /dev/ttyUSB0)

```
esp32:~/esp/labs/hello_world$ ls /dev/ttyUSB*
/dev/ttyUSB0
```

- Flash the board and push the BOOT button to launch the programming if necessary (depending on your USB configuration)

```
esp32:~/esp/labs/hello_world$ idf.py -p /dev/ttyUSB0 flash
esptool.py -p /dev/ttyUSB0 -b 460800 --before default_reset --after
   hard_reset write_flash --flash_mode dio --flash_freq 40m --flash_size 2MB
    0x8000 partition_table/partition-table.bin 0x1000 bootloader/bootloader.
   bin 0x10000 hello-world.bin
esptool.py v2.8
Serial port /dev/ttyUSB0
Connecting........_____....._
Detecting chip type... ESP32
Chip is ESP32-PICO-D4 (revision 1)
...
Compressed 147232 bytes to 76527...
Wrote 147232 bytes (76527 compressed) at 0x00010000 in 1.7 seconds (effective
    675.4 kbit/s)...
Hash of data verified.
Leaving...
Hard resetting via RTS pin...
Done
```

- Monitor console messages sent by the program running on the ESP32 card. To exit monitoring, typing « Ctrl+AltGr+] »

```
esp32:~/esp/labs/hello_world$ idf.py -p /dev/ttyUSB0 monitor
...
I (294) spi_flash: flash io: dio
W (294) spi_flash: Detected size(4096k) larger than the size in the binary
    image header(2048k). Using the size in the binary image header.
I (304) cpu_start: Starting scheduler on PRO CPU.
I (0) cpu_start: Starting scheduler on APP CPU.
Hello world!
This is ESP32 chip with 2 CPU cores, WiFi/BT/BLE, silicon revision 1, 2MB
    embedded flash
Restarting in 10 seconds...
Restarting in 9 seconds...
Restarting in 8 seconds...
```

1.1.6 Push the code to GitHub

We are going to push (or save) your code to GitHub.

Firstly, you do not have the *.gitignore* file at the root the repository *Labs*. It is important to filter the files you want to push or save to Git.

- In the repository Labs, create the *.gitignore* file.

```
esp32:~/esp/labs$ gedit .gitignore
```

- Copy the contents of the *.gitignore* file in GitHub ESP32 project template from this link.

Then, we can commit and push your code.

```
esp32:~/esp/labs$ git add .gitignore
esp32:~/esp/labs$ git add *
esp32:~/esp/labs$ git status
esp32:~/esp/labs$ git commit -a -m "first commit"
esp32:~/esp/labs$ git push origin main
```

1.2 Visual Studio Code with ESP-IDF

In order to develop applications in a user-friendly way, we use Microsoft Visual Studio Code throughout these Labs. Moreover, we use a Visual Studio Code project template located in GitHub : `https://github.com/fmuller-pns/esp32-vscode-project-template`.

To use the template :

- Go to your repository where we will create the first lab named « part1_iot_framework ».

```
esp32:~$ cd labs
esp32:~/labs$ mkdir part1_iot_framework
esp32:~/labs$ cd part1_iot_framework
```

- Clone the template project named « Visual Studio Code Template for ESP32 »

```
esp32:~/labs/part1_iot_framework$ git clone https://github.com/fmuller-pns/
    esp32-vscode-project-template.git
Cloning into 'esp32-vscode-project-template'...
remote: Enumerating objects: 30, done.
remote: Counting objects: 100% (30/30), done.
remote: Compressing objects: 100% (23/23), done.
remote: Total 30 (delta 8), reused 23 (delta 4), pack-reused 0
Unpacking objects: 100% (30/30), done.
```

- List working directory

```
esp32:~/labs/part1_iot_framework$ ll
total 12
drwxr-xr-x  3 esp32 esp32 4096 mai   26 16:17 ./
drwxr-xr-x 24 esp32 esp32 4096 mai   26 16:16 ../
drwxr-xr-x  5 esp32 esp32 4096 mai   26 16:18 esp32-vscode-project-template/
```

- Rename the folder.

```
esp32:~/labs/part1_iot_framework$ mv esp32-vscode-project-template
    lab1_framework
```

- Delete *.git* folder of the new project « lab1_framework ». Be careful, **do not delete the *.git* folder** located in the « labs » folder.

```
esp32:~/labs/part1_iot_framework$ cd lab1_framework
esp32:~/labs/part1_iot_framework/lab1_framework$ rm -fR .git
```

- Open Visual Studio Code for the new project.

```
esp32:~/labs/part1_iot_framework/lab1_framework$ code .
```

- Add Extensions in Visual Studio Code (press keys : CTRL + MAJ + X). Search and install *C/C++ (Microsoft)* and *C/C++ Extension Pack (Microsoft)*
- Follow the section Getting Started to run the program on the board.
- Change the message in the *app_main()* function located in the *main.c* file.
- Build and run the program.
- You must commit and push the modification in GitHub. Follow the section Using GitHub with Visual Studio Code to do it.

Tip : Using a script to create an ESP32 project

We provide an alias *esp32-new-project*.

- Open a new terminal. You normally have an error because you must pass the name of the project.

```
esp32:~$ esp32-new-project
Error: add the name of the project
Example:  esp32-new-prj-template <my_project>
```

Now, you are ready to use Visual Studio Code with ESP-IDF for other projects !

Part II

Inputs Outputs

LAB N° 1

Digital to Analog Converter, Analog to Digital Converter & High Resolution Timer

Lab Objectives

- Using a Digital to Analog Converter (DAC)
- Using a Analog to Digital Converter (ADC)
- Creating, suspending and deleting a High Resolution Timer to generate waveforms

1.1 Preparation

Help you of the DAC documentation (DAC Web help) to answer the following questions :

- Study the parameter of the following function : *dac_output_enable()*. How many channels are there ? What is the correspondence with the pins of the GPIO ?

- Study the parameters of the following function : *dac_output_voltage()*. If the value of the second parameter (*dac_value*) is equal to 100, what is the value of the output voltage ?

Likewise, use the High Resolution Timer documentation (High Resolution Timer Web help) to answer these questions :

- What are the limitations of the FreeRTOS software timers compared to High Resolution Timer ?

- Study the *esp_timer_create()* function and its parameters. Take an interest in the *esp_timer_create_args_t* type and more particularly in the fields *callback* and *name*. What is the callback prototype ?

- What is the role of the *esp_timer_start_periodic()*, *esp_timer_stop()*,*esp_timer_delete()* functions.

Finally, use the ADC documentation (ADC Web help) to answer these questions :

- What is the type of ADC converter ?

- What is the maximum voltage range (in mV) of ADC converter ? With what attenuation ?

- What is the precision range (in bits) of ADC converter ?

1.2 DAC - Triangular wave generator (Lab1-1)

The goal is to create a triangular signal generator of 3.3v amplitude as shown in the figure 1.1. We will use the Channel 1 of the DAC. To test it, we connect a LED protected by a resistance to the output of the DAC.

1. Create a new folder named « part4_inputs_outputs ».
2. In the « part4_inputs_outputs » folder, create the « lab1-1_dac » lab from « esp32-vscode-project-template » GitHub repository.
3. Overwrite the « main.c » file by the provided code of the « lab1-1/main.c » file.
4. Copy the provided « lab1-1/sdkconfig.defaults » file to the project folder.
5. What is the maximum DAC value and the period of the timer to a period of 5 seconds

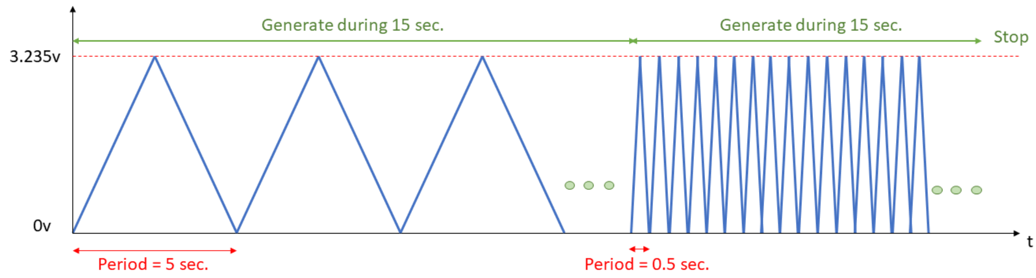


FIGURE 1.1 – *Specification of the triangular generator.*

and 0.5 seconds ? Justify.

6. What is the GPIO pin number used for the channel 1 ?

7. Write the code to generate the triangular waveform for the scenario :
   - Generate waveform (period = 5 sec.) during 15 sec.
   - Generate waveform (period = 0.5 sec.) during 15 sec.
   - Stop waveform
8. Perform the wiring on the board (resistance and LED).
9. Build and run the program.

1.3 ADC - Voltage divider bridge (Lab1-2)

The goal is to create a voltage divider bridge of V_{max} amplitude as shown in the figure 1.2. We will use the Channel 1 of the DAC. To test it, we connect a potentiometer protected by a resistance.

- In the « part4_inputs_outputs » folder, create the « lab1-2_adc » lab from « esp32-vscode-project-template » GitHub repository.
- Overwrite the « main.c » file by the provided code of the « lab1-2/main.c » file.
- Copy the provided « lab1-2/sdkconfig.defaults » file to the project folder.
- Calculate the normalized resistance value (in $K\Omega$) for the maximum voltage of the ADC.

- What return the *esp_timer_get_time()* function.

- To complete the DEFAULT_VREF in the *main.c* file, run the command below. What

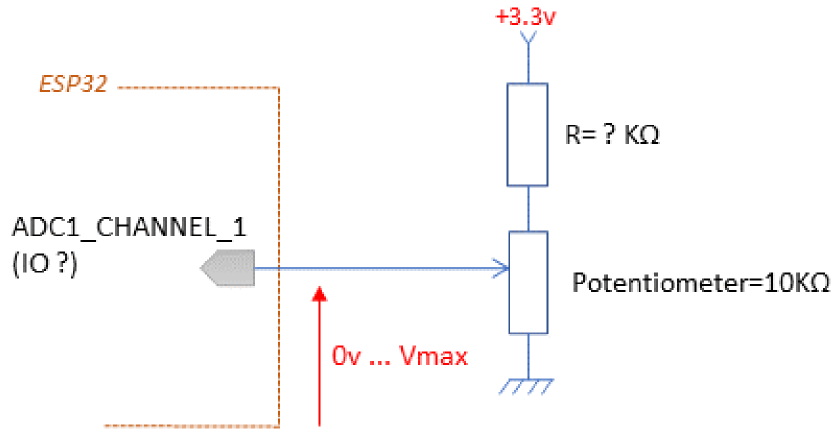


FIGURE 1.2 – *Schematic of the voltage divider bridge.*

is the VRef value ?
Find the *espefuse.py* script path as below :

```
esp32:~/../part4_inputs_outputs/lab1-2_adc$ find ~/ -name espefuse.py

/home/iot/esp/esp-idf/components/esptool_py/esptool/espefuse.py
```

Copy the result of the find command to perform the *espefuse.py* script.

```
esp32:~/../part4_inputs_outputs/lab1-2_adc$ ~/esp/esp-idf/components/
   esptool_py/esptool/espefuse.py --port /dev/ttyUSB0 adc_info
```

- We use the *ADC 1* with *channel 1* with a precision of 10 bits and we want the maximum range of voltage. Complete the code of the *main.c* file
- Perform the wiring on the board (cf. potentiometer documentation in the `Documentation/potentiometer_10K_datasheet.pdf`).
- Build and run the program. What is the time conversion ?
- Vary the potentiometer and write the min and max voltage value.
- Change the attenuation to *ADC_ATTEN_DB_2_5*. What do you remark when you vary the potentiometer ? Why ?

LAB N° 2

Pulse Width Modulation (PWM)

Lab Objectives

- Understand the principle of Pulse Width Modulation (PWM).

2.1 Preparation

Help you of the PWM documentation (PWM Web help) to answer the following questions :

- How many PWM channels can be used ? Why the channels are they divided into two groups ?

- What is the fade ?

2.2 PWM application with a LED (Lab2)

In order to learn how to use a PWM, we will configure the PWM with a GPIO pin as depicted in the figure 2.1. The objective will be to gradually turn on or turn off the LED.

1. In the « part4_inputs_outputs » folder, create the « lab2_pwm » lab from « esp32-vscode-project-template » GitHub repository.
2. Overwrite the « main.c » file by the provided code of the « lab2_pwm/main.c » file.
3. Copy the provided « lab2_pwm/sdkconfig.defaults » file to the project folder.

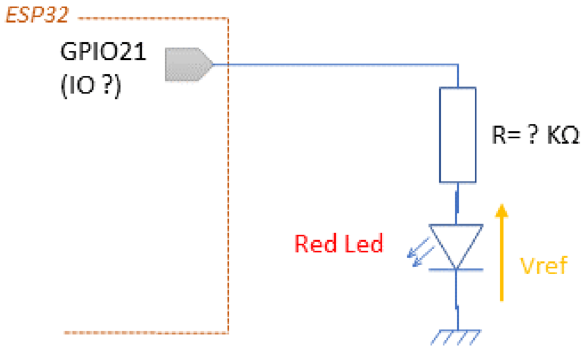


Figure 2.1 – *PWM schematic with a LED.*

4. Complete the code for the PWM configuration describe in the main file as commentary. Help you of the *ledc_timer_config_t* and the *ledc_channel_config_t* types of from Visual Studio Code and from the PWM Web help 1 and PWM Web help 2.
5. We want a current of around $5mA$. The $V_{ref} \approx 1.9V$ for the Led and the maximum average output voltage of the GPIO is $\approx 3.3V$, What is the nearest value of the standardized resistor to use ?

6. Perform the wiring on the board.
7. Build and run the program.
8. Explain the scenario 1. Why do we use the value 1024 ?

9. Explain the scenario 2.

LAB N° 3

Universal Asynchronous Receiver-Transmitter (UART) communication

Lab Objectives

- Using UART to transfer data.

3.1 Preparation

Help you of the DAC documentation (UART Web help) to answer the following questions :

- How many UART can be used ? What is the correspondence with the TXD/RXD pins of the GPIO for UART 2 ?
- What is the role of the CTS and RTS pins ?
- Is it mandatory to connect the CTS and RTS pins ? Justify.

3.2 Echo application (Lab3, part 1)

In order to learn how to use an UART, we will connect the transmit and receive pins to form a loop and therefore an echo as depicted in the figure 3.1. We can also see the UART configuration on the figure.

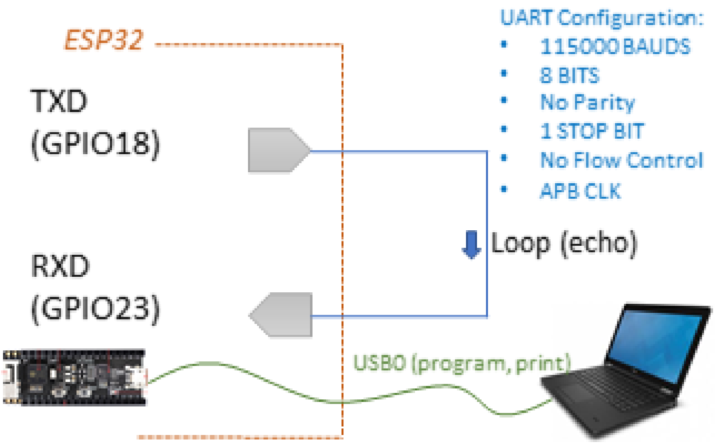


Figure 3.1 – *Echo application.*

1. In the « part4_inputs_outputs » folder, create the « lab3_uart » lab from « esp32-vscode-project-template » GitHub repository.
2. Overwrite the « main.c » file by the provided code of the « lab3_uart/main.c » file.
3. Copy the provided « lab3_uart/sdkconfig.defaults » file to the project folder.
4. Complete the code for the UART configuration depicted in the figure 3.1. Help you of the *uart_config_t* type of from Visual Studio Code and from the UART Web help
5. Connect the RxD and TxD pins on the board.
6. Build and run the program.

3.3 Transmission to a terminal of the computer (Lab3, part 2)

When we want to retrieve data from sensors, it is necessary to transmit them to a computer which will store the data. One solution consists in using the serial link previously studied as depicted in the figure 3.2. We will keep the previous wiring of the board using in the Lab3, part 1.

1. Connect the *esp-prog-jtag* board. Only TXD (yellow wire) and GND must be connected. The RXD wire (green wire) is disconnected. Be careful, we use the **Program Interface**, not the JTAG Interface, ESP PROG JTAG Board Web help.
2. Connect the USB cable. We use the */dev/ttyUSB2* port. Check if it exists with *ls* command.
3. Open a new Terminal and run the command below. We use *minicom* program to receive data.

```
esp32:~/../part4_inputs_outputs/lab3_uart$ minicom -D /dev/ttyUSB2
```

4. Build and run the program. What do you see in the minicom terminal ?

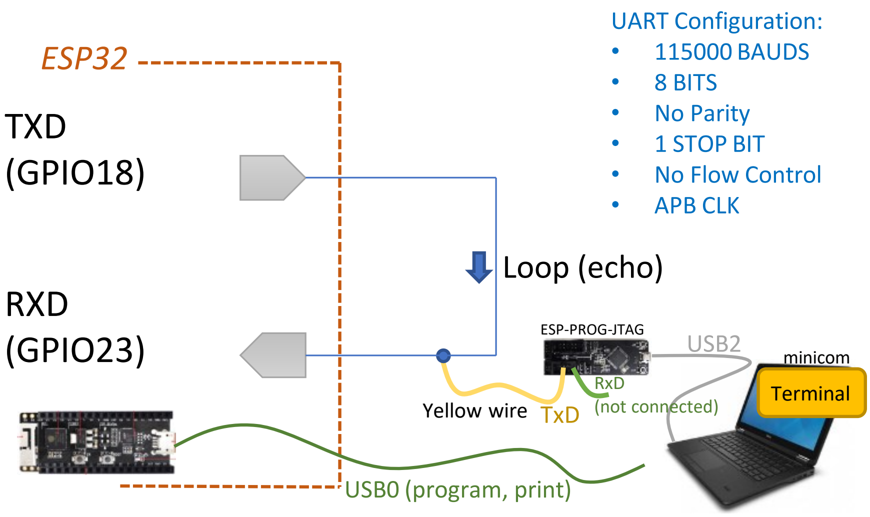


FIGURE 3.2 – *Transmission to a terminal of the computer.*

LAB N° 4

Application with Human-machine interface

Lab Objectives

- Put into practice the knowledge of previous labs.
- Human-machine interface (HMI) with Node-RED.

4.1 Installation of Node-RED

To install Node-RED, follow the items. For more information, you can help you of the Node-RED documentation (Node-RED Web help).

- To install prerequisites for Node-RED, run the commands below :

```
esp32:~$ sudo apt update
esp32:~$ sudo apt -y install curl dirmngr apt-transport-https lsb-release ca-
   certificates
esp32:~$ sudo apt -y install nodejs
esp32:~$ sudo apt install npm
esp32:~$ sudo npm cache clean -f
esp32:~$ sudo npm install -g n
esp32:~$ sudo n stable
```

- To install Node-RED, run the command below :

```
esp32:~$ sudo npm install -g --unsafe-perm node-red


/usr/local/bin/node-red -> /usr/local/lib/node_modules/node-red/red.js
/usr/local/bin/node-red-pi -> /usr/local/lib/node_modules/node-red/bin/node-
   red-pi

```

```
> bcrypt@5.0.1 install /usr/local/lib/node_modules/node-red/node_modules/
    bcrypt
> node-pre-gyp install --fallback-to-build

[bcrypt] Success: "/usr/local/lib/node_modules/node-red/node_modules/bcrypt/
    lib/binding/napi-v3/bcrypt_lib.node" is installed via remote
+ node-red@2.1.3
added 290 packages from 372 contributors in 22.001s
```

- Run node-RED.

```
esp32:~$ node-red

5 Nov 11:34:47 - [info]

Welcome to Node-RED
===================

5 Nov 11:34:47 - [info] Node-RED version: v2.1.3
5 Nov 11:34:47 - [info] Node.js  version: v12.18.4
5 Nov 11:34:47 - [info] Linux 5.4.0-54-generic x64 LE
5 Nov 11:34:47 - [info] Loading palette nodes
5 Nov 11:34:47 - [info] Settings file  : /home/lab/.node-red/settings.js
5 Nov 11:34:47 - [info] Context store  : 'default' [module=memory]
5 Nov 11:34:47 - [info] User directory : /home/lab/.node-red
5 Nov 11:34:47 - [warn] Projects disabled : editorTheme.projects.enabled=
    false
5 Nov 11:34:47 - [info] Flows file     : /home/lab/.node-red/flows.json
5 Nov 11:34:47 - [info] Creating new flow file
5 Nov 11:34:47 - [warn]

---------------------------------------------------------------------
Your flow credentials file is encrypted using a system-generated key.

If the system-generated key is lost for any reason, your credentials
file will not be recoverable, you will have to delete it and re-enter
your credentials.

You should set your own key using the 'credentialSecret' option in
your settings file. Node-RED will then re-encrypt your credentials
file using your chosen key the next time you deploy a change.
---------------------------------------------------------------------

5 Nov 11:34:47 - [info] Server now running at http://127.0.0.1:1880/
5 Nov 11:34:47 - [info] Starting flows
5 Nov 11:34:47 - [info] Started flows
```

- Open a navigator and use the URL http://127.0.0.1:1880/ for the programming part.

- Install Node *node-red-node-serialport*, *node-red-contrib-chartjs* and *node-red-dashboard*.
  — Go to *Manage palette* Menu.
- For the HMI part, use the URL `http://127.0.0.1:1880/ui`

4.2 Specification of the application (Lab4)

The application consists of detecting light in a room every second, then sending the value in mV to a human-machine interface (HMI) managed with Node-RED as depicted in the figure 4.1. The TXD output format just sends the integer voltage of the GL5506 photoresistance in mV. The format of the RXD output is a string with the color name and the duty cycle value (integer range 0 to 1024). Look at the documentation of the GL5506 photoresistance.

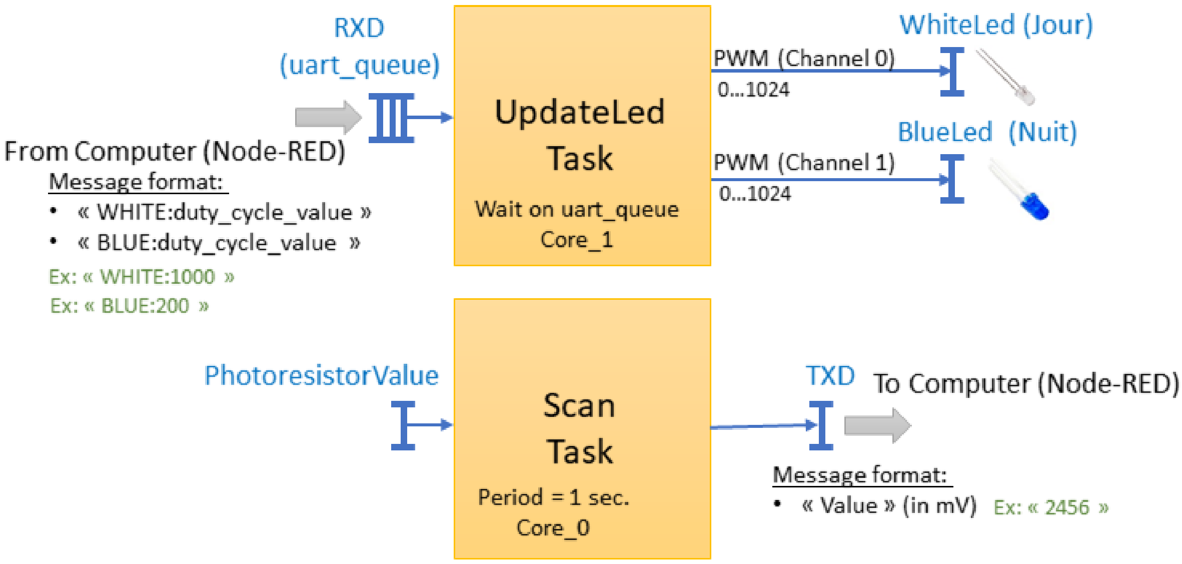


FIGURE 4.1 – *Specification of the application.*

The hardware specification is also depicted in the figure 4.2.

1. In the « part4_inputs_outputs » folder, create the « lab4_app_photoresistor » lab from « esp32-vscode-project-template » GitHub repository.
2. Overwrite the « main.c » file by the provided code of the « lab4_app_photoresistor/main.c » file.
3. Copy the provided « lab4_app_photoresistor/sdkconfig.defaults » file to the project folder.
4. Also add the « my_helper_fct.h » used in FreeRTOS labs. You have the DISPLAY() macro.
5. Complete the code taking into account the comments. Do not forget to add *DISPLAY()* macros for the debug.

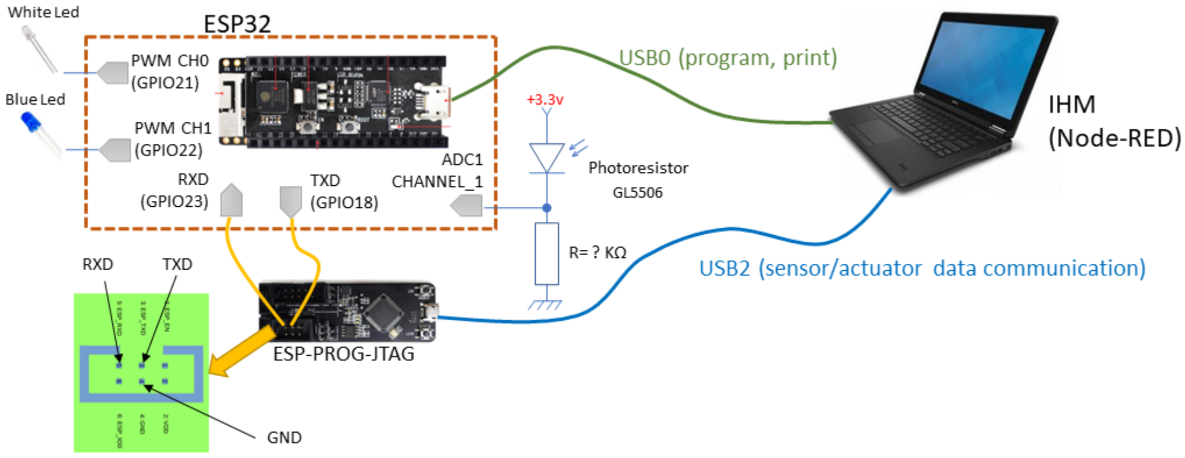


FIGURE 4.2 – *Hardware specification.*

6. Write and create the *ScanTask* task that reads the voltage and send to the UART every second. The message format is a string. For example, the message is "1254\n" for an integer value equals to 1254 mv. Use *sprintf()* to convert the interger value to a string.
7. Wiring the board as depicted in the figure 4.2.
8. Build and run the program with the console in the computer as a previous lab.
9. Import the provided « nodered-v1.json » file to Node-RED (Top Right Menu : Import and select the file) as depicted in the figure 4.3. This file is the « low-code » program of Node-RED. Look at the contents of the *Compute* function. The code is written in JavaScript.
10. Deploy the Node-RED program (*Deploy* button at top right) and go to the HMI (`http://127.0.0.1:1880/ui`).
11. Build and run the ESP32 program.

We want to add a feature in the Node-RED program that allows controlling the LEDs depending on the input voltage value. We keep the manual control of the LEDs.

1. Add a *Control* function to the Node-RED program between the *serial in* node and the *serial out* node as depicted in the figure 4.4. Add the JavaScript code in the *Control* function.

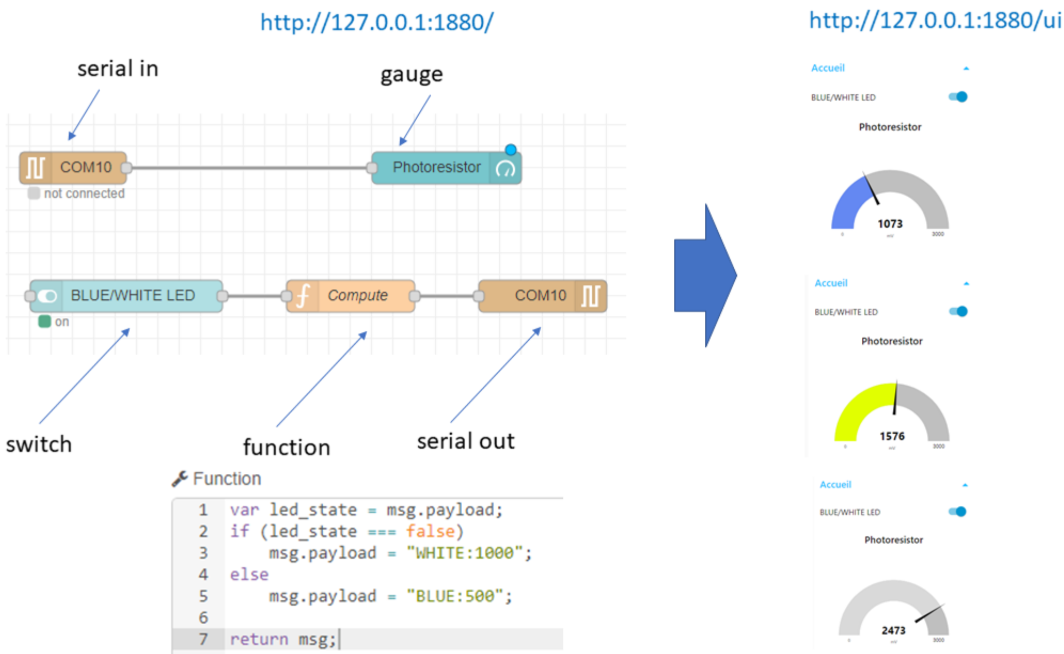


FIGURE 4.3 – *Node-RED application.*

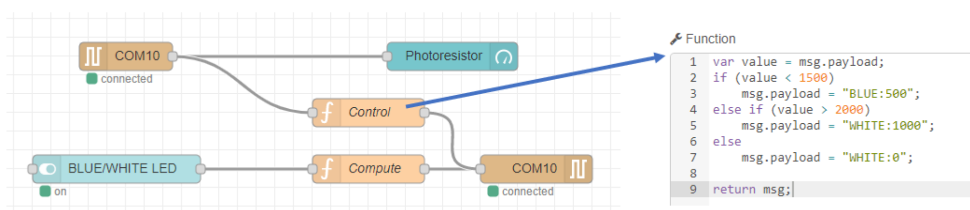


FIGURE 4.4 – *Node-RED application with Control node.*

Part III

Inter-Integrated Circuit Communication (I2C)

LAB N° 1

Inter-Integrated Circuit Communication (I2C)

Lab Objectives

- Study the I2C protocol.
- Implement a temperature sensor (LM75A) with an I2C communication.

1.1 Preparation

Help you of the provided LM75A temperature sensor documentation to answer the following questions :

- What is the 2 signals for an I2C communication ?
- How many bits is the LM75A slave address coded in ? What is the slave address of the LM75A sensor ? Is it possible to modify the slave address ?
- What do the *Thyst* and *Tos* register represent ?
- What is the role of the *OS* output ?

1.2 Using I2C provided tools (Lab1)

The goal is to implement of the temperature sensor with the I2C communication. Before learning to program the I2C communications transactions by functions, we will start by testing the communication of the I2C sensor with tools that allow direct access to the registers of the sensor. For this, we will rely on an example provided by *esp-idf*.

1. Create a new folder named « part5_i2c_com » and copy the *i2c_tools esp-idf* project as below :

```
esp32:~$ cd ~/esp/labs
esp32:~$ mkdir part5_i2c_com
esp32:~$ cp -R ~/esp/esp-idf/examples/peripherals/i2c/i2c_tools ~/esp/labs/
    part5_i2c_com/lab1_i2c_tools
esp32:~$ cd ~/esp/labs/part5_i2c_com/lab1_i2c_tools
```

2. Copy a « .vscode » folder of a previous project to allow configuring Visual Studio Code. The « .vscode » folder is located a root project as « main » folder.
3. Open Visual Code Studio. The documentation of this project is in *README.md* file.
4. Wiring the sensor as shown in the figure 1.1.

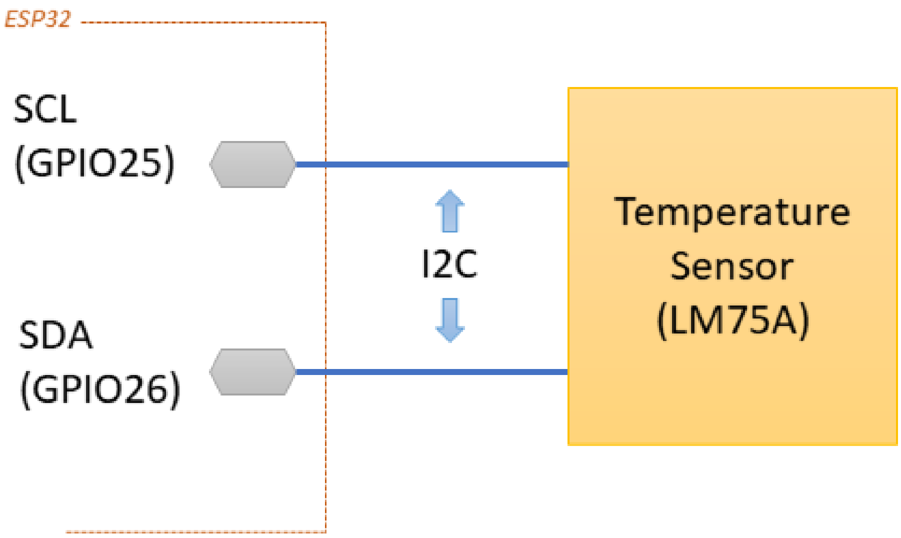


FIGURE 1.1 – *Temperature sensor connections.*

5. Build and run the program. You can see a console to perform commands. Execute each step below and explain it.

6. Explain the step 1.
```
i2cconfig --sda=26 --scl=25
```

7. Explain what it is displayed.
```
i2cdetect
```

8. Explain what it is displayed. From this data, compute the temperature in degree.
```
i2cget -c 0x48 -r 0 -l 2
```

Now, we can start to program the temperature sensor with C functions.

LAB N° 2

Programming I2C Communication

Lab Objectives

- Understand the I2C protocol and implement it with C functions.
- Program the temperature sensor (LM75A) with an I2C communication.
- Using interrupt with temperature sensor (LM75A).

2.1 Preparation

Help you of the provided LM75A temperature sensor documentation to answer the following questions :

- What is the address offset of the *Temp*, *Conf*, *Thyst* and *Tos* registers while using the LM75A datasheet ? We remind that the address of a register is the sum between the slave base address and the offset (called *pointer value* in the LM75A datasheet).

- What is the role of the *Thyst* and *Tos* registers ? What is the behavior of the *OS* output ?

2.2 Programming LM75A temperature sensor (Lab2-1)

The goal is to program the LM75A temperature sensor based on the I2C transactions provided by the LM75A documentation. We will be interested in the transaction depicted in the figure 2.1 (cf. documentation page 13, figure 10). This transaction will allow the temperature register to be read because the pointer is by default at zero.

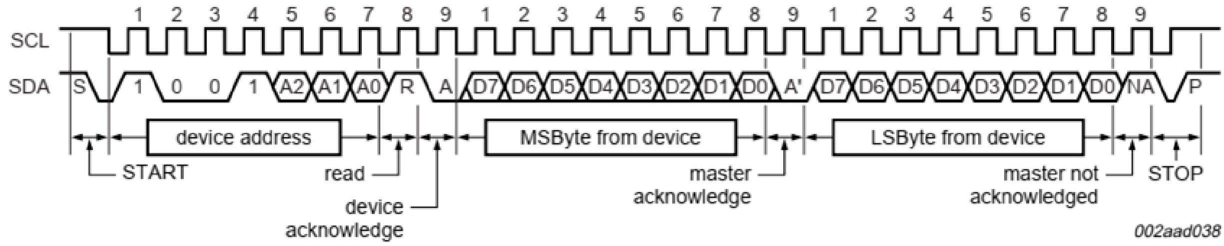
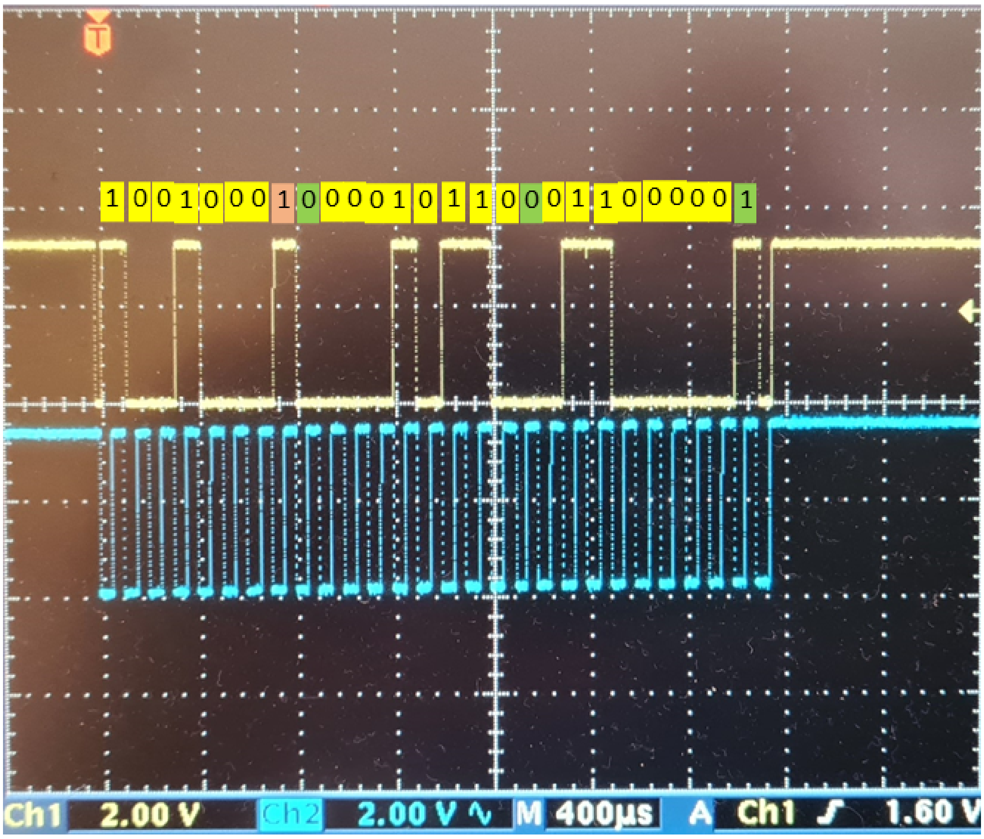


Figure 2.1 – *I2C transaction for reading registers with preset pointer (2-byte data).*

1. In the « part5_i2c_com » folder, create the « lab2-1_temp_sensor » lab from « esp32-vscode-project-template » GitHub repository.
2. Overwrite the « main.c » file by the provided code of the « lab2-1_temp_sensor/main.c » file.
3. Copy the « lab2-1_temp_sensor/LM75A.c » and « lab2-1_temp_sensor/LM75A.h » files to the *main* folder.
4. Copy the provided « lab2-1_temp_sensor/sdkconfig.defaults » file to the project folder.
5. Complete the *LM75A_ADDRESS* and *XXX_REG_OFFSET* macros (XXX = name of register) defined in « LM75A.h » file.
6. Complete the « main.c » file by helping you comments and the data structures whose documentation is directly accessible in Visual Studio Code (*F12* key or *right click* on mouse button, *Go to Definition* menu).
7. Build without execution. You should not have any errors.
8. Complete the *lm75a_readRegister()* function in « LM75A.c » file. Do not forget to manage error for each function. For example, the *i2c_master_start()* function returns *ESP_OK* constant if no error. In case of error, you should immediately exit the function while returning the error.
9. Complete in the « main.c » file the call of the *lm75a_readRegister()* function. Do not forget to check error and display the *raw* values in hexadecimal.
10. Build and run the program. Compute the temperature according to the raw value.

11. Captures a transaction of the SDA and SCL signals from the temperature reading using the oscilloscope. Connect the SDA (Channel 1) and SCL (Channel 2) of the oscilloscope. Wait on Channel 1, SINGLE SEQ. Analyze our transaction (slave address, ack, read, raws) as shown in the figure 2.2.



Slave address = 0x48 (100_1000)
Read = 1

raw[0](MSB) 0x16 (0001_0110)
raw[1](LSB) 0x60 (0110_0000)

Figure 2.2 – *I2C transaction for reading temperature.*

12. Complete the *convertRawToTemperature()* function to convert the raw value to temperature.

13. Complete in the « main.c » file the call of the *convertRawToTemperature()* function. Do not forget to check error and display the *raw* values in hexadecimal.

14. Build and run the program. The temperature must be displayed each 2 seconds.

2.3 Programming all registers of the temperature sensor (Lab2-2)

We now want to access the other registers of the sensor (TOS, THYST and Config registers). It is necessary to add new functions which will carry out the I2C transactions in order to access the registers. In order to test the functions, we will modify the THYST and TOS registers when the application starts. Likewise, we are going to create a periodic control task that will read the temperature every 2 seconds. The figure 2.3 depicts the application.

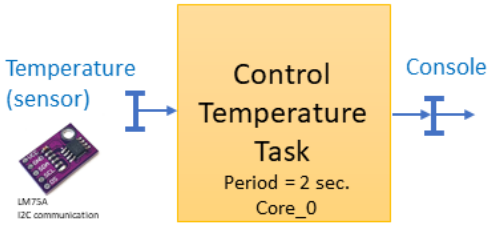


FIGURE 2.3 – *Specification of the LM75A application.*

1. In the « part5_i2c_com » folder, duplicate the « lab2-2_temp_sensor » lab and name it « lab2-2_temp_sensor_tune ». Do not forget to remove the *build* folder.
2. We add 3 functions for the register access : *lm75a_writeConfigRegister()*, *lm75a_readRegisterWithPointer()*, *lm75a_writeThysOrTosRegister()*. Add the provided code « add_LM75A.c » file at the end of the « LM75A.c ». Do not overwrite it !
3. Complete the *lm75a_writeConfigRegister()* function to write in configuration register by helping you figure with the Figure 2.4. Build the program.

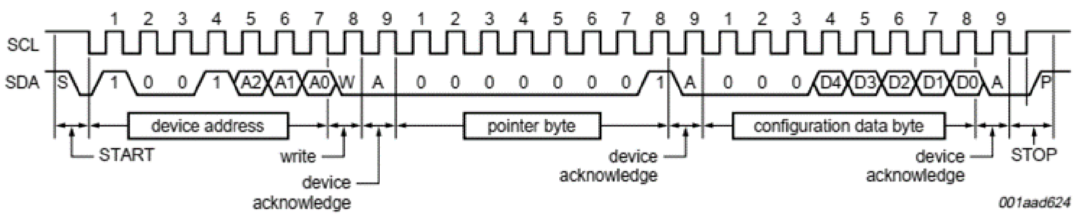


FIGURE 2.4 – *Write configuration register.*

4. Complete the *lm75a_readRegisterWithPointer()* function to read *Temp*, *Tos* or *Thyst* registers including pointer byte by helping you figure with the Figure 2.5. Build the program.
5. Complete the *lm75a_writeThysOrTosRegister()* function to write *Tos* or *Thyst* registers (2-byte data) by helping you figure with the Figure 2.6. Build the program.
6. Add the declarations of functions in the « LM75A.h ».

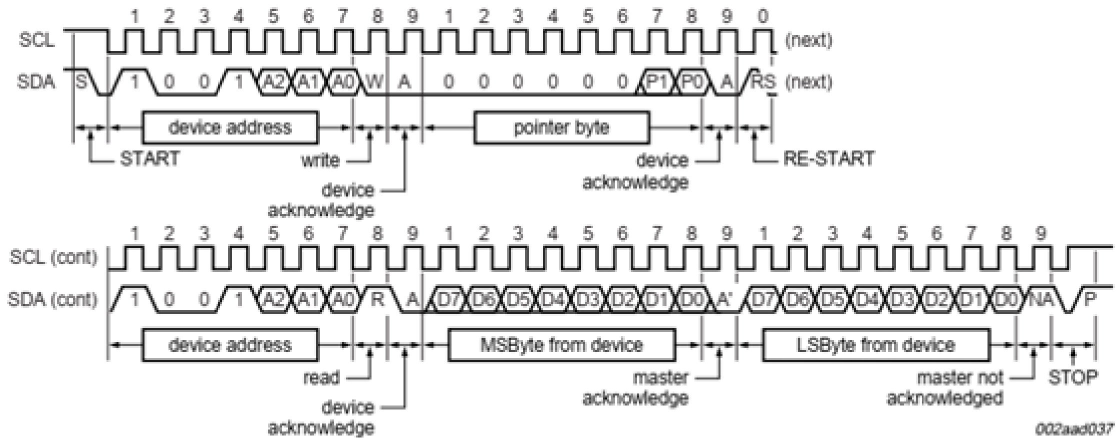


FIGURE 2.5 – *Read Temp, Tos or Thyst registers including pointer byte (2-byte data).*

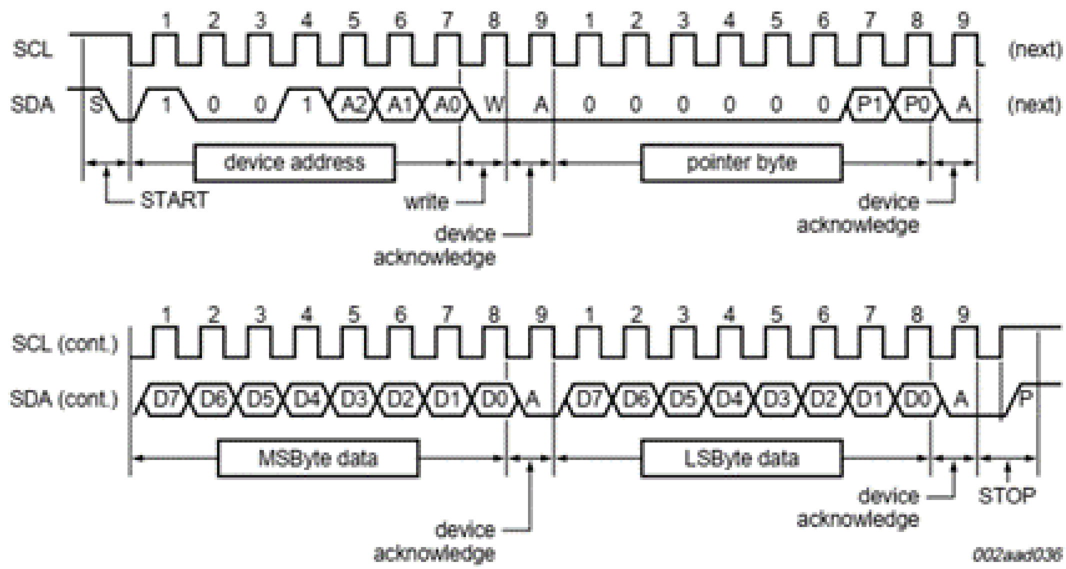


FIGURE 2.6 – *Write Tos or Thyst registers (2-byte data).*

7. What is the role of the *convertTemperatureToRaw()* function ?

8. From the provided « add_main.c » file, modify the current « main.c ». What do the code do in the *app_main()* ? How the *Tos* or *Thyst* registers are they initialize ?

   - Add declarations.
   - Add *vTaskControlTemperature* task.
   - Add and adapt the code in the *app_main()* from your previous *app_main()*.
   - Create the *vTaskControlTemperature* task at the end of the *app_main()*.

9. Complete the *vTaskControlTemperature()*.
10. Build and run the program. The temperature must be displayed each 2 seconds. What are the messages displayed ? What is the state of the red LED of the LM75A sensor and the *OS* output ? Note that the red LED is mounted in pull-up on the *OS* output. This means that when the *OS* output is high, the red LED is turn off.

11. Blow on the sensor to raise the temperature.What are the messages displayed and the behavior ?

12. Initialize the configuration register in comparator mode and *OS* output is active high. What is the behavior of the red LED while running program ?

2.4 Using interrupt with the temperature sensor (Lab2-3, Optional Work)

We want the task to be triggered when the OS output is active (low level, LED on), i.e. on the falling edge of the OS output. To do that, we use the *semTos* semaphore which wakes up the *Control Temperature* Task. If no interrupt is arise before 2 seconds (by using the timeout of the semaphore), the task displays the current temperature as you see in the given code of the *vTaskControlTemperature* task. Note that the *OS* output is reset to zero when a reading is made in one of the sensor registers, here when the task reads the temperature. The figure 2.7 depicts the application.

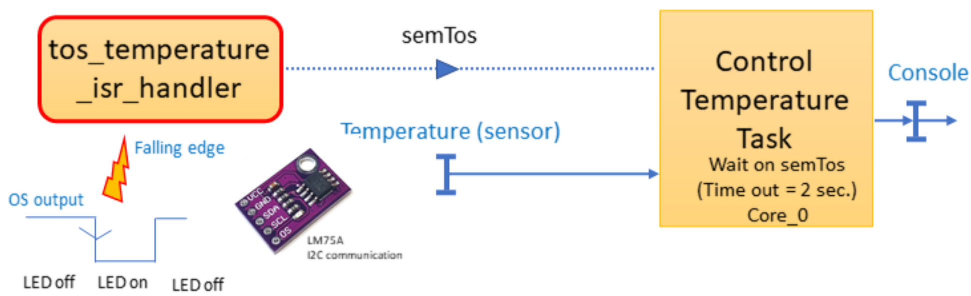


Figure 2.7 – *Specification of the LM75A application with interrupt.*

1. In the « part5_i2c_com » folder, duplicate the « lab2-2_temp_sensor » lab and name it « lab2-3_temp_sensor_intr ». Do not forget to remove the *build* folder.

2. Modify the « main.c » from the provided « add_main.c » file :
   - Declare (global declaration) and create (in the *app_main()*) the *xSemTos* semaphore.
   - Initialize the configuration register of the LM75A : Interrupt mode and *OS* output must be active low (in the *app_main()*).
   - Configure the *OS* input (in the *app_main()*).
   - Install the interrupt handler (in the *app_main()*).
   - Complete the *vTaskControlTemperature()* task.
   - Copy the *tos_temperature_isr_handler()* interrupt handler and complete the body to give the *xSemTos* semaphore (Help on the *xSemaphoreGiveFromISR()* function).
3. Build and run the program.
4. Connect the OS (Channel 1) and SDA (Channel 2) of the oscilloscope. Wait on Channel 1, SINGLE SEQ.
5. Blow on the sensor to raise the temperature until the interrupt is triggered. You must have the capture shown in the figure 2.8. Explain the behavior.

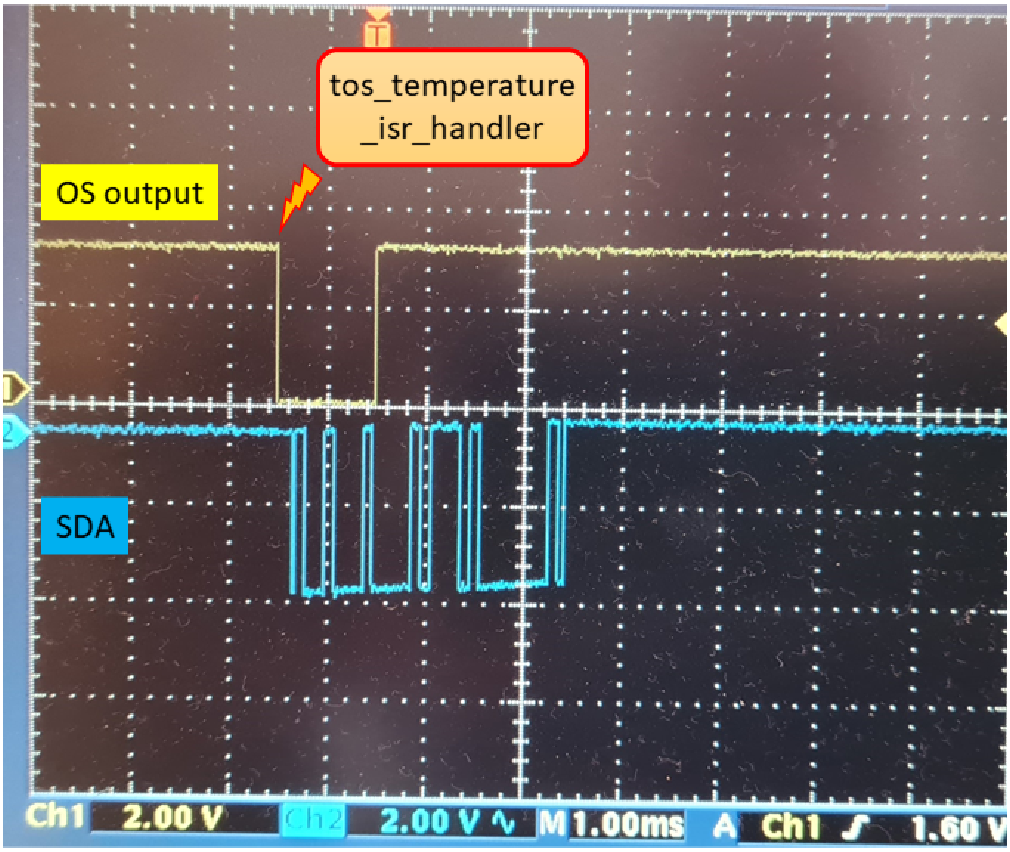


FIGURE 2.8 – *Capture of the SDA and OS output.*

Part IV

WiFi Networking

LAB N° 1

WiFi connection & HTTP protocol

Lab Objectives

- Implement a single WIFI connection to a Gateway or Home Box.
- Understand the WiFi software architecture.
- Fetch an HTML page from a HTTP request.
- Update the time/date of the ESP32 board from a time server.

1.1 WiFi connection (Lab1-1)

We want to connect our ESP32 board to access the internet, for example the Google site. To do this, we must connect our ESP32 board to an Access Point (AP) as we would do for a box at home. The Raspberry PI board will act as a box and also a Gateway. Indeed, the Raspberry PI is configured as a gateway between WIFI (Access Point mode) and the Ethernet connection connected to the Intranet and the Internet. However, it will be necessary to know the name of the Access Point and the password. The SSID (Service Set IDentifier) of the Raspberry PI board (Gateway) is *raspi-box-esp32*. The figure 1.1 summarizes the network architecture that we want to implement. It will also be possible to access our local computer if we know its IP address and if it is connected to the network.

Let's start by creating a new project.

1. Create a new folder named « part7_wifi ».
2. In the « part7_wifi » folder, create the « lab1-1_wifi_connection » lab from « esp32-vscode-project-template » GitHub repository.
3. Overwrite the « main.c » file by the provided code of the « lab1-1_wifi_connection/main.c » file.

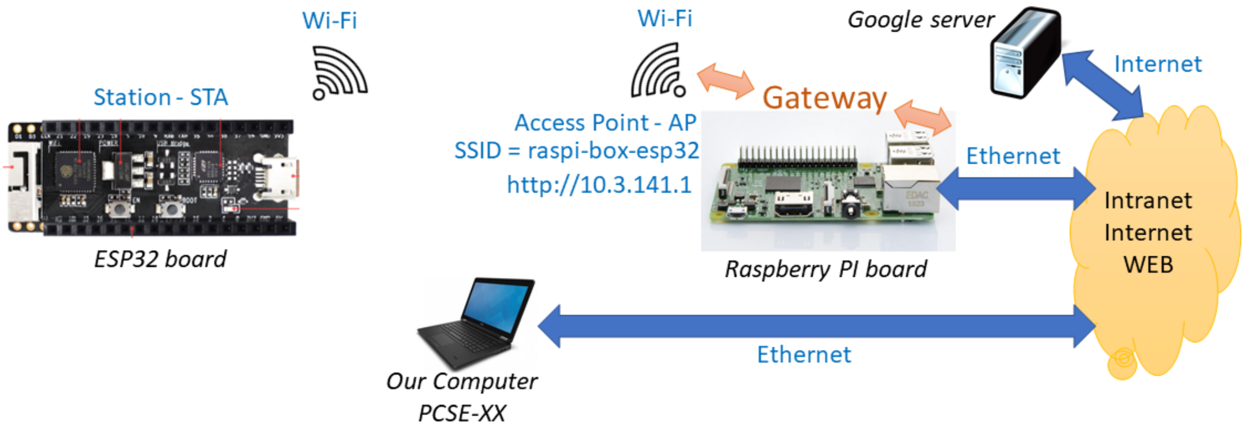


Figure 1.1 – *Network architecture.*

4. Copy the provided « wifi_connect.c », « wifi_connect.h » and « Kconfig.projbuild » files to the *main* folder.
5. Copy the provided « sdkconfig.defaults » file to the project root folder.

We will focus on the software architecture on the ESP32 in order to make a WiFi connection possible as depicted in the figure 1.2. We will write the *ConnectedWifi* task and the *app_main* task. The *connectionWifi* event corresponds to a semaphore which is declared in the « wifi_connect.c » file. In the « wifi_connect.h », you find the functions to start a WiFi connection and the SSID and the password macros which will be specified using the configuration file « sdkconfig.defaults ».

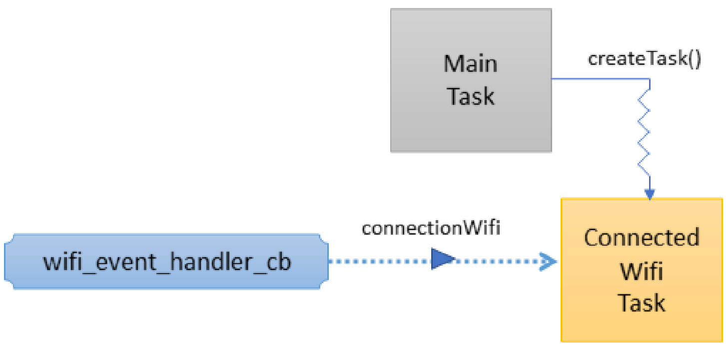


Figure 1.2 – *Software architecture for a WiFi connection.*

1. In the configuration file « sdkconfig.defaults », the SSID is right. The password will be given to you during the lab.
2. Build to check the compilation.
3. Study the *wifiInit()* function located in the « wifi_connect.c » file.
   (a) What do the *esp_event_handler_register()* function do ?

(b) What do the *esp_wifi_set_config()* function do ?

4. Write the *ConnectedWifi* task as described in the algorithm 1.1. Some recommendations :

   - An event is a binary semaphore (refer to the FreeRTOS documentation or Labs).
   - The *WaitOnTime()* function corresponds to the FreeRTOS *vTaskDelay()* function.
   - The *reboot()* function corresponds to the ESP32 *esp_restart()* function.

```
Task ConnectedWifi is
    Properties: Priority = 5, Stack Size = 3×1024
    In          : connectionWifi is event
    Cycle :
        WaitOnEvent (connectionWifi, timeout=10 seconds);
        if Not Timeout On connectionWifi Event then
            print("Connected on %s",SSID);
            print("Run Application");
            /* You can start your application here */
            WaitOnEvent (connectionWifi, timeout=INFINITE);
            print("Retried connection on %s",SSID);
        else
            print("Failed to connect. Retry in");
            for i :=1 to 5 do
                print("... %d",i);
                WaitOnTime (1 second);
            end
            reboot();
        end
    end
end
```

Algorithm 1.1: ConnectedWifi Task algorithm.

5. Complete the *app_main()* task.

6. Build and run the program. Extract the SSID (Service Set ID, it is the name of the Access Point), BSSID (Basic Service Set ID, ID = Access Point MAC Address), security mode, the STA IP address and the gateway IP address.

1.2 HTTP data (Lab1-2)

We want to retrieve data by sending HTTP requests. We will take as an example an access to the URL of the Google site : `http://www.google.com`. To do this, we will use an API (*esp_http_client*) for making HTTP/S requests from ESP-IDF programs as described in the ESP HTTP Client documentation.

1. In the « part7_wifi » folder, duplicate the « lab1-1_wifi_connection »lab and name it « lab1-2_wifi_http_data ». Do not forget to remove the *build* folder.
2. Add the provided code of the « lab1-2_wifi_http_data/add_main.c » file into the « main.c » file, just after the TAG constant declaration.
3. Build to check the compilation.
4. Study the *fetchHttpData()* function be helping you of the ESP HTTP Client documentation. What are the 3 main steps to execute an HTTP request ?
5. Study the *wifi_http_event_handler_cb()* function. What is the role of the function according to its body description ?
6. Add the call to the *fetchHttpData()* function in the right place in the « main.c » file. We will fetch the Google site : `http://www.google.com`.
7. Run the program.
8. Note the sequence of events. What is the request status ?
9. How many *HTTP_EVENT_ON_DATA* events are there and the length of the total data ?
   Modify the *wifi_http_event_handler_cb()* function to know the exact values automatically.

1.3 HTTP buffered data (Lab1-3)

We notice that the data arrives in packets. We do not store the data of the complete client request but rather display it as the packet data is received at each

HTTP_EVENT_ON_DATA event. The objective is to store all the data using a buffer that we are going to allocate dynamically. To do this, an *http_param_t* structure will contain the buffer and its size. It will be necessary to modify the *wifi_http_event_handler_cb* function for the *HTTP_EVENT_ON_DATA* and *HTTP_EVENT_ON_FINISH* events in order to save the data in the buffer.

1. In the « part7_wifi » folder, duplicate the « lab1-2_wifi_http_data » lab and name it « lab1-3_wifi_http_buffered_data ». Do not forget to remove the *build* folder.
2. Copy the provided « lab1-3_wifi_http_buffered_data/http_data.c » and « lab1-3_wifi_http_buffered_data/http_data.h » in the « main » folder.
3. In the « main.c » file, you currently have the *fetchHttpData()* and *wifi_http_event_handler_cb()* functions. Use the code of these functions to complete the body of the *fetchHttpData()* and *wifi_http_event_handler_cb()* functions in the « http_data.c ».
   At the end, you must remove the *fetchHttpData()* and *wifi_http_event_handler_cb()* functions which are in the « main.c » file.
   How is the buffer passed to the *wifi_http_event_handler_cb* function ?
4. Adapt the *connectedWifiTask* task to use the new *fetchHttpData()* function. Do not forget to free the memory (*free()* function) used by the structure after displaying the buffer. Attention, it is necessary to free the memory only if the buffer is not NULL.
5. Build, run the program and check the behavior.

We can now load a full HTML WEB page !

1.4 Update Time with the Network Time Protocol - NTP (Lab1-4, Optional Work)

Sometimes it is useful to date this sensor data. One solution is to use an NTP (Network Time Protocol) server. NTP is intended to synchronize date and time all participating systems to within a few milliseconds of Coordinated Universal Time (UTC). NTP is a standard Internet Protocol (IP) for synchronizing the computer clocks to some reference over a network. The current protocol is version 4 (NTPv4).
NTP uses a hierarchical architecture. Each level in the hierarchy is known as a **stratum**.

- *stratum 0* : high-precision timekeeping devices (GPS, atomic or radio clocks)
- *stratum 1* : servers have a direct connection to a stratum 0 and have the most accurate time.

The basic working principle for our ESP32 board is as follows :

1. The client device (ESP32 board) connects to the server using the *User Datagram Protocol* (UDP) on port 123.

2. The client transmits a request packet to a NTP server (interval time for ESP32 API)
3. The NTP server sends a time stamp packet. A time stamp packet contains information like UNIX timestamp, accuracy, delay or timezone.
4. The client parses out current date and time values.

We will use the previous lab because we need internet communication through the WiFi.

1. In the « part7_wifi » folder, duplicate the « lab1-3_wifi_http_data » lab and name it « lab1-4_wifi_ntp ». Do not forget to remove the *build* folder.
2. Copy the provided « lab1-4_wifi_ntp/ntp_time.c » and « lab1-4_wifi_ntp/ntp_time.h » in the « main » folder.
3. Explain directly in the code with comments the *initialize_sntp()* function. What is the role of the *sntp_set_time_sync_notification_cb()* ?

4. Call the *initialize_sntp()* function in the right place in the « main.c » file. We have 4 arguments :
   (a) *tz* : What is the Time Zone using CET and CEST format for the France country ? For example, it is *CET-8* for China. Web Help.

   (b) *interval* : We want 15 seconds of the time syncho interval.
   (c) *server* : The server name is `pool.ntp.org`.
   (d) *callback* : Declare a callback function named *time_synchro_cb()*. The callback example is described below.

```
void time_synchro_cb(struct timeval *tv) {
  printf("time at callback: %ld secs\n", tv->tv_sec);
  struct tm *timeinfo = localtime(&tv->tv_sec);
  printf("time at callback: %s", asctime(timeinfo));
}
```

5. Below the call of the *initialize_sntp()* function, print the current date/time using the function provided *getCurrentTimeToString()* in the « ntp_time.c » file.
6. Build, run the program. What is the problem about the date/time prints with the *getCurrentTimeToString()* ?

7. To solve the date/time update, we propose to add a function *initialize_sntp_and_wait()* that waits on a *timeSynchro* event. This *timeSynchro* event is sent by the *updated_time_synchro_cb()* callback as we use *time_synchro_cb()*

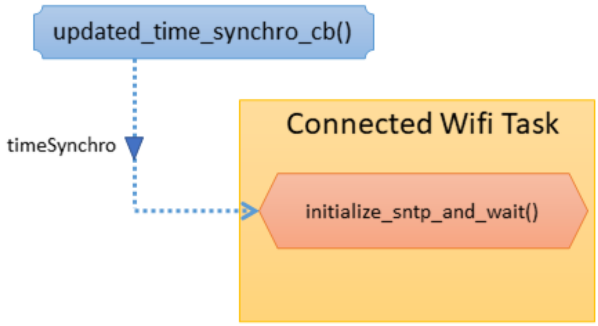


FIGURE 1.3 – *Software architecture for a WiFi connection.*

callback previously. The figure 1.3 depicts the principle.

To do this, we must add function and callback in the « ntp_time.c » file to use it as a API.

- Declare a local *timeSynchroSem* semaphore corresponding to the *timeSynchro* event.
- In the *updated_time_synchro_cb()* callback function, we will call the user callback as *time_synchro_cb()* callback. Therefore, we declare locally a *user_time_synchro_cb* variable. The type is *sntp_sync_time_cb_t* that represents the pointer of the user callback function.
- Add the *initialize_sntp_and_wait()* function. The prototype is the same as the *initialize_sntp()* function except the return value that is *bool* type. In this function, we just create a semaphore, call the *initialize_sntp()* function and wait on the *timeSynchroSem* semaphore (timeout = 30 seconds). The function returns the value of the waiting function of the *timeSynchroSem* semaphore, pdTRUE is no error, pdFALSE if the timeout is trigged.
- Add the *updated_time_synchro_cb()* callback function. This callback just gives the *timeSynchroSem* semaphore and if the *user_time_synchro_cb* pointer is not NULL, call user callback through the *user_time_synchro_cb* pointer.
- Do not forget to declare the prototype function in the « ntp_time.h » file.

8. Replace the *initialize_sntp()* function to *initialize_sntp_and_wait()* in the *connected Wifi* task.
9. Build, run the program. Check that the date/time prints correctly with the *getCurrentTimeToString()*.
10. Find another NTP server on the web and test it.

We can now have the date and time in the ESP32 board !

LAB N° 2

REST API for getting sensor information

Lab Objectives

- Understand the usage of the REST API.
- Implement GET HTTP methods through a WEB REST API (http://openweathermap.org) for getting sensor information.

2.1 REST API for getting weather report information (Lab2)

A REST API (REpresentational State Transfer) is an Application Programming Interface (API or web API) that respects the constraints of the REST architecture style and makes it possible to interact with RESTful web services. In a REST web service, requests made to the URI of a resource produce a response whose body is formatted in HTML, XML, JSON, or some other format. An URI (Uniform Resource Identifier) is a string containing characters that identify a physical or logical resource, like a page, or book, or a document. Not to be confused with URL (Uniform Resource Locator) that is special type of URI that also tells you how to access it, such as HTTP, HTTPs, FTP. In fact, URL is a subset of URI that specifies where a resource exists and the mechanism for retrieving it, while URI is a superset of URL that identifies a resource. The syntax of an URL is :

```
http://www.domainname.com:port_number/folder_name/filename.html
```

We can divide the above URL into the following parts :

- *Protocol* : It is the first part of the URL, here HTTP.
- *www.domainname.com* : It is your domain name. It is also known as server id or the host.

- *port_number* : Determine the port the server uses for connections. For HTTP, it is 80. See reference to the list of TCP and UDP port numbers.
- *folder_name* : It indicates that the website page referenced in filed in a given folder on the webserver
- *filename.html* : It is the web page filename in HTML format according to the extension file.

In our case, we use JSON format for data or resource. When the HTTP protocol is used, as is often the case and will be our case, the available HTTP methods are GET, HEAD, POST, PUT, PATCH, DELETE, CONNECT, OPTIONS and TRACE. For now, we will only use the GET methods through HTTP protocol.
In our lab, we will be using the Current Weather Data web API. This web API accesses current weather data for any location on Earth including over 200,000 cities. A free account allows you to make 60 requests per minute, which will be enough for our Lab. However, it is necessary to obtain a key which will be that of my account.

2.1.1 How to use the REST API ?

Before starting to program, we will test a request that we will implement in ESP32. In the documentation you will find the syntax. The key used is **bfaf90865d45e39c390da17ffa61e195**. Take the example of the weather in Cannes. I the WEB navigator, test the syntax below (replace KEY by the correct value) :

```
http://api.openweathermap.org/data/2.5/weather?q=Cannes&appid=KEY
```

The web API response is a JSON format :


```
{"coord":{"lon":7.0167,"lat":43.55},"weather":[{"id":804,"main":"Clouds","
    description":"overcast clouds","icon":"04d"}],"base":"stations","main":{"temp"
    :286.88,"feels_like":286.61,"temp_min":284.1,"temp_max":288.98,"pressure":1017,
    "humidity":88},"visibility":10000,"wind":{"speed":1.03,"deg":0},"clouds":{"all"
    :90},"dt":1637141218,"sys":{"type":1,"id":6507,"country":"FR","sunrise"
    :1637130509,"sunset":1637165137},"timezone":3600,"id":6446684,"name":"Cannes","
    cod":200}
```


Another solution is to use *curl* command instead of the WEB navigator. In a terminal, run the command below (replace KEY by the correct value). The response is the same as previously.

```
curl "http://api.openweathermap.org/data/2.5/weather?q=Cannes&appid=KEY"
```

We notice that the default units are in kelvin instead of in degree. Modify the request to use degree.

The web API response is in degree as you see below.


```
{"coord":{"lon":7.0167,"lat":43.55},"weather":[{"id":804,"main":"Clouds","
    description":"overcast clouds","icon":"04d"}],"base":"stations","main":{"temp"
```

```
:13.73,"feels_like":13.46,"temp_min":10.95,"temp_max":17.01,"pressure":1017,"
humidity":88},"visibility":10000,"wind":{"speed":1.03,"deg":0},"clouds":{"all"
:90},"dt":1637141396,"sys":{"type":1,"id":6507,"country":"FR","sunrise"
:1637130509,"sunset":1637165137},"timezone":3600,"id":6446684,"name":"Cannes","
cod":200}
```


2.1.2 Using with ESP32

We previously learned how to request `http:\www.google.com` from an ESP32. We will therefore be able to take a previous lab and adapt it for our REST API requests.

1. In the « part7_wifi » folder, duplicate the « lab1-3_wifi_http_buffered_data » lab and name it « lab2_wifi_rest_api ». Do not forget to remove the *build* folder.
2. If you have done the optional « lab1-4_wifi_ntp » lab, copy the « ntp_time.c » and « ntp_time.h » in the « main » folder.
3. Correct the code « main.c » file to request the weather in Cannes.
4. Build, run the program and check the JSON response.

2.1.3 Parsing JSON format

First of all, it is important to study the JSON format that you will find on the JSON Web site. We use the cJSON API for the ESP32 labs.
Let's take an example where we want to extract the latitude and longitude from the JSON response we got earlier. We have a *coord* object which includes 2 values of type *object*. These objects correspond to the longitude (*lon* object) and latitude (*lat* object). These 2 objects are of type *double*. Let's write the example with cJSON API.

1. Adapt the « main.c » file by helping you of the provided « add_main.c » file. You have an example of the *extractJSONWeatherMapInformation()* function to extract the JSON information from the WeatherMapInformation response.
2. Build, run the program and check the weather map information, latitude and longitude.
3. Complete the *extractJSONWeatherMapInformation()* function, the *weathermapinfo_t* structure and *connectedWifi* task to extract and to print the temperature (temp, feels_like, temp_min, temp_max), the pressure and the humidity.
4. Build and run the program.
5. Complete again the « main.c » file to extract and to print the description field.
6. Build and run the program.

We now know how to request data through a REST API!

LAB N° 3

REST API for sending email (Optional Work)

Lab Objectives

- Implement POST HTTP methods through a WEB REST API (`http://api.mailjet.com`) for sending sensor information.

3.1 Introduction

In an IoT architecture, it is generally necessary to send alerts depending on the state of the sensors. Alerts can be email, SMS, notification to your phone, etc. We will focus on sending an alert by email. To do this, it is necessary to connect to a mail server. In our lab, we will use a Web REST API which will act as a relay with your professional or personal mailbox. In the signature of the email received, you will have the signature *« <your email address> (<your name> via bnc3.mailjet.com) »* for example. The Mailjet API is organized around web REST API. It has predictable, resource-oriented URLs, and uses HTTP response codes to indicate API errors. All request and response bodies are encoded in JSON, including errors.

3.2 Preparation (Lab3)

In order to be able to use the REST API, it is necessary to create an account by going to the mailjet web site web API, « Sign Up ». Then, create an main account with your sender professional or personal email address and generate the Master API Key as shown the figure 3.1. The API key will be used to identify us (user and password) when using the REST API.

We can now start studying an example to send an email with the *curl* command as below.

```
curl -s \
```

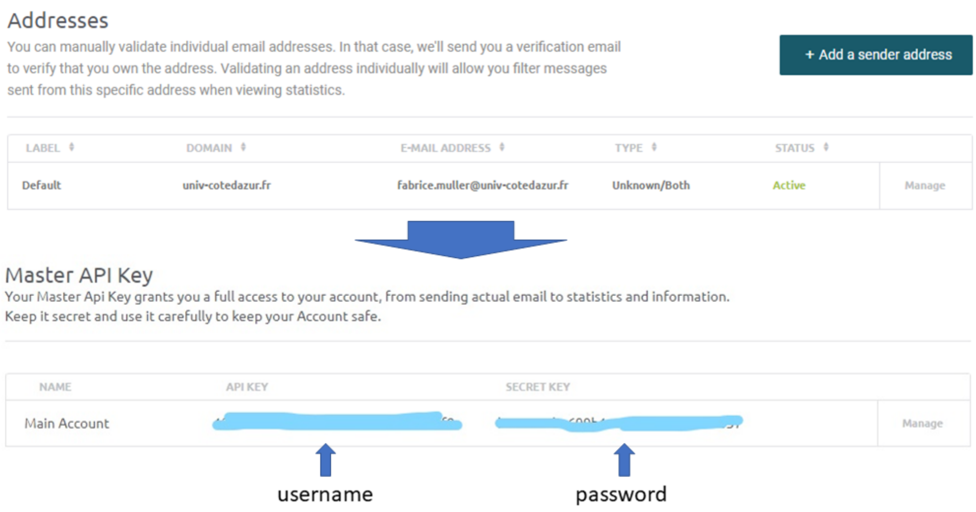


Figure 3.1 – *emailjet Account.*

```
-X POST \
--user "$MJ_APIKEY_PUBLIC:$MJ_APIKEY_PRIVATE" \
https://api.mailjet.com/v3.1/send \
-H 'Content-Type: application/json' \
-d '{
  "Messages":[
      {
          "From": {
              "Email": "$SENDER_EMAIL",
              "Name": "Me"
          },
          "To": [
              {
                  "Email": "$RECIPIENT_EMAIL",
                  "Name": "You"
              }
          ],
          "Subject": "My first Mailjet Email!",
          "TextPart": "Greetings from Mailjet!",
          "HTMLPart": "<h3>Dear passenger 1, welcome to <a href=\"https://www.
 mailjet.com/\">Mailjet</a>!</h3><br />May the delivery force be with you!"
      }
  ]
}'
```

Using the curl documentation, examples and the Send API v3.1 to answer the following questions.

1. What does *-X POST* mean ?

2. What does *–user "MJ_APIKEY_PUBLIC :MJ_APIKEY_PRIVATE"* mean ?

3. What does *https ://api.mailjet.com/v3.1/send* represent ?

4. What does *-H 'Content-Type : application/json'* mean ? What is a MIME (Multipurpose Internet Mail Extensions) type ?

5. What does *-d ...* command represent ?

6. Comment on the different fields (type of JSON object, role) of the message using the Send API v3.1.

3.3 Implementation of sending email from a REST API (Lab3, Optional Work)

In the « lab2_wifi_rest_api » lab, we collected weather data. We use the equivalent of the GET method of the REST API. In the case of sending an email by the POST method, other information is necessary : user, password, MIME type, etc. In the ESP HTTP Client interface, there are functions to initialize these fields.

1. In the « part7_wifi » folder, duplicate the « lab2_wifi_rest_api » lab and name it « lab3_wifi_post_rest_api ». Do not forget to remove the *build* folder.
2. If you have done the optional « lab1-4_wifi_ntp » lab, copy the « ntp_time.c » and « ntp_time.h » in the « main » folder.
3. Modify the « http_data.h » from the provided « add_http_data.h » file :
   - Declare *httpMethod_enum* and *header_t* types.
   - Add fields in the *http_param_t* structure. Each field corresponds to a parameter of the Send API v3.1.

4. Modify the « http_data.c » from the provided « add_http_data.c » file for the *fetchHttpData()* function by helping you comments.
5. Set HTTP parameter fields in the *connectedWifiTask()* task.
6. Build, run the program. An error occurs when sending the email from REST API. What does the error relate to ?

7. To solve the problem, we must change ESP-TLS configuration. In the *sdkconfig.defaults* file, add these 4 lines. Do not forget to delete the *sdkconfig* file to rebuild all the project.

```
# ESP-TLS
CONFIG_ESP_TLS_USING_MBEDTLS=y
CONFIG_ESP_TLS_INSECURE=y
CONFIG_ESP_TLS_SKIP_SERVER_CERT_VERIFY=y
```

8. Build, run the program and check the sent mail in your mailbox. Check the JSON response.

Part V

MQTT Protocol

LAB N° 1

MQTT services

Lab Objectives

- Install and run Mosquitto broker
- Understand the Publish/Subscribe services with Mosquitto client
- Using of the MQTT API of ESP-IDF for ESP32

1.1 Mosquitto broker

1.1.1 Install Mosquitto

You have to install Mosquitto broker by following the script lines below. For more information, you can help you of the Mosquitto documentation (Mosquitto Web help).

```
sudo apt-add-repository ppa:mosquitto-dev/mosquitto-ppa
sudo apt-get update
sudo apt-get install mosquitto
sudo apt-get install mosquitto-clients
sudo apt clean
```

1.1.2 Run Mosquitto broker

Perform in a Terminal each of these commands in order to understand the operating principle of launching the broker.

- Launch broker in verbose mode with default port (number : 1883).

```
mosquitto -p 1883
```

- Open a new terminal and check if Mosquitto is running on 1883 port.

```
netstat -nptl | grep 1883
```

- Stop broker with *CTRL+C* key command.
- In the terminal where you stopped the broker, launch again the broker in deamon mode with default port (number : 1883).

```
mosquitto -p 1883 -d
```

- To stop the broker, we fetch the PID and kill the PID process. If you have an error, it is that the Mosquitto broker does not exist.

```
sudo kill -9 `pgrep mosquitto`
```

- Check if Mosquitto broker has been stopped.

```
netstat -nptl | grep 1883
```

You can also run Mosquitto as a service.

- Start Mosquitto as a service

```
sudo service mosquitto start
```

- Stop the Moquitto service

```
sudo service mosquitto stop
```

1.2 Mosquitto Client (Lab1-1)

1.2.1 Reach the IP address of the broker

Firstly, you must launch the broker in a Terminal.

```
mosquitto -p 1883
```

The Mosquitto broker is running on the *localhost* (*127.0.0.1*). However, the computer also has an IP address other than *localhost*. Run the command below.

```
ifconfig | grep inet
```

The result is depicted below. You extract the IP address of the computer (*192.168.1.29*) and the localhost (*127.0.0.1*). The IP address of the *localhost* is always *127.0.0.1*. For you, the IP address is probably different of *192.168.1.29*.

```
inet 192.168.1.29  netmask 255.255.255.0  broadcast 192.168.1.255
inet6 fe80::2e54:665c:6aa0:568d  prefixlen 64  scopeid 0x20<link>
inet 127.0.0.1  netmask 255.0.0.0
inet6 ::1  prefixlen 128  scopeid 0x10<host>
```

So, you can independently reach the computer in these 3 ways :

```
ping localhost
ping 127.0.0.1
ping 192.168.1.29
```

You have to configure the Mosquitto broker in anonymous mode and do not filter IP address. To do this, add the 2 lines in the *mosquitto.conf* file located at */etc/mosquitto/mosquitto.conf*. Then, restart the Mosquitto broker.

```
bind_address 0.0.0.0
allow_anonymous true
```

In conclusion, we can reach the Mosquitto broker from the *localhost* (*127.0.0.1*) or *192.168.1.29* IP address.

1.2.2 Using Mosquitto Client

We will now understand the behavior of an MQTT broker. To illustrate the principle of operation, we will open one terminal for publication and another one for subscription.

For the subscription, we use the *mosquitto_sub* command. For example, we subscribe on *school/roomE110/temperature* topic.

```
mosquitto_sub -h localhost -t "school/roomE110/temperature"
```

To publish the temperature (22.5 degrees) of *school/roomE110/temperature* topic, we use the *mosquitto_pub* command.

```
mosquitto_pub -h localhost -t "school/roomE110/temperature" -m 22.5
```

For debugging, you can use the *-d* flag for subscription and publication. First, stop the subscription by *CTRL+C* key command.

For the subscription :

```
mosquitto_sub -h localhost -t "school/roomE110/temperature" -d
```

For the publication :

```
mosquitto_pub -h localhost -t "school/roomE110/temperature" -m 22.5 -d
```

You can also publish JSON payload.

```
mosquitto_pub -h localhost -t "school/roomE110/temperature" -m '{"cur":22.5,"min
   ":12.5,"max":42.5}' -d
```

1.2.3 Application - Simulation of End-Nodes with Mosquitto client

We want to collect the temperature in various rooms located in many buildings. The format of the topic is : *place/building/room/sensor* where

- *place* = {Polytech}
- *building* = {AREA1, AREA2}
- *room* = {E110, S133, S218}
- *sensor* = {temperature, humidity}

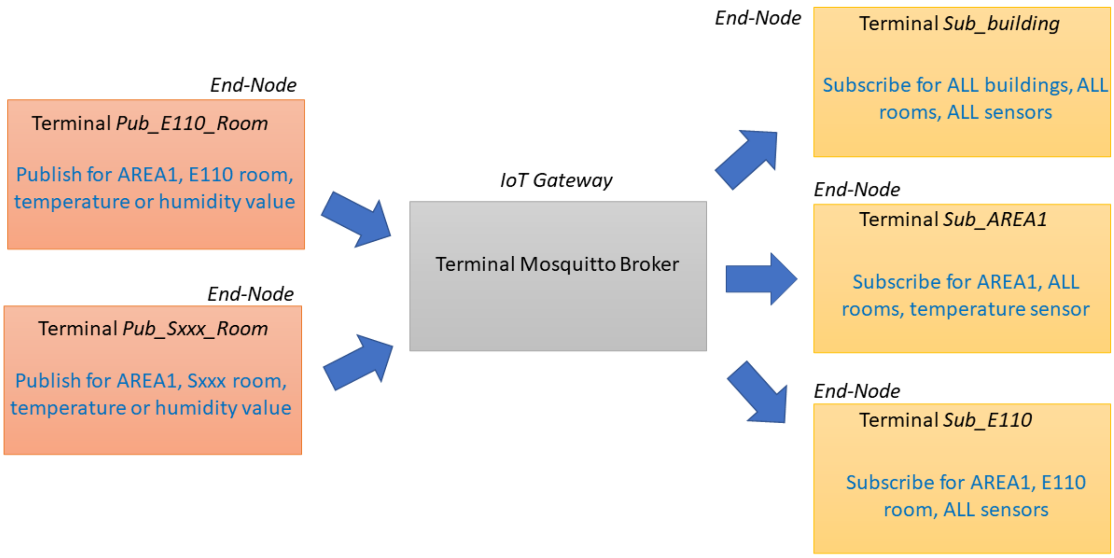


FIGURE 1.1 – *IoT Architecture for the exercise.*

The figure 1.1 summarizes the IoT architecture for the exercise.
To do this, we will open 6 terminals as shown in the figure 1.1.

1. Start the Mosquitto broker.
2. Write and run the command for the *Sub_building* Terminal.
3. Write and run the command for the *Sub_AREA1* Terminal.
4. Write and run the command for the *Sub_E110* Terminal.
5. Write and run the commands for the *Pub_E110_Room* Terminal. Temperature = 23.2 degrees and humidity = 46.8%. Interpret the behavior of the 3 subscription terminals.
6. Write and run the commands for the *Pub_Sxxx_Room* Terminal. Interpret the behavior of the 3 subscription terminals.
   - Room S113 : Temperature = 20.1 degrees and humidity = 42.1%
   - Room S218 : Temperature = 18.6 degrees and humidity = 53.4%

1.3 End-Node with ESP32 board (Lab1-2)

We are going to add an ESP32 End-Node which will come in addition to the terminals of the previous lab. This ESP32 End-Node will publish and subscribe to the previous topics of the Lab1-1. Do not close your 6 terminals!

1. Create the « part8_mqtt » folder.
2. In the « part8_mqtt » folder, copy the last WiFi project that runs correctly. Otherwise, use the first « lab1-1_wifi_connection » lab. Name it « lab1_mqtt_services ». Do not forget to remove the *build* folder.
3. Copy the provided « mqtt_tcp.c » and « mqtt_tcp.h » files to the *main* folder.
4. Study the *mqtt_start()* function located in the « mqtt_tcp.c » file.
   (a) What do the *esp_mqtt_client_init()* function do?

   (b) What do the *esp_mqtt_client_register_event()* function do?

5. What is the IP address of your local broker (help you of previous lab1-1)?

6. Adapt only the *connectedWifiTask()* task of the « main.c » file by helping you of the provided « add_main.c » file. Declare but do not complete the *testMqttTask()* task.
7. Build and run the program to test the connection to the broker.
8. Complete the *testMqttTask()* task for testing subscription and publication of topics as in the previous lab1-1.
9. Build and run the program. Look in the subscription terminals (*Sub_building*, ...) if the ESP32 End-Node correctly publishes the topics.
10. Publish topics from the *Pub_E110_Room* terminal and check the ESP32 End-Node subscription.

Part VI

Appendix

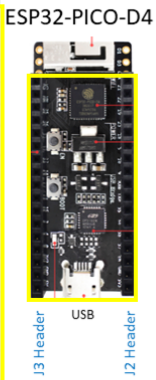
ANNEXE A

ESP32 Board

J3 Header

No.	Name	Type	Function
1	FLASH_CS (FCS)	I/O	GPIO16, HS1_DATA4 (See 1), U2RXD, EMAC_CLK_OUT
2	FLASH_SD0 (FSD0)	I/O	GPIO17, HS1_DATA5 (See 1), U2TXD, EMAC_CLK_OUT_180
3	FLASH_SD2 (FSD2)	I/O	GPIO11, SD_CMD, SPICS0, HS1_CMD (See 1), U1RTS
4	SENSOR_VP (FSVP)	I	GPIO36, ADC1_CH0, RTC_GPIO0
5	SENSOR_VN (FSVN)	I	GPIO39, ADC1_CH3, RTC_GPIO3
6	IO25	I/O	GPIO25, DAC_1, ADC2_CH8, RTC_GPIO6, EMAC_RXD0
7	IO26	I/O	GPIO26, DAC_2, ADC2_CH9, RTC_GPIO7, EMAC_RXD1
8	IO32	I/O	32K_XP (See 2a), ADC1_CH4, TOUCH9, RTC_GPIO9
9	IO33	I/O	32K_XN (See 2b), ADC1_CH5, TOUCH8, RTC_GPIO8
10	IO27	I/O	GPIO27, ADC2_CH7, TOUCH7, RTC_GPIO17 EMAC_RX_DV
11	IO14	I/O	ADC2_CH6, TOUCH6, RTC_GPIO16, MTMS, HSPICLK, HS2_CLK, SD_CLK, EMAC_TXD2
12	IO12	I/O	ADC2_CH5, TOUCH5, RTC_GPIO15, MTDI (See 4), HSPIQ, HS2_DATA2, SD_DATA2, EMAC_TXD3
13	IO13	I/O	ADC2_CH4, TOUCH4, RTC_GPIO14, MTCK, HSPID, HS2_DATA3, SD_DATA3, EMAC_RX_ER
14	IO15	I/O	ADC2_CH3, TOUCH3, RTC_GPIO13, MTDO, HSPICS0 HS2_CMD, SD_CMD, EMAC_RXD3
15	IO2	I/O	ADC2_CH2, TOUCH2, RTC_GPIO12, HSPIWP, HS2_DATA0, SD_DATA0
16	IO4	I/O	ADC2_CH0, TOUCH0, RTC_GPIO10, HSPIHD, HS2_DATA1, SD_DATA1, EMAC_TX_ER
17	IO0	I/O	ADC2_CH1, TOUCH1, RTC_GPIO11, CLK_OUT1 EMAC_TX_CLK
18	VDD33 (3V3)	P	3.3V power supply
19	GND	P	Ground
20	EXT_5V (5V)	P	5V power supply

3. This pin is connected to the pin of the USB bridge chip on the board.
4. The operating voltage of ESP32-PICO-KIT's embedded SPI flash is 3.3V. Therefore, the strapping pin MTDI should hold bit zero during the module power-on reset. If connected, please make sure that this pin is not held up on reset.



J2 Header

No.	Name	Type	Function
1	FLASH_SD1 (FSD1)	I/O	GPIO8, SD_DATA1, SPID, HS1_DATA1 (See 1), U2CTS
2	FLASH_SD3 (FSD3)	I/O	GPIO7, SD_DATA0, SPIQ, HS1_DATA0 (See 1), U2RTS
3	FLASH_CLK (FCLK)	I/O	GPIO6, SD_CLK, SPICLK, HS1_CLK (See 1), U1CTS
4	IO21	I/O	GPIO21, VSPIHD, EMAC_TX_EN
5	IO22	I/O	GPIO22, VSPIWP, U0RTS, EMAC_TXD1
6	IO19	I/O	GPIO19, VSPIQ, U0CTS, EMAC_TXD0
7	IO23	I/O	GPIO23, VSPID, HS1_STROBE
8	IO18	I/O	GPIO18, VSPICLK, HS1_DATA7
9	IO5	I/O	GPIO5, VSPICS0, HS1_DATA6, EMAC_RX_CLK
10	IO10	I/O	GPIO10, SD_DATA3, SPIWP, HS1_DATA3, U1TXD
11	IO9	I/O	GPIO9, SD_DATA2, SPIHD, HS1_DATA2, U1RXD
12	RXD0	I/O	GPIO3, U0RXD (See 3), CLK_OUT2
13	TXD0	I/O	GPIO1, U0TXD (See 3), CLK_OUT3, EMAC_RXD2
14	IO35	I	ADC1_CH7, RTC_GPIO5
15	IO34	I	ADC1_CH6, RTC_GPIO4
16	IO38	I	GPIO38, ADC1_CH2, RTC_GPIO2
17	IO37	I	GPIO37, ADC1_CH1, RTC_GPIO1
18	EN	I	CHIP_PU
19	GND	P	Ground
20	VDD33 (3V3)	P	3.3V power supply

1. This pin is connected to the flash pin of ESP32-PICO-D4.
2. 32.768 kHz crystal oscillator: a) input b) output

Figure A.1 – *Pin description of ESP32-PICO-D4 board.*

References