University Cote d'Azur

Polytech Nice Sophia

Department
Engineering of Electronic Systems

Labs
IoT Design

Fabrice MULLER
✉ : Fabrice.Muller@univ-cotedazur.fr

- 2021/2022 -

Contents

I IoT Framework **1**

1 Framework **3**
1.1 Espressif IoT Development Framework . . . 3
1.2 Visual Studio Code with ESP-IDF . . . 10

2 Debugging with Visual Studio Code **13**
2.1 Principle . . . 13
2.2 Wiring ESP-PROG JTAG board . . . 13
2.3 Create the Lab . . . 13
2.4 First run for Debug . . . 15
2.5 Debugging . . . 15
2.6 Debugging Walk through . . . 15

3 Working with C and IDF framework **19**
3.1 Useful functions (Lab3-1) . . . 19
3.2 Understanding the structures (Lab3-2) . . . 22
3.3 Refactoring code (Lab3-3) . . . 25
3.4 Dynamic allocation (Lab3-4) . . . 28
3.5 Linked list with pointers (Lab3-5) . . . 30

4 IDF Configuration and Components **33**
4.1 IDF Configurations (Lab4-1) . . . 33
4.2 IDF Components (Lab4-2) . . . 36

II IoT Architecture **39**

1 ESP32 Architecture **41**
1.1 Architecture (Lab1-1) . . . 41
1.2 System Time (Lab1-2) . . . 43

2 Memory **47**
2.1 Flash & SRAM Memories (Lab2-1) . . . 47
2.2 The sections . . . 50
2.3 Flash Memory - Custom partition table (Lab2-2) . . . 53
2.4 Storing files directly in flash memory (Lab2-3) . . . 56
2.5 Non Volatile Storage memory (Lab2-4) . . . 58
2.6 SPI Flash File System (Lab2-5) . . . 62

3 General Purpose Input/Output (GPIO) **67**
3.1 GPIO Output (Lab3-1) . . . 67
3.2 GPIO Input (Lab3-2) . . . 68
3.3 Advanced configuration for GPIO (Lab3-3) . . . 69
3.4 Exercise - Display state of the push button . . . 70

4 Interrupt with a General Purpose Input/Output (GPIO) **71**
4.1 Polling and Interrupt mode . . . 71
4.2 Polling mode (Lab4-1) . . . 71
4.3 Interrupt mode with GPIO (Lab4-2) . . . 72

III FreeRTOS **75**

1 Task & Scheduling **77**
1.1 Task scheduling on one core (Lab1-1) . . . 77
1.2 Task scheduling on two cores (Lab1-2) . . . 85
1.3 Approximated/Exactly periodic task (Lab1-3) . . . 89
1.4 Task handler and dynamic priority (Lab1-4, Optional work) . . . 90

2 Message Queue & Interrupt service **93**
2.1 Single Message Queue (Lab2-1) . . . 93
2.2 Message Queue with time out (Lab2-2) . . . 96
2.3 Blocking on single Queue (Lab2-3, Optional work) . . . 96
2.4 Using message queue with interrupts (Lab2-4) . . . 99

3 Semaphore & Mutex **101**
3.1 Specification of the application . . . 101
3.2 First Task synchronization (Lab3-1) . . . 104
3.3 Task synchronization with 2 semaphores (Lab3-2) . . . 106
3.4 Mutual Exclusion (Lab3-3) . . . 112

4 Direct task notification **119**
4.1 Direct task notification (Lab4-1) . . . 119
4.2 Direct task notification with a event value (Lab4-2) . . . 121

5 Application **123**
5.1 Specification of the application 123
5.2 Implementation of the application (Lab5) 126

IV Appendix **129**

A ESP32 Board **131**

References **133**

Part I

IoT 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

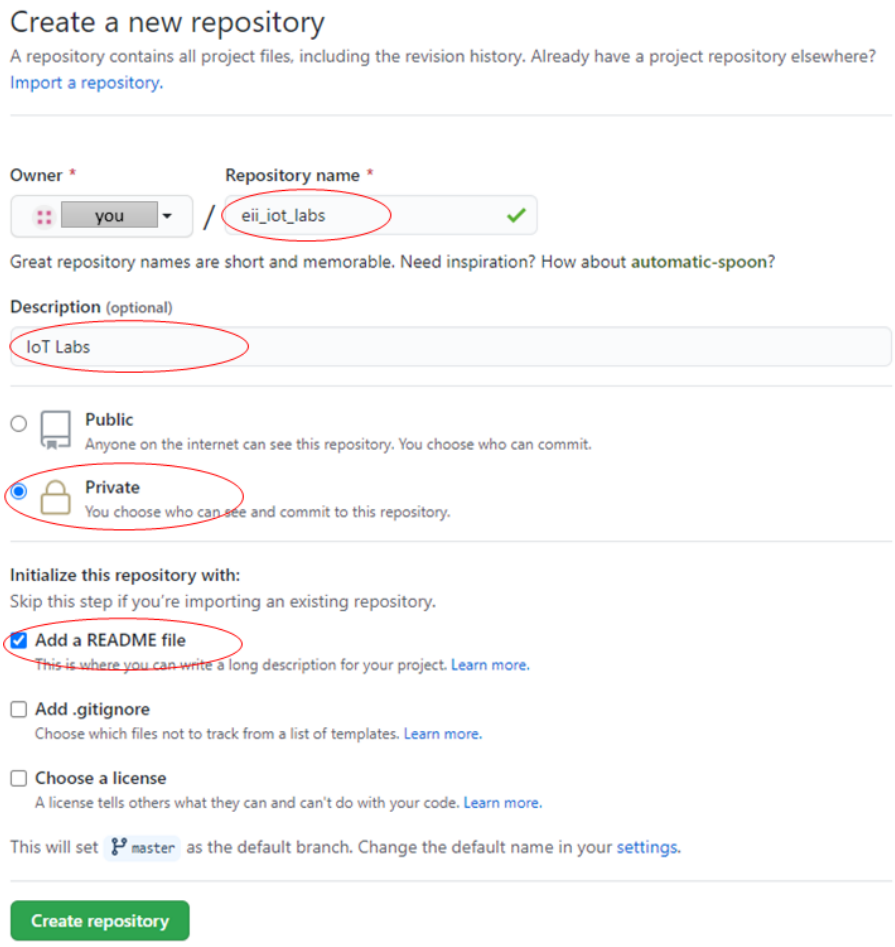


Figure 1.1 – *Create a GitHub repository.*

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.

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)

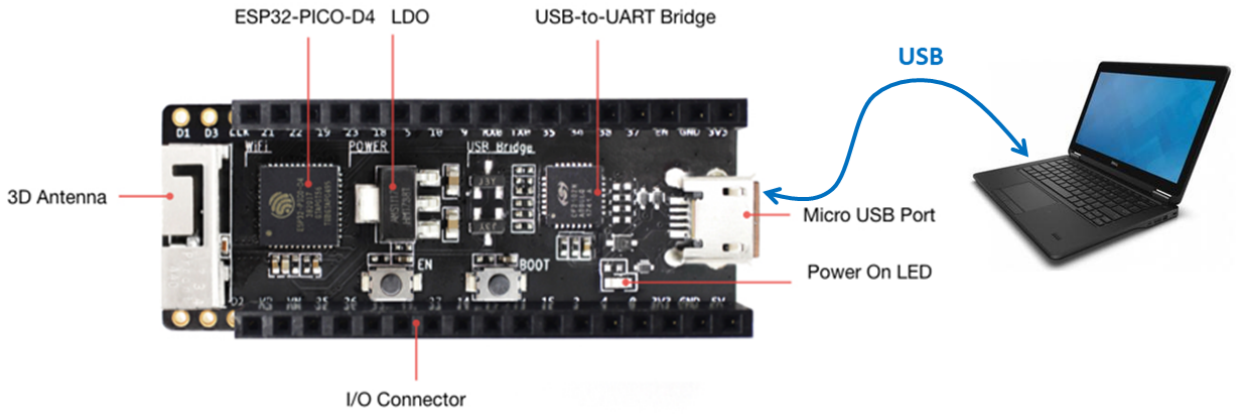


Figure 1.2 – *ESP32-PICO-KIT board connections.*

- 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

```
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...
```

1.2 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 .
```

- 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 a script file *create_esp32_prj.sh* in Labs folder. You have to copy the contains at the end of the *.bashrc* file.

- Edit the *.bashrc* file.

```
esp32:~/labs/part1_iot_framework/lab1_framework$ gedit ~/.bashrc
```

- Copy the contents of *create_esp32_prj.sh* file to the end of *.bashrc* file.
- Close the terminal
- Open a new terminal and check if the function now exists. 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!

LAB N° 2

Debugging with Visual Studio Code

Lab Objectives

- Understand the principle of the on-chip debugger.
- Debug a program with Visual Studio Code.

2.1 Principle

Figure 2.1 illustrates the operating principle for debugging the ESP32 processor. Debugging is generally done using a debugging tool which will be GDB (GNU Project Debugger). GDB communicates with Visual Studio Code and OpenOCD (Open On-Chip Debugger). OpenOCD communicates with the ESP-PROG JTAG board from Espressif Systems which is connected to the ESP32-PICO-D4 board via a JTAG communication (Joint Test Action Group, Industry standard IEEE 1149.1) and to the computer via an USB communication. This ESP-PROG JTAG card acts as a gateway between the ESP32-PICO-D4 board and OpenOCD board.

2.2 Wiring ESP-PROG JTAG board

Figure 2.2 shows how to wire the JTAG communication. You have 4 wires for the TAP (Test Access Port), respectively Test Data In (TDI), Test Mode Select (TMS), Test Clock (TCK), Test Data Out (TDO). Do not forget to connect the GND wire.

2.3 Create the Lab

We will create the « lab2_debug » project and run Visual Studio Code as below :

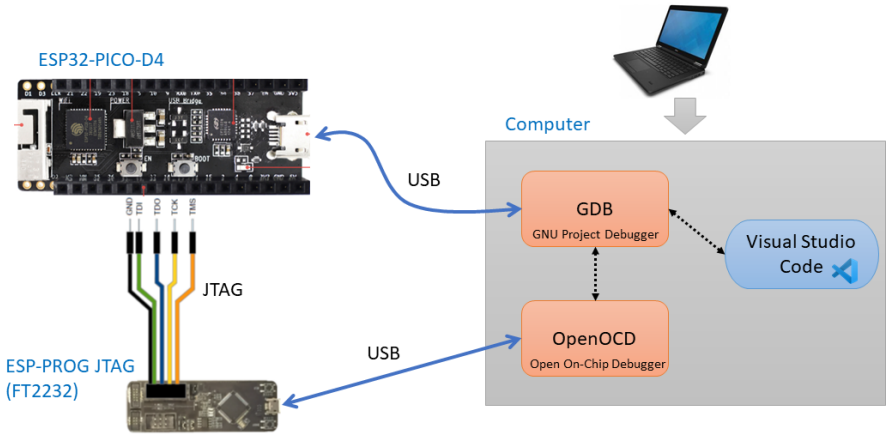


Figure 2.1 – *Debugging principle.*

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

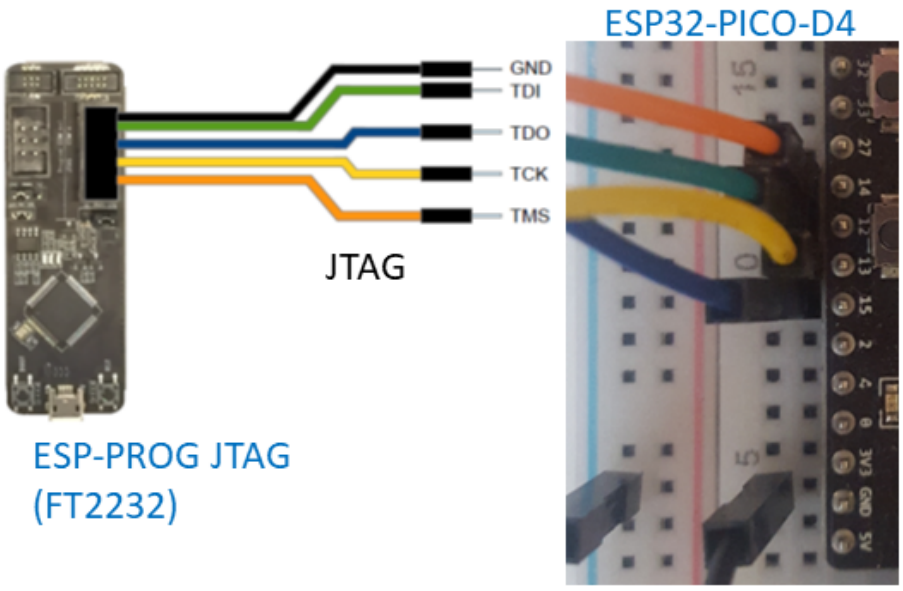


Figure 2.2 – *Wiring ESP-PROG JTAG board.*

1. Create the lab

```
esp32:~$ cd ~/labs
esp32:~/labs$ cd part1_iot_framework
esp32:~/../part1_iot_framework$ git clone https://github.com/fmuller-pns/
    esp32-vscode-project-template.git
esp32:~/../part1_iot_frameworks$ mv esp32-vscode-project-template lab2_debug
```

```
esp32:~/../part1_iot_framework$ cd lab2_debug
esp32:~/../part1_iot_framework/lab2_debug$ rm -fR .git
esp32:~/../part1_iot_framework/lab2_debug$ code .
```

2. Overwrite the « main.c » file by the provided code of the « main.c » file in the « lab2_esp32_debug » folder.

2.4 First run for Debug

The procedure to start the debugger is explained in the Visual Studio Code project template located in GitHub.

2.5 Debugging

2.5.1 Execution of the program

1. Run the program.
2. What does this program do ?

2.6 Debugging Walk through

1. Indicate a breakpoint at the line below. To do this, click to the right of the line number to display a red dot

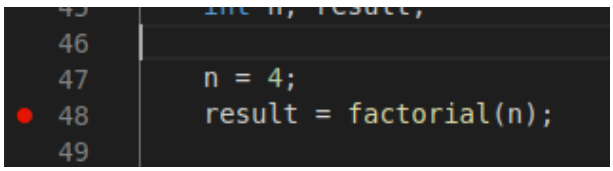


Figure 2.3 – *Wiring ESP-PROG JTAG board.*

2. What does this program do ?

3. Run the debugger to the breakpoint (cf. figure 2.4)
4. Examine the situation at the breakpoint : variables, call stacks, watch, breakpoints.
5. Click on *Step into* command until entering the loop of the factorial function.
6. Examine the new situation : variables, call stacks, watch, breakpoints.
7. Click on *Step into* command to run step by step.

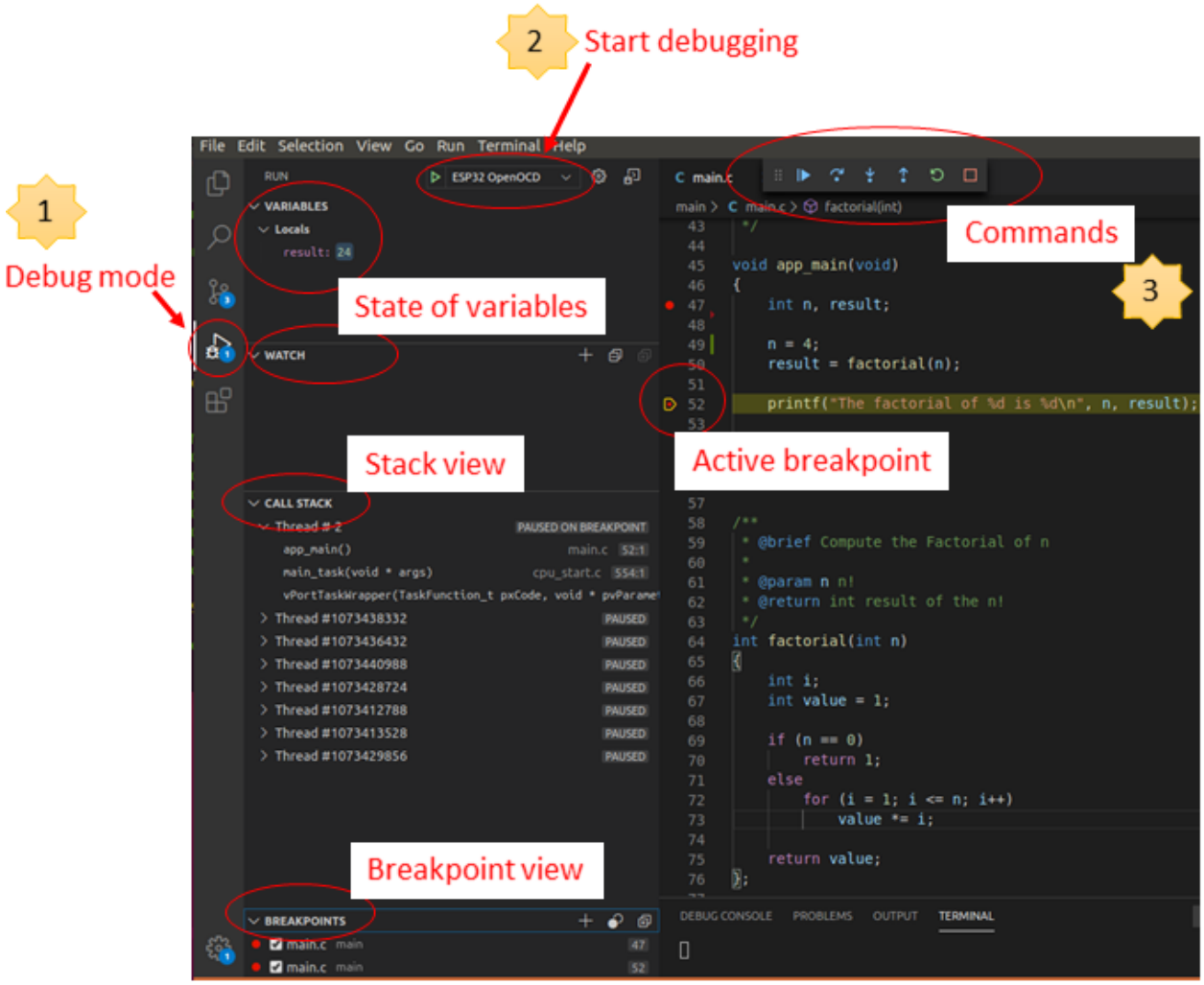


Figure 2.4 – *Debugging with Visual Studio Code.*

8. Click on *Step out* command to exit of the factorial function.
9. Click on *Continue till next break* command. The program is executed until the end because there is only 1 breakpoint defined.
10. Add a second breakpoint in the loop of the factorial function.
11. Click on *Restart* command.
12. Click on *Continue till next break* command. You are normally on the second breakpoint (loop of the factorial function)
13. Examine the new situation : variables, call stacks, watch, breakpoints.
14. Click on *Continue till next break* command. You will be on the second iteration of the loop.
15. Examine the new situation : variables, call stacks, watch, breakpoints.
16. Unvalidate the breakpoint that is in the loop. Click on the red dot or uncheck the correct breakpoint in the breakpoint view.
17. Click on *Continue till next break* command. The program runs until the end.

Now you know how to debug a program!

LAB N° 3

Working with C and IDF framework

Lab Objectives

- Useful functions : usleep, random, sprintf, strcpy, malloc, ESP_LOG ...
- Preprocessor commands (#define, #ifdef, #include ...)
- Master useful keywords like static, const, extern ...
- Define a structure, an enumeration and an union. Use designated initializer. Define structures as function arguments, access structure members.
- Refactor program with header files.
- Use dynamic allocation.
- Define IDF components.
- Use GitHub and Doxygen.

3.1 Useful functions (Lab3-1)

3.1.1 Create the Lab

We will create the « lab3-1_useful_functions » and run Visual Studio Code as below :

1. Create the lab

```
esp32:~$ cd ~/labs
esp32:~/labs$ cd part1_iot_framework
esp32:~/../part1_iot_framework$ git clone https://github.com/fmuller-pns/
    esp32-vscode-project-template.git
esp32:~/../part1_iot_frameworks$ mv esp32-vscode-project-template lab3-1
    _useful_functions
esp32:~/../part1_iot_framework$ cd lab3-1_useful_functions
esp32:~/../part1_iot_framework/lab3-1_useful_functions$ rm -fR .git
esp32:~/../part1_iot_framework/lab3-1_useful_functions$ code .
```

2. Overwrite the « main.c » file by the provided code of the « lab3-1_main.c » file.

3.1.2 Understanding the program

The entry point in IDF Framework is the *app_main()* function (we will see later, it is more like a task). Remember that in the classic C program, the entry point is the *main()* function. We are not interested in the *keyboardInput()* function at this time.

1. Answer the following questions :
   - What is #include ? Web help
   - What is #define ? Web help
2. Find out on the Web how to use these functions :
   - printf()
   - usleep()
   - ESP_LOGE(), ESP_LOGW(), ESP_LOGI(), ESP_LOGD(), ESP_LOGV(), Web help
   - esp_log_level_set()
3. Complete the program except the *getTemperature()* function that we will see later.
4. Launch the program and analyze the result.
   Tip : Remember that the full help is in the section Getting Started to run the program on the board.

3.1.3 Behavior of the temperature function code

1. Find out on the Web how to use these functions :
   - esp_random(), Web help

- abs()

2. What does the *getTemperature()* function return as a result ?

3. Write the behavior of the *getTemperature()* function that returns a random temperature value between the offset and a maximum value.
4. Launch the program and analyze the result.

3.1.4 keyboardInput() function

We will need to enter values or strings throughout the Labs. The following function meets this need.

1. Find out on the Web how to use these functions :
   - memset()
   - getchar(), Web help
   - strlen()
   - atoi()
   - ESP_OK, ESP_ERROR_CHECK, Web help

2. In the *app_main* function, add this code below just before the while loop.

```
char str[10];
printf("Enter offset : ");
ESP_ERROR_CHECK ( keyboardInput(str, 10) );
offset = atoi(str);
```

3. Build the project. What is wrong ? correct the problem.

4. What does this code do ?

5. Launch the program and analyze the result.

3.2 Understanding the structures (Lab3-2)

3.2.1 Create the new Lab

We will create the « lab3-2-1_working_with_c » and run Visual Studio Code as below :

1. Create the Lab

```
esp32:~$ cd ~/labs
esp32:~/labs$ cd part1_iot_framework
esp32:~/../part1_iot_framework$ git clone https://github.com/fmuller-pns/
    esp32-vscode-project-template.git
esp32:~/../part1_iot_framework$ mv esp32-vscode-project-template lab3-2-1
    _working_with_c
esp32:~/../part1_iot_framework$ cd lab3-2-1_working_with_c
esp32:~/../part1_iot_framework/lab3-2-1_working_with_c$ rm -fR .git
esp32:~/../part1_iot_framework/lab3-2-1_working_with_c$ code .
```

2. Overwrite the « main.c » file by the provided code of the « lab3-2-1_main.c » file.

3.2.2 Working on the program

1. Answer the following questions :
   - What is a structure ? Web help
   - What is an union ? Web help
   - What is an enumerate ? Web help

- What is a typedef ? Web help

- What are these keywords used for ?
  — static Web help

  — const Web help

- what is a designated initializer ? Web help

2. Declare and initialize a *sensorSet* variable (with a designated initializer) In the *app_main()* function. The variable is a *SensorSetStruct* structure. The values are :
   - Temperature name = "TEMP"
   - Temperature type = FLOAT
   - Temperature value = 0.0
   - Pressure name = "PRES"
   - Pressure type = INT
   - Pressure value = 0
3. Launch the program and analyze the result.
   Tip : Remember that the full help is in the section Getting Started to run the program on the board.
4. Add a breakpoint in the *app_main()* function, at line where we call *updateAndDisplaySensorValues()* function.
5. Launch the program in debug mode and watch the *sensorSet* variable at each iteration.

3.2.3 Function arguments

We will pass the two *Update()* and *Display()* default functions through parameters of the *updateAndDisplaySensorValues()* function as below :

```
void updateAndDisplaySensorValues(
        struct SensorSetStruct *sensorSet,
        void (*Update)(struct SensorSetStruct *sensorSet),
        void (*Display)(struct SensorSetStruct *sensorSet)) {

/* TODO */

}
```

1. Duplicate the « lab3-2-1_working_with_c » folder to « lab3-2-2_working_with_c »
```
esp32:~/../part1_iot_framework$ cp -r lab3-2-1_working_with_c lab3-2-2
    _working_with_c
esp32:~/../part1_iot_framework$ cd lab3-2-2_working_with_c
esp32:~/../part1_iot_framework/lab3-2-2_working_with_c$ idf.py fullclean
esp32:~/../part1_iot_framework/lab3-2-2_working_with_c$ code .
```
2. Modify behavior of the *updateAndDisplaySensorValues()* function to use function arguments.
3. Modify the call of the *updateAndDisplaySensorValues()* function.
4. Launch the program and analyze the result.
5. Add a breakpoint in the *app_main()* function, at line where we call *updateAndDisplaySensorValues()* function.
6. Launch the program in debug mode and watch the *sensorSet* variable at each iteration.

3.2.4 Structure members

We will now directly declare the *Update()* and *Display()* functions in the *SensorSetStruct* structure as below :

```
typedef struct SensorSetStruct
{
  Sensor temperature;
  Sensor pressure;
  /* Add structure members */
  void (*Update)(struct SensorSetStruct *sensorSet);
  void (*Display)(struct SensorSetStruct *sensorSet);
} SensorSet;


void updateAndDisplaySensorValues(
        struct SensorSetStruct *sensorSet) {

/* TODO */

}
```

1. Duplicate the « lab3-2-2_working_with_c » folder to « lab3-2-3_working_with_c »
```
esp32:~/../part1_iot_framework$ cp -r lab3-2-2_working_with_c lab3-2-3
    _working_with_c
esp32:~/../part1_iot_framework$ cd lab3-2-3_working_with_c
esp32:~/../part1_iot_framework/lab3-2-3_working_with_c$ idf.py fullclean
esp32:~/../part1_iot_framework/lab3-2-3_working_with_c$ code .
```
2. Add the two structure members to the *SensorSetStruct* structure.
3. Modify the *updateAndDisplaySensorValues()* function behavior.

4. Add the default functions (i.e. *defaultUpdateSensorValues()* and *defaultPrintSensorValues()*) to the designated initializer in the *app_main()* function.
5. Launch the program and analyze the result.
6. Add a breakpoint in the *app_main()* function, at line where we call *updateAndDisplaySensorValues()* function.
7. Launch the program in debug mode and watch the *sensorSet* variable at each iteration.
   - What are the address of the *Update* and *Display* fields of the *SensorSetStruct* structure ?
   - What do these 2 addresses correspond to ?
   - In what memory are the functions stored ?
8. What is the interest of declaring these functions in the structure ?

3.3 Refactoring code (Lab3-3)

All the code is in one file. It is recommended to refactor the code for better readability, flexibility and usability. We are going to move sensor functions into separate files. We usually have the header file and the source file which contains the function code.

3.3.1 Refactoring code

1. Duplicate the « lab3-2-3_working_with_c » folder to « lab3-3-1_c_header_files »

```
esp32:~/../part1_iot_framework$ cp -r lab3-2-3_working_with_c lab3-3-1
   _c_header_files
esp32:~/../part1_iot_framework$ cd lab3-3-1_c_header_files
esp32:~/../part1_iot_framework/lab3-3-1_c_header_files$ idf.py fullclean
esp32:~/../part1_iot_framework/lab3-3-1_c_header_files$ code .
```

2. Create two new blank files : *sensors.h* and *sensors.c*
3. In the *sensors.c*
   - Include header files in the *sensors.c* (#include ...)
   - Move TAG constant
   - Move *defaultUpdateSensorValues()*, *defaultPrintSensorValues()*, *updateAndDisplaySensorValues()* functions
   - Move *sensorSet* variable and rename it *defaultSensorSet* as below :

```
struct SensorSetStruct defaultSensorSet = {
...
};
```

4. What are the *extern* keyword used for ?

5. In the *sensors.h*
   - Add conditional macros. Why does we do that ?

```
#ifndef _SENSORS_H_
#define _SENSORS_H_

...

#endif
```

   - Move enumeration, structures and types
   - Declare the 3 prototype functions : *defaultUpdateSensorValues()*, *defaultPrintSensorValues()*, *updateAndDisplaySensorValues()*

6. the *main.c* file becomes very small. The only function is *app_main()*.
   - Modify the code of these function as below :

```
void app_main(void) {
  int i;
  while (1) {
    ESP_LOGI(TAG, "DEFAULT Functions - SENSORS");
    for (i=0; i<5; i++) {
      // Wait for 1 sec.
      usleep(1000000);
      updateAndDisplaySensorValues(&defaultSensorSet);
    }
  }
}
```

   - Do not forget to put this line in the right place to reference function and so on.

```
#include "sensors.h"
```

7. What does the *app_main()* function exactly do ?

8. Build the code. Why is there an error on the *defaultSensorSet* variable ?

9. In the *sensors.h*, add this line. What does the line do ?

```
extern struct SensorSetStruct defaultSensorSet;
```

10. Launch the program and analyze the result.

3.3.2 Reference custom functions

1. Duplicate the « lab3-3-1_c_header_files » folder to « lab3-3-2_c_header_files »

```
esp32:~/../part1_iot_framework$ cp -r lab3-3-1_c_header_files lab3-3-2
    _c_header_files
esp32:~/../part1_iot_framework$ cd lab3-3-2_c_header_files
esp32:~/../part1_iot_framework/lab3-3-2_c_header_files$ idf.py fullclean
esp32:~/../part1_iot_framework/lab3-3-2_c_header_files$ code .
```

2. Overwrite the provided code of the « lab3-3-2_main.c » file in the « main.c » file.
3. Explain the code of the *app_main()* function, among others :
   - The use of *updateSensorValues()* and *printSensorValues()* functions
   - The global behavior

4. Launch the program and analyze the result.
5. What's wrong with this code if we're interested in the *defaultSensorSet* variable ?

6. What is the solution ? Do not code the solution.

3.4 Dynamic allocation (Lab3-4)

3.4.1 Create the Lab

We will create the « lab3-4_memory_allocation » and run Visual Studio Code as below :

1. Duplicate the « lab3-3-2_c_header_files » folder to « lab3-4_memory_allocation »

```
esp32:~/../part1_iot_framework$ cp -r lab3-3-2_c_header_files lab3-4
    _memory_allocation
esp32:~/../part1_iot_framework$ cd lab3-4_memory_allocation
esp32:~/../part1_iot_framework/lab3-4_memory_allocation$ idf.py fullclean
esp32:~/../part1_iot_framework/lab3-4_memory_allocation$ code .
```

2. Overwrite the « main.c » file by the provided code of the « lab3-4_main.c file.

3.4.2 Understanding the program

1. Answer the following questions :
   - What is the *sizeof* keyword ? Web help
   - What is a pointer ? Web help
   - How does arithmetic work for pointers ? Web help
   - What is an array of pointers ? Web help
2. Find out on the Web how to use these functions :
   - malloc(), calloc(), free(), Web help
   - memcpy(), strlen(), strcpy(), Web help

- sprintf(), Web help

- esp_get_free_heap_size(), Web help

3. Create a new function *createDefaultSensorSet()* at the end of the *sensors.c* file. This function returns a copy of the *defaultSensorSet* global variable already defined in the same file.

```
struct SensorSetStruct * createDefaultSensorSet() {
  // Declare a local variable named sensorSet which will be return
     later

  // Allocate a memory space for the sensorSet variable

  // Check that there was no allocation error

  // Copy the default structure (i.e. defaultSensorSet global
     variable) in the sensorSet variable

}
```

4. Modify in the *sensors.h* file the name field. We use

```
typedef struct SensorStruct
{
  char *name;    // char name[10];
  enum SensorValueType type;
  union SensorValueUnion value;
} Sensor;
```

5. Understand the behavior of *app_main()* function.
6. Build. Adapt the code to remove compilation errors.
7. Add a breakpoint at the line below :

```
ESP_LOGI(TAG, "DEFAULT Functions - SENSORS");
```

8. Launch the program in debug mode and watch defaultSensorSet and customSensorSet variables.
9. Launch the program till the end of the execution (about 12 sec.) and analyze the result.

- What is the size of the *SensorSetStruct* structure ?
- Interpret the evaluations of $h_i - h_{i+1}$ printed at the end of the program

3.5 Linked list with pointers (Lab3-5)

3.5.1 Create the Lab

We will create the « lab3-5_linked_list » and run Visual Studio Code as below :

1. Create the lab

```
esp32:~$ cd ~/labs
esp32:~/labs$ cd part1_iot_framework
esp32:~/../part1_iot_framework$ git clone https://github.com/fmuller-pns/
    esp32-vscode-project-template.git
esp32:~/../part1_iot_frameworks$ mv esp32-vscode-project-template lab3-5
    _linked_list
esp32:~/../part1_iot_framework$ cd lab3-5_linked_list
esp32:~/../part1_iot_framework/lab3-5_linked_list$ rm -fR .git
esp32:~/../part1_iot_framework/lab3-5_linked_list$ code .
```

2. Overwrite the « main.c » file by the provided code of the « lab3-5_main.c file.
3. Copy provided « lab3-5_sensors.c » and « lab3-5_sensors.h » files and rename it : « sensors.c » and « sensors.h ».

3.5.2 Understanding the program

1. Answer the following questions :
   - What is the linked list ? Web help

2. Explain the *SensorStruct* and *SensorSetStruct* types in the *sensor.h* file.

3. Explain the behavior of the *createDefaultSensorSet()* function (main steps) by adding comments in the code.

4. Explain the behavior of the *freeSensorSet()* function (main steps) by adding comments in the code.

5. Understand the behavior of *updateSensorValues()* and *printSensorValues()* functions.
6. Understand the behavior of app_main() function.
7. Add a breakpoint a the line below :

```
ESP_LOGI(TAG, "DEFAULT Functions - SENSORS");
```

8. Launch the program in debug mode and watch *defaultSensorSet* and *customSensorSet* variables. Check that the addresses of the pointers are different for the 2 variables.

9. Launch the program till the end of the execution and analyze the result.
   - What is the size of the *SensorSetStruct* structure ? Why is it smaller than in the previous « lab3-4_memory_allocation » ?

   - Interpret the evaluations of $h_i - h_{i+1}$ printed at the end of the program.

10. What is the advantage of this linked list version over the previous one in « lab3-4_memory_allocation » ?

LAB N° 4

IDF Configuration and Components

Lab Objectives

- Define an IDF Configuration
- Create an IDF Components

4.1 IDF Configurations (Lab4-1)

The objective is to configure the « lab3-5_linked_list » lab by using the configuration menu available in the framework. For this we are going to create a new lab « lab4-1_menuconfig ».

4.1.1 Create the Lab

1. Duplicate the « lab3-5_linked_list » folder to « lab4-1_menuconfig »

```
esp32:~/../part1_iot_framework$ cp -r lab3-5_linked_list lab4-1_menuconfig
esp32:~/../part1_iot_framework$ cd lab4-1_menuconfig
esp32:~/../part1_iot_framework/lab4-1_menuconfig$ idf.py fullclean
esp32:~/../part1_iot_framework/lab4-1_menuconfig$ code .
```

2. Overwrite the « main.c » file by the provided code of the « lab4-1_main.c » file. Do not forget to rename it to « main.c »).
3. Copy the « lab4-1_Kconfig.projbuild » provided file to *main* folder. Do not forget to rename it to « Kconfig.projbuild »).

4.1.2 Manipulate configuration menu

When the program starts to run, a warning appears repeatedly. The objective is to remove this warning by using the configuration menu.

1. Run the program.
```
esp32:~/../part1_iot_framework/lab4-1_menuconfig$ idf.py -p /dev/ttyUSB0
    flash monitor
...
I (292) spi_flash: flash io: dio
W (292) spi_flash: Detected size(4096k) larger than the size in the binary
    image header(2048k). Using the size in the binary image header.
I (302) cpu_start: Starting scheduler on PRO CPU.
I (0) cpu_start: Starting scheduler on APP CPU.
```

2. Look for the following warning. This is due to a bad default configuration. Indeed, the ESP32-PICO-D4 board embeds 4MiB of flash memory instead of 2MiB.
3. Open the *sdkconfig* file and search the « CONFIG_ESPTOOLPY_FLASHSIZE_2MB » and « CONFIG_ESPTOOLPY_FLASHSIZE » parameters. We would rather have the parameter « CONFIG_ESPTOOLPY_FLASHSIZE_4MB » defined !
4. Run the menu config
```
esp32:~/../part1_iot_framework/lab4-1_menuconfig$ idf.py menuconfig
```
5. Go to « Serial flasher config » / « Flash size ». Press Return, Select 4MB by press Spacebar and then exit with saving.
6. Open the *sdkconfig* file and search the « CONFIG_ESPTOOLPY_FLASHSIZE_2MB » and « CONFIG_ESPTOOLPY_FLASHSIZE_4MB » parameters. The configuration is now correct. We will restart the program.

```
esp32:~/../part1_iot_framework/lab4-1_menuconfig$ idf.py -p /dev/ttyUSB0
    flash monitor
```

7. What do you see ?

4.1.3 Customizing menu configuration

1. Answer the following question :
   - What represent the « Kconfig.projbuild » ? Web help 1, Web help 2

2. Explain the content of the « Kconfig.projbuild » configuration file.

- What is the name of the custom menu ?

- What is the type of *CONSTANT* and *CUSTOM_FUNCTIONS* parameters ?

3. Explain the role of the *CONFIG_CONSTANT* and *CONFIG_CUSTOM_FUNCTIONS* configuration value located in the *main.c* file.

4. Run the configuration menu and activate the CUSTOM_FUNCTIONS
```
esp32:~/../part1_iot_framework/lab4-1_menuconfig$ idf.py menuconfig
```

5. Open the *sdkconfig* file and check the *CUSTOM_FUNCTIONS* parameter
6. Run the program and check if the modification has been taken into account.

```
esp32:~/../part1_iot_framework/lab4-1_menuconfig$ idf.py -p /dev/ttyUSB0
   flash monitor
```

7. We would like to display a title editable via configuration menu.
   - Add the « INFORMATION_TITLE » parameter of type *string*. The default title will be « My Title ».
   - Add a help thanks to the *help* keyword (*help* hello !)
   - Run the configuration menu and change the default title (for example « Sensor Information »)
     ```
     esp32:~/../part1_iot_framework/lab4-1_menuconfig$ idf.py menuconfig
     ```
   - Modify the code. The line of code that displays the title is at the start of the *printSensorValues()* function.
   - Run the program and verify.
8. We want to change the delay (currently 1 sec.) via configuration menu.
   - Add the « UPDATE_DELAY » parameter of type *int*. The unit will be in ms. The default value will be « 1000 » (1 sec.).
   - We want the values to be between 1 second and 5 seconds. The *range* keyword can do it (*range* min_value max_value).
   - Run the configuration menu and change the default value (for example 1.2 sec.)
   - Modify the 2 lines of the code where the *usleep()* function is called.
   - Run the program and verify.

4.1.4 Default configuration

If you don't want to specify a full *sdkconfig* configuration, it is possible to override some key values from the ESP-IDF default configuration. So you can create a file *sdkconfig.defaults* in the project root directory (Web help). The syntax is the same as the *sdkconfig* file.

1. Create a *sdkconfig.defaults* file in Visual Studio Code.
2. We will define by default the size of the flash memory (4MiB) which will avoid the warning that we saw in section 4.1.2. Add these lines in the file :
```
# ESP32 with 4MB flash size
CONFIG_ESPTOOLPY_FLASHSIZE_4MB=y
CONFIG_ESPTOOLPY_FLASHSIZE="4MB"
```
3. Delete the *sdkconfig* file.
4. Run the configuration menu to verify the new default size of the flash.
5. Build the project
   - Verify the sdkconfig file
   - Run and verify if the warning is not present.
6. Add a default custom configuration for user parameters.
   - CUSTOM_FUNCTIONS : true
   - UPDATE_DELAY : 1.5 second
   - INFORMATION_TITLE : =- Informations -=
   - RANDOM_VALUE : ONLY_RANDOM
7. Delete the *sdkconfig* file.
8. Run the configuration menu to verify the new default parameters.
9. Run the program and verify.

4.2 IDF Components (Lab4-2)

The *sensors.h* and *sensors.c* files could be put in library in order to have a more reusable code structuring. So, we are going to create a IDF component (Web help). Indeed, it is recommended grouping most developments into components instead of putting them all in main directory.

For this we are going to create a new lab « lab4-2_idf_components ».

4.2.1 Create the Lab

1. Duplicate the « lab4-1_menuconfig » folder to « lab4-2_idf_components »
```
esp32:~/../part1_iot_framework$ cp -r lab4-1_menuconfig lab4-2_idf_components
esp32:~/../part1_iot_framework$ cd lab4-2_idf_components
esp32:~/../part1_iot_framework/lab4-2_idf_components$ idf.py fullclean
esp32:~/../part1_iot_framework/lab4-2_idf_components$ code .
```

4.2.2 Create the IDF Component

We are going to create a *sensorsManagement* component which contains *sensor.h* and *sensor.c* files.

1. Create a *components* folder.
2. In this folder, create a *sensorsManagement* folder that corresponds to *sensorsManagement* component.
3. Move *sensor.c* files to *sensorsManagement* folder.
4. In the *sensorsManagement* folder, create a *include* folder.
5. Move *sensor.h* files to *sensorsManagement* folder.
6. We need to configure the component using the *CMakeLists.txt* and *component.mk* files. Theses 2 files may contain variable definitions to control the build process of the component.
   - *CMakeLists.txt* file is used for CMake (Web help).
   - *component.mk* file is used for Make (Web help). The minimal *component.mk* file is an empty file!
   - Copy the « lab4-2_CMakeLists.txt » provided file to *sensorsManagement* folder. Do not forget to rename it to « CMakeLists.txt »).
   - Copy also the « lab4-2_component.mk » provided file to *sensorsManagement* folder. Do not forget to rename it to « component.mk »).
7. Explain the content of the *CMakeLists.txt* file (Web help).

8. Build and run the program.

4.2.3 Customizing menu configuration for the IDF Component

In the *main* folder, we have added a *Kconfig.projbuild* file to configure the program. In the case of an IDF component, the file is called *Kconfig*.

1. Create a blank *Kconfig* file in the *sensors Management* component.
2. We will create 2 parameters, TEMP_SENSOR_NAME (default value = "TEMPERATURE") and PRES_SENSOR_NAME (default value = "PRESSURE") which will be the default value of the *name* fields of the *defaultSensorSet* static variable defined in *sensors.c* file. The name of the configuration menu will call "sensors Management".
   - Write the code in the *Kconfig* file.
   - Modify the *sensors.c* file in order to use these 2 parameters.
3. Run configuration menu and change the default value of the 2 parameters located in the *components* directory.

```
esp32:~/../part1_iot_framework/lab4-2_idf_components$ idf.py menuconfig
```

4. Run the program and check display.

Part II

IoT Architecture

LAB N° 1

ESP32 Architecture

Lab Objectives

- How to detect the configuration of the ESP32 architecture.
- Understanding of the role of different memories.
- Programming a simple component : GPIO (General Purpose Input/Output).
- Understanding the interrupts.

1.1 Architecture (Lab1-1)

We will focus on the various components of the ESP32 architecture and their configuration. Each component can be configured using the configuration menu that we studied previously (cf. section 4.1). We can access this information from functions provided by ESP-IDF that we will discover in this Lab.

1.1.1 Create the Lab

We will create the « part2_architecture » folder, then create the « lab1-1_architecture » lab and run Visual Studio Code as below :

1. Create the lab.

```
esp32:~$ cd ~/labs
esp32:~/labs$ mkdir part2_architecture
esp32:~/labs$ cd part2_architecture
esp32:~/../part2_architecture$ git clone https://github.com/fmuller-pns/esp32
    -vscode-project-template.git
esp32:~/../part2_architecture$ mv esp32-vscode-project-template lab1-1
    _architecture
esp32:~/../part2_architecture$ cd lab1-1_architecture
```

```
esp32:~/../part2_architecture/lab1-1_architecture$ rm -fR .git
esp32:~/../part2_architecture/lab1-1_architecture$ code .
```

2. Overwrite the « main.c » file by the provided code of the « lab1-1_main.c » file.

1.1.2 Understanding the program

1. Run the program.
2. Answer the following questions :
   - What is the CPU clock and its current frequency ? See in provided `Documentation/ESP32/ESP32_datasheet.pdf`, search C̈PU Clock¨
   - What is the number of CPU cores ?
   - What is the name of CPU cores ? Web help
   - What is the APB (Advanced Peripheral Bus) clock and its current frequency ? Web help
   - What is the XTAL clock frequency ?
   - What is the RTC (Real-Time Clock) and its current frequency ? Web help
3. Study the fields of the *esp_chip_info_t* structure. (To help you, put the cursor on the name of the structure and press *F12*)

4. Study the *spi_flash_get_chip_size()* function. Go to *esp_spi_flash.h* file to find out information

5. Complete the program to print Flash memory information (external/internal and size), Wifi and Bluetooth (Classic/LE) using *esp_chip_info_t* structure and *spi_flash_get_chip_size()* function.
6. Run the program and verify the information. What is the size of the flash memory ?

7. The ESP32-PICO-D4 board embeds 4MiB flash memory. Change the value as specify in section 4.1.2.
8. Run the program and verify the flash memory size.
9. Complete the program to print MAC (Media Access Control) address from the *esp_efuse_mac_get_default()* function. Go to *esp_system.h* file to find out information. We want to display the address in this format : XX-XX-XX-XX-XX-XX, X represents a hexadecimal number (for example, MAC address can be : d8-a0-1d-5c-c9-10). To print hexadecimal number, watch the example below (Web help) :

```
int a = 0x45;
printf(" Print a = %02x\n", a); // Result:  Print a = 45
```

10. Run the program. What is the MAC address ?

11. Complete the program to known if the OCD Debug mode is active from the *esp_cpu_in_ocd_debug_mode()* function. Go to *cpu.h* file.
12. Run the program without connecting the JTAG and after trying to connect the JTAG and run again the program.

1.2 System Time (Lab1-2)

It is generally important in an IoT architecture to have a time reference. This system time is implemented from an Real-Time Clock (RTC). This time system will make it possible to tag a sensor measurement, to measure a time or a latency, to trigger a measurement or quite simply to obtain the time. Moreover, we had already studied the *usleep()* function which allows to wait for a certain duration. Note that this card does not keep the current time

when it is off. We will see later how to update the time from an internet connection.
First, we create a new lab :

1. Create the « lab1-2_rtc » lab, change memory flash size (4MiB) and run Visual Studio Code.
2. Overwrite the « main.c » file by the provided code of the « lab1-2-1_main.c » file.

1.2.1 System time - gettimeofday() function

The first solution is to use the *gettimeofday()* function.

1. Answer the following questions :
   - What is the *gettimeofday()* function and the *timeval* structure ? Web help
   - What is the *ets_delay_us()* function ? (To help you, put the cursor on the name of the function and press *F12*)
2. What does the value *197130* of *tv_sec* field correspond to ?
3. Display tv_n times (n=1,2,3 and 4), i.e. *tv_sec* and *tv_usec* fields. Also display $tv_{n+1} - tv_n$ time differences. Note that to display a long type, use the %ld specifier. Web help

```
long a = 123456789;
printf(" Print a = %ld\n", a); // Result:  Print a = 123456789
```

4. Run the program. Record the different times and interpret the results.

1.2.2 System time - <time.h> library

You can also use the standard *time.h* library.

1. Insert (do not overwrite !) at the end of the *app_main* function the provided code of the « lab1-2-2_code.c » file.

2. Answer the following questions :
   - Study the *time()*, *localtime_r()*, *mktime()* functions and the *tm* structure Web help, also press *F12* on the name to obtain help in Visual Studio Code.

   - What is the *strftime()* function ? Web help

3. Run the program and interpret the results.

1.2.3 Application - CPU Frequency and performance

We want to check the influence of the processor frequency on a piece of code below.

```
// Code to evaluate
volatile uint64_t i;
for (i=0; i<5000000; i++) {
}
```

1. What does the keyword *volatile* mean ? Web help

2. Evaluate the performance in time (in μs) of this piece of code according to the frequencies of the CPU (Freq = 80MHz, 160Mhz, 240Mhz).
   - Help you from *gettimeofday()* function.
   - Change the frequency using the configuration menu, cf. section 4.1.2 (menu : *config Component config/ESP32-Specific*).

   Write the frequency and time found.

3. Interpret the result, $Time = F(Freq)$.

LAB N° 2

Memory

Lab Objectives

- Utility of a memory mapping.
- Understanding the role of different memories.
- Understanding where the instructions and data are stored.
- Correctly manage the stack.

2.1 Flash & SRAM Memories (Lab2-1)

The figure 2.1 illustrates the interesting memories. The complete document is in `Documentation/ESP32/ESP32_datasheet.pdf`, chapter « Functional Description ».

The Flash memory is an external memory which stores permanent data (write and read access). A data can be a program (instructions) or a data (variables, arrays ...). The size depends on the board. The ESP32-PICO-D4 board embeds 4MiB flash memory as we saw previously.

The ESP32 SRAM (Static Random Access Memory) is an internal memory (on-chip SRAM) which stores volatile data (write and read access) and instructions, i.e. when you power off the board, the data is no longer present in the SRAM. This size is 520 KiB.

We have 8 KiB of RTC FAST SRAM (can be used for data storage) and 8 KiB of RTC SLOW SRAM (can be accessed by the co-processor during the Deep-sleep mode). We will see these memories later.

First, we create a new lab :

1. Create the « lab2-1_iram_dram » lab from « esp32-vscode-project-template » GitHub

Category	Target	Start Address	End Address	Size
Embedded Memory	Internal ROM 0	0x4000_0000	0x4005_FFFF	384 KB
	Internal ROM 1	0x3FF9_0000	0x3FF9_FFFF	64 KB
	Internal SRAM 0	0x4007_0000	0x4009_FFFF	192 KB
	Internal SRAM 1	0x3FFE_0000	0x3FFF_FFFF	128 KB
		0x400A_0000	0x400B_FFFF	
	Internal SRAM 2	0x3FFA_E000	0x3FFD_FFFF	200 KB
	RTC FAST Memory	0x3FF8_0000	0x3FF8_1FFF	8 KB
		0x400C_0000	0x400C_1FFF	
	RTC SLOW Memory	0x5000_0000	0x5000_1FFF	8 KB
External Memory	External Flash	0x3F40_0000	0x3F7F_FFFF	4 MB
		0x400C_2000	0x40BF_FFFF	11 MB+248 KB
	External RAM	0x3F80_0000	0x3FBF_FFFF	4 MB

FIGURE 2.1 – *Mapping table.*

repository, change memory flash size (4MiB for ESP32-PICO-D4 board) and run Visual Studio Code.

2. Overwrite the « main.c » file by the provided code of the « lab2-1_main.c » file.
3. Answer the following questions :
   - What does the *heap_caps_get_free_size()* function return with a *MALLOC_CAP_EXEC*, *MALLOC_CAP_8BIT* and *MALLOC_CAP_32BIT* argument ? (To help you, put the cursor on the name of the function and press *F12*) and Web help

   - What does *heap_caps_get_largest_free_block()* function return ?

4. Run the program and extract these lines to your console. The values are not necessarily identical.

```
I (0) cpu_start: App cpu up.
I (235) heap_init: Initializing. RAM available for dynamic allocation:
I (242) heap_init: At 3FFAE6E0 len 00001920 (6 KiB): DRAM
I (248) heap_init: At 3FFB30E8 len 0002CF18 (179 KiB): DRAM
I (254) heap_init: At 3FFE0440 len 00003AE0 (14 KiB): D/IRAM
I (261) heap_init: At 3FFE4350 len 0001BCB0 (111 KiB): D/IRAM
I (267) heap_init: At 400899F4 len 0001660C (89 KiB): IRAM

...

DRAM run executable code = 220464 Bytes (215 KiB)
```

```
DRAM run executable code 32 bit = 389552 Bytes (380 KiB)
IRAM = 91624 bytes (89 KiB)
DRAM (Total) = 297928 Bytes (290 KiB)
DRAM Free Heap Size : 297928 Bytes (290 KiB)
DRAM (Largest free block) = 169088 Bytes (165 KiB)
```

5. What is an IRAM and DRAM ? Web help

6. Find the values that we have displayed (except the line « DRAM run executable code 32 bit ») in the program according to the lines starting with « heap_init : » and which are displayed by the kernel. For example, the line « IRAM = 91624 bytes (89 KiB) » corresponds to the line « heap_init : At 400899F4 len 0001660C (89 KiB) : IRAM ».

7. Create an Excel Sheet (in your project folder) with 3 columns : Memory, Size (Byte), Size (KiB) as shown in figure 2.2.

	A	B	C
1		Size (Byte)	Size (KiB)
2	IRAM		
3	DRAM		
4			
5	Used static DRAM		
6	Used static IRAM		
7			
8	CPU Cache		
9	reserved		
10	padding		
11			
12	TOTAL	0	0
13			

Figure 2.2 – *Memory size.*

8. Complete the 2 values for « IRAM » and « DRAM (Total) ».

9. Each core has 32KiB cache memory, so we have 64KiB reserved for cache. Add this information in the « CPU Cache » line.
10. Set « Reserved » line is 8KiB.

2.2 The sections

The program is divided into sections. Each section contains a part of the program. We will study the main sections of a program.

2.2.1 The main sections

1. Summarizes the use of the following 3 sections : *.bss*, *.data* and *.rodata*. Web help

2. Run this command below.
```
esp32:~/../part2_architecture/lab2-1_iram_dram$ idf.py size
```

3. The Used static DRAM and IRAM is the size of the program after compilation (static keyword), respectively for the Data section and Instructions section. Add these values in the Excel sheet. What is the total size ?

4. Complete the padding line to obtain the 520KiB (SRAM memory size). Now it is right !
5. Run this new command below. What is the result ?
```
esp32:~/../part2_architecture/lab2-1_iram_dram$ idf.py size-files
```

6. identify the name of each columns.

7. To filter the code of the main.c file, run this command below.
```
esp32:~/../part2_architecture/lab2-1_iram_dram$ idf.py size-files | grep "
    main\|Total"
```

8. What is the size of .data and .bss section for DRAM ?

9. Add 2 variables before the *app_ main()* function.
```
uint32_t value = 12;
uint8_t table[1000];

void app_main(void) {
...
```

10. Run again the command below. What is the result ? Why ?
```
esp32:~/../part2_architecture/lab2-1_iram_dram$ idf.py size-files | grep main
```

11. Initialize the *table* with the *value* variable. Run again the command. What is the result ? Why ?

12. Move the *value* and *table* variables inside the *app_ main()* function. Run again the command below. What is the result ? Why ?

2.2.2 The stack section

The stack section is a section which is used to save local data declared in a function or a task, to save the return addresses during a function call or to save the context of an interrupt. The size reserved for the stack is therefore very large so as not to overflow. We are going to take an interest in this issue.

1. What does *uxTaskGetStackHighWaterMark()* function return ?

2. Add these 2 lines at the end of the *app_ main()* function.

```
int stackmem = uxTaskGetStackHighWaterMark(NULL);
printf("Stack space (app_main task) = %d Bytes (%.2f KiB)\n",
    stackmem, ((float)stackmem/1024.0));
char buffer [5000];
```

3. Run the program and get the size of the stack.

4. In which section is the *buffer[]* declared ?

5. Add this line to use the *buffer[]*. What does this *memset()* function do ?

```
memset(&buffer,1,sizeof(buffer));
printf("End of initialisation\n");
```

6. Run the program. What is the problem ?

7. Suggest 2 solutions to solve the problem.

2.2.3 Exercise : where are the strings ?

When using the *printf()* function, we usually define the string within the function as below :

```
printf("Hello!\n");
```

In order to display the size of the string, we will write the code in this way at the end of *app_ main* function :

```
char *mess = "Hello, where are the string?, length = %d\n";
printf(mess, strlen(mess));
```

We want to know in which section the strings are located.

1. Run the program. What is the length of the string ?

2. Run the command below and note the section values (in Bytes).
```
esp32:~/../part2_architecture/lab2-1_iram_dram$ idf.py size-files | grep "
    main\|Total"
```

3. Delete the start of the string : "Hello, ".
4. Run again the program. What is the new length of the string ?

5. Find out the new section values (in Bytes).

6. In which section the strings are located ? Justify.

7. Declare constant the mess *variable*.
8. In which section the string are located ? Justify.
```
const char *mess = ...
```

9. What is the conclusion ?

2.3 Flash Memory - Custom partition table (Lab2-2)

The objective is to learn how to customize the partition table for next labs. Indeed, a flash memory for ESP32 can contain multiple apps, as well as many different kinds of data : calibration data, file-systems, parameter storage, OTA app, etc.
First, we create a new lab :

1. Create the new « lab2-2_partitions » lab from « esp32-vscode-project-template » GitHub repository, change memory flash size (4MiB) and run Visual Studio Code.

2. Build the program.
```
esp32:~/../part2_architecture/lab2-2_partitions$ idf.py build
```

2.3.1 Customizing the partition table

The goal is to understand how to configure a partition table.

1. The *partition-table.bin* file (located at `./build/partition_table` contains the partition table but in a binary format. To generate a readable format, Python scripts exist in the component « partition_table » located in `~/esp/esp-idf/components/partition_table`.
   - Open a new terminal.
   - Go to the directory and list it.
```
esp32:~/../part2_architecture/lab1-2_partitions$ cd ~/esp/esp-idf/
   components/partition_table
esp32:~/../components/partition_table$ ll
```

   In this directory, you have a *gen_ esp32part.py* script to generate a CSV (Comma-separated values) format. You can open it with Visual Code Studio and Excel/Sheet. You have also a *partitions_ singleapp.csv* file which is a template for customize the partition table. However, another solution is to generate the CSV file from the built project.

2. Check if the *partition-table.bin* file exists.
```
esp32:~/../part2_architecture/lab2-2_partitions$ ll build/partition_table
```

3. Generate the partition table (CSV format) from the built project :

```
esp32:~/../part2_architecture/lab2-2_partitions$ gen_esp32part.py build/
   partition_table/partition-table.bin my_partitions.csv
```

4. Open the *my_ partitions.csv* file from Visual Studio Code. You have (SubType column) :
   - data, nvs : Non Volatile Storage (NVS) will see later. For example, it is used to store per-device PHY calibration data, to store WiFi data, etc.
   - data, phy : for storing PHY initialisation data. By default, PHY initialisation data is compiled into the app itself.
   - app, factory : it is the default app partition (type *app*). The partition of *app* type have to be placed at offsets aligned to 0x10000 (64Ki).

   More documentation can be found here.

5. You must now specify the custom partition table in the configuration.
   - Run menuconfig
```
esp32:~/../part2_architecture/lab2-2_partitions$ idf.py menuconfig
```
   - Go to *Partition Table* menu

- Go to *Partition Table (...)*
- Go to *Custom partition table CSV*
- Go to *Custom partition CSV file* and change the filename : *my_partitions.csv*
- Save and Exit

6. Verify the generated sdkconfig file. Search these lines :

```
CONFIG_PARTITION_TABLE_CUSTOM=y
CONFIG_PARTITION_TABLE_CUSTOM_FILENAME="my_partitions.csv"
CONFIG_PARTITION_TABLE_FILENAME="my_partitions.csv"
```

7. Build the project to check if there is no error.

2.3.2 Partition table with Over-The-Air (OTA)

The goal is to generate a partition table to use the OTA feature (OTA documentation). The OTA update mechanism allows a device to update itself based on data received while the normal firmware is running (for example, over WiFi or Bluetooth). This will be studied in a future lab.

1. You must now specify the custom partition table in the configuration.
   - Run menuconfig

     ```
     esp32:~/../part2_architecture/lab2-2_partitions$ idf.py menuconfig
     ```

   - Go to *Partition Table* menu
   - Go to *Partition Table (...)*
   - Go to *Factory app, two OTA definitions*
   - Save and Exit
2. Build the project. What is the problem ? Correct it.

3. Generate the partition table (CSV file) : *my_partitions_ota.csv*

```
esp32:~/../part2_architecture/lab2-2_partitions$ gen_esp32part.py build/
   partition_table/partition-table.bin my_partitions_ota.csv
```

4. Open the *my_partitions_ota.csv* file from Visual Studio Code. You have another lines (SubType column) :
   - ota, data : the OTA data partition specifies which OTA app slot partition should be booted next.
   - ota_0,app : OTA slot 0, version X of the program
   - ota_1,app : OTA slot 1, version Y of the program
5. You must now specify the new custom OTA partition table in the configuration.
   - Run menuconfig

     ```
     esp32:~/../part2_architecture/lab1-2_partitions$ idf.py menuconfig
     ```

   - Go to *Partition Table* menu

- Go to *Partition Table (...)*
- Go to *Custom partition table CSV*
- Go to *Custom partition CSV file* and change the filename : *my_partitions_ota.csv*
- Save and Exit

6. Verify the generated *sdkconfig* file. Search these lines :

```
CONFIG_PARTITION_TABLE_CUSTOM=y
CONFIG_PARTITION_TABLE_CUSTOM_FILENAME="my_partitions_ota.csv"
CONFIG_PARTITION_TABLE_FILENAME="my_partitions_ota.csv"
```

7. Build the project to check if there is no error.
   Now we are able to specify our own partition table for next labs !

2.3.3 Quiz

1. What is the use of a partition table ? For what memory ?

2. What elements are specified in this partition table file ?

3. What is the partition table file format ?

4. Where do we specify the partition table file ? In what way ?

5. What does OTA mean ? What is it used for ?

2.4 Storing files directly in flash memory (Lab2-3)

Sometimes you have a file with some binary or text data that you would like to reference to your program. It is precisely possible to store data from files which are referenced by variables in our program directly in flash memory. These data are generally read-only data (images, html documents, default configuration, etc.).
In this lab, we will store binary and text files in the flash memory and display them from within the program.

First, we create a new lab :

1. Create the « lab2-3_files_directly_in_flash » lab, change memory flash size (4MiB) and run Visual Studio Code.
2. Copy provided *files* folder to the *main* folder of your new lab.
3. Overwrite the « main.c » file by the provided code of the « lab2-3_main.c » file.

1. Build the program.
2. What is the problem ?

3. To solve it, first of all, it is necessary to add the variable *EMBED_TXTFILES* or *EMBED_FILES* in the `main/CMakeList.txt` file (more documentation) as below :

```
idf_component_register(
            SRC_DIRS "."
            INCLUDE_DIRS "."
            EMBED_TXTFILES "files/index.html" "files/note.txt"
            EMBED_FILES "files/polytech.png"
            )
```

4. If the project will become a component, it is recommended to reference file in the `main/component.mk` file as below :

```
COMPONENT_EMBED_TXTFILES := files/index.html
COMPONENT_EMBED_TXTFILES += files/note.txt
COMPONENT_EMBED_FILES := files/polytech.png
```

5. What is the difference between *EMBED_TXTFILES* or *EMBED_FILES* ((documentation for help)) ?

6. Build the program.
7. What is the first character of the *note.txt* file ? What is the length of the file ?

8. Open the *note.txt.S* assembler file located in *build* folder. Explain the link between the declaration in the *main.c* file and the *note.txt.S* file by helping you from the previous question.

To use a file referenced in flash memory, you must declare in the program a variable. From the (documentation) and *main.c* file, answer the following questions

1. What is the type of declaration for a variable in the program ?

2. How do you define the *start*, *end* and *length* symbol name of a file in assembler (directive *asm()*) ?

3. In which section are the variables declared ?

4. Run the program.

Now, we would like to create a new text file which contents my identity.

1. Create a text file (« my_identity.txt »).
2. Edit the file and add your identity.
3. Add code at the end of the *app_main* function to display the identity.
4. Test your program.

2.5 Non Volatile Storage memory (Lab2-4)

The figure 2.3 illustrates the steps for using Non Volatile Storage (NVS).

- Initialize flash memory.
- Open a namespace and get a storage handle. The default NVS partition is the one that is labeled « nvs » in the partition table.
- Read and Write values.
- Commit to ensure writing any pending changes to non-volatile storage.
- Close the storage handle and free any allocated resources.

2.5.1 First step with the NVS memory (Lab2-4-1)

1. Create the « lab2-4-1_nvs » lab from « esp32-vscode-project-template » GitHub repository, change memory flash size (4MiB for ESP32-PICO-D4 board) and run Visual Studio Code.

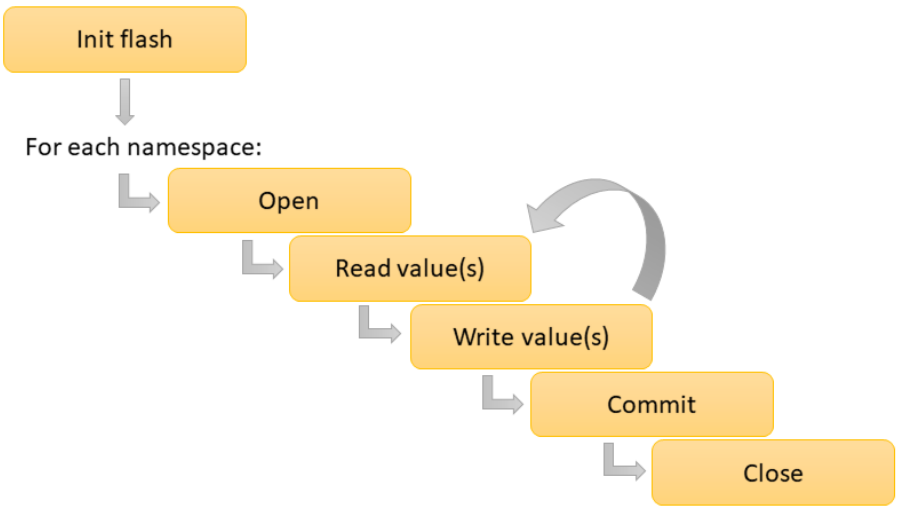


FIGURE 2.3 – *Non Volatile Storage programming steps.*

2. Overwrite the « main.c » file by the provided code of the « lab2-4-1_main.c » file.
3. Add comments in the *app_main* function to find the different steps.
4. Build the program.
5. Before running the program, we will make sure the *nvs* partition is blank. Thus, we will add a directive (*erase_flash*) to the usually used command.

```
esp32:~/../part2_architecture/lab2-4-1_nvs$ idf.py -p /dev/ttyUSB0
    erase_flash flash monitor
```

6. What is the message info ?

7. Push the reset button. What is the new message info ?

8. It is possible to modify the code without erasing values.
   - Add a message at the start of the *app_main()* function, using the *printf()* function.
   - Run the program with usually command.

```
esp32:~/../part2_architecture/lab2-4-1_nvs$ idf.py -p /dev/ttyUSB0 flash
    monitor
```

   - What is the message info ?

   - Push the reset button. What is the new message info and the value ? Conclusion ?

It is also possible to erase all contents of the default NVS partition labeled « nvs » from the *nvs_flash_erase()* function. You also can just erase one partition from the *nvs_flash_erase_partition()* function.

2.5.2 Customizing the NVS memory partition (Lab2-4-2)

In the previous section 2.5.1, we used the default « nvs » partition to write simple values. In this lab, we will learn how to use our own « nvs » partition using the partition table that we are going to modify. This will allow us to have more flexibility in terms of the memory size of our partition.

1. Duplicate the « lab2-4-1_nvs » folder to « lab2-4-2_custom_nvs ».
2. Generate the partition table (CSV format) from the built project as we did in a previous lab (cf. section 2.3.1).

```
esp32:~/../part2_architecture/lab1-2_partitions$ gen_esp32part.py build/
   partition_table/partition-table.bin my_partitions.csv
```

3. Open the *my_partitions.csv* file. What is the size in KiB of the default « nvs » partition ?

4. Declare at the end of the *my_partitions.csv* file a new partition named « my_nvs » (type : data, subtype : nvs) with a size of 1MiB. Do not specify offset and flag.
5. Reference the *my_partitions.csv* file in *sdkconfig* file from the menuconfig command. Refer to section 2.3.1 if necessary.
6. We need to initialize our own partition « my_nvs » and open it. It is necessary to modify the functions of the first 2 steps of the previous lab.
   - Study the following 2 functions : *nvs_flash_init_partition()* and *nvs_open_from_partition()*. What is the difference compared to the functions of the previous lab ?

   - Modify the code in the *app_main()* function. You can declare a const value for the partitino name as below :

     ```
     const char *MY_NVS = "my_nvs";
     ```

7. Run the program.

2.5.3 Statistics for NVS

We want to get statistics on partitions.

1. Add the following code at the end of the function :

```
nvs_stats_t nvsStats;
nvs_get_stats(MY_NVS,&nvsStats);
```

2. Study the fields of the *nvs_stats_t* structure.

3. What is a namespace ? Web help

4. What is the name of the namespace in our code ?

5. Add code to print the information.
6. Run the program and comment on the values.

7. Modify the name of the namespace « my_storage », for example « my_storage_2 ».
8. Run the program and comment on the values.

2.5.4 Exercise - Storing complex structure in the NVS

We will take the Lab Refactoring code of the Working with C and IDF framework in the IoT Framework in order to store in the flash memory the last values of the *SensorSetStruct* structure.

1. Duplicate the « lab3-3-1_c_header_files » (cf. Refactoring code) folder to « lab2-4-3_complex_data_nvs ».
2. Generate the partition table (CSV format) from the built project as we did in a previous lab (cf. section 2.3.1).
3. Declare a new partition named « my_nvs » with a size of 2MiB in the *my_partitions.csv* file and reference it in *sdkconfig* file (use menuconfig).

4. Answer the following questions : :
   - how to use the *nvs_get_blob()* and *nvs_set_blob()* functions ? Web help
   - What is a *key* parameter in these functions ? Web help
   - What is the maximum length of the key parameter ? Web help
5. Use the *app_main()* function from the previous lab (i.e. Customizing the NVS memory partition (Lab2-4-2)) and adapt the *app_main()* function to store the *defaultSensorSet* variable instead of the *value* variable.
6. Run the program and verify.

2.6 SPI Flash File System (Lab2-5)

We studied in a previous lab (cf. Storing files directly in flash memory (Lab2-3)) the possibility of adding files directly to flash memory via variables. We will use another solution to access files through the SPI Flash File System (SPIFFS). This solution can store a full directory into SPI Flash memory.

2.6.1 Flash SPIFFS with code

1. Create the « lab2-5-1_spiffs » lab from « esp32-vscode-project-template » GitHub repository, change memory flash size (4MiB for ESP32-PICO-D4 board) and run Visual Studio Code.
2. Overwrite the « main.c » file by the provided code of the « lab2-5-1_main.c » file.
3. Copy the *spiffs_dir* folder to the *main* folder.
4. Answer the following questions :
   - What does the *spiffs_dir* folder contain ?
   - Study the following 3 functions : *fopen()*, *fgets()*, *fclose()* used in the *readFile()* function. (To help you, put the cursor on the name of the function and press *F12*) and Web help

- Study the fields of the *esp_vfs_spiffs_conf_t* structure. (To help you, put the cursor on the name of the function and press *F12*) and Web help

- Study the *esp_vfs_spiffs_register()*/*esp_vfs_spiffs_unregister()* functions. Web help

- What does the *esp_spiffs_info()* function do ? Web help

- Study the following 3 functions : *opendir()*, *readdir()*, *stat()* used in the *app_main()* function. Web help

5. Study the behavior of the *readFile()* function by adding comments in the code.
6. Study the behavior of the *app_main()* function by adding comments in the code.
7. Build the project and run it. What is the error ?

8. Generate the partition table (CSV format) from the built project as we did in a previous lab (cf. section 2.3.1).

```
esp32:~/../part2_architecture/lab2-5-1_spiffs$ gen_esp32part.py build/
   partition_table/partition-table.bin my_partitions.csv
```

9. Open the *my_partitions.csv* file. and declare at the end of the *my_partitions.csv* file a new partition named « my_storage » (type : data, subtype : spiffs) with a size of 1MiB. Do not specify offset and flag.

10. Reference the *my_partitions.csv* file in *sdkconfig* file from the menuconfig command. Refer to section 2.3.1 if necessary.
11. The easiest is to invoke *spiffsgen.py* (generation of the *spiffs_dir.bin*) file directly from the build system by calling *spiffs_create_partition_image()* function in the *CMake* file located in *main* folder (more information.

```
spiffs_create_partition_image(my_storage ../spiffs_dir
    FLASH_IN_PROJECT)
```

12. Run the program and comment on the console display.

2.6.2 Updating only SPIFFS partition

It is possible to update a spiffs partition without flashing the code. In this section, we will change the content of a file without flashing the program code but just flashing the *my_storage* partition.

1. Change the text of the *note.txt* file located in the `spiffs_dir/doc` folder.
2. The first step is to generate the *spiffs_dir.bin*) file as below :

```
esp32:~/../part2_architecture/lab2-5-1_spiffs$  $IDF_PATH/components/spiffs/
    spiffsgen.py 0x100000 spiffs_dir spiffs_dir.bin
```

3. To ensure that the value of the offset for the *my_storage* partition, launch the command below. What is the offset of *my_storage* partition ?

```
esp32:~/../part2_architecture/lab2-5-1_spiffs$  gen_esp32part.py build/
    partition_table/partition-table.bin
```

4. Now, we can flash the partition as below after replacing the *OFFSET_MY_STORAGE* by the offset of *my_storage* partition :

```
esp32:~/../part2_architecture/lab2-5-1_spiffs$  $IDF_PATH/components/
    esptool_py/esptool/esptool.py --chip esp32 --port /dev/ttyUSB0
    write_flash -z OFFSET_MY_STORAGE spiffs_dir.bin
```

5. Just display now the console without flashing the code !

```
esp32:~/../part2_architecture/lab2-5-1_spiffs$  idf.py monitor
```

6. What do you remark ?

2.6.3 Exercise - SPIFFS partition

1. Append to the end of the *app_ main* function the reading of the *index.html* file.
2. Build and run the program.

2.6.4 Analyze a full example

1. Copy "spiffs" folder located in `$IDF_PATH/examples/storage` to a new lab « lab2-5-2_spiffs-full-example » without opening Visual Studio Code.
2. Copy *.vscode* folder of the previous project (lab2-5-1_spiffs)
3. Open Visual Studio Code.
4. Change flash size to 4MiB with menuconfig.
5. Examine the code.
6. Run the program.

.

LAB N° 3

General Purpose Input/Output (GPIO)

Lab Objectives

- Understanding of GPIO as output and input.

3.1 GPIO Output (Lab3-1)

We want to blink a red LED every second. To do this, we will use the GPIO12 output. The figure 3.1 illustrates the schematic.

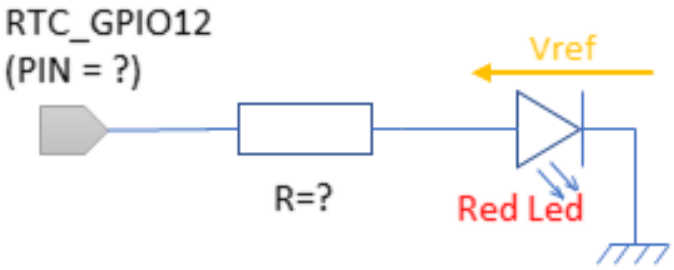


Figure 3.1 – *Schematic for GPIO and Led.*

1. We want a current of around $1mA$. The $V_{ref} \approx 1.9V$ for the Led and the output voltage of the GPIO is $\approx 3.3V$, What is the nearest value of the standardized resistor to use ?

2. What is the pin number of the ESP32-PICO-D4 for RTC_GPIO12 (cf. Appendix Pin description of ESP32-PICO-D4 board.) ?

3. Create the « lab3-1_gpio_output » lab from « esp32-vscode-project-template » GitHub repository, change memory flash size (4MiB for ESP32-PICO-D4 board) and run Visual Studio Code.
4. Overwrite the « main.c » file by the provided code of the « lab3-1_main.c » file.
5. Answer the following questions :
   - Study the following function : *gpio_pad_select_gpio()* from « gpio.h » file in Visual Studio Code.

   - Study the following function : *gpio_set_direction()*. What are the possible modes for the GPIO ? Web help

   - Study the following function : *gpio_set_level()*. Web help

6. Perform the wiring on the board.
7. Run the program.

3.2 GPIO Input (Lab3-2)

We now want to blink the red led every second and if the push button is not pressed. The detection of the pressed button is also each second. If this is pressed, the led remains in the current state and we scan the push button continuously without waiting for a second. To do this, we are going to add a *RTC_GPIO13* input which will make it possible to recover the state of the push button as shown in the figure 3.2. The pullup resistor is directly included in the GPIO pin.

1. Duplicate the « lab3-1_gpio_output » folder to « lab3-2_gpio_input » for the new lab.
2. What is the pin number of the ESP32-PICO-D4 for RTC_GPIO13 (cf. Appendix Pin description of ESP32-PICO-D4 board.) ?

3. Study the following function : *gpio_pullup_en()*. Web help

4. Declare the PIN_PUSH_BUTTON constant.

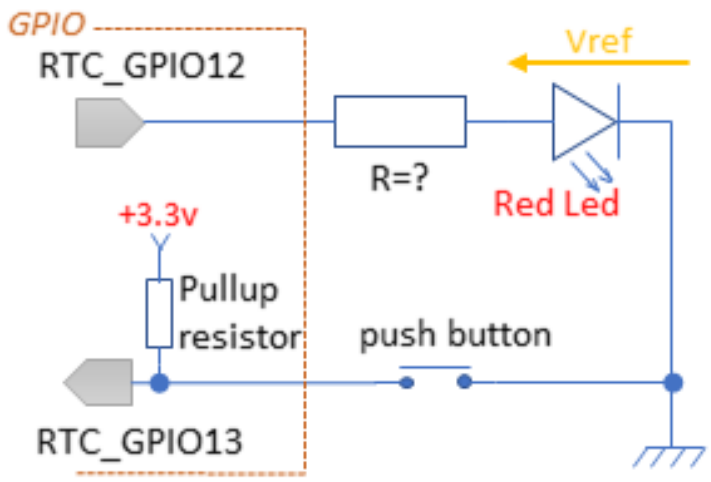


Figure 3.2 – *Schematic for GPIO, Led and push button.*

5. Complete the code in the *app_main()* function.
   - Before the infinite loop for initialization of the GPIO pin.
   - In the loop to take into account the new behavior with push button.
6. Perform the wiring on the board (cf. push button documentation in the `Documentation/push_button_datasheet.pdf`).
7. Run the program.

3.3 Advanced configuration for GPIO (Lab3-3)

We will study a more efficient solution to configure the GPIO by using a *gpio_config_t* structure.

1. Duplicate the « lab3-2_gpio_input » folder to « lab3-3_advanced_gpio_cfg » for the new lab.
2. Study the fields of the *gpio_config_t* structure. Web help

3. Taking the example of the code below, modify the *app_main* function.

```
/* GPIO outputs */
gpio_config_t config_out;
config_out.intr_type = ???
config_out.mode = ???
config_out.pin_bit_mask = (1ULL<<PIN_LED);
gpio_config(&config_out);

/* GPIO inputs */
gpio_config_t config_in;
```

```
config_in.intr_type = ???
config_in.mode = ???
config_in.pull_down_en = ??
config_in.pull_up_en = ??
config_in.pin_bit_mask = ??
gpio_config(&config_in);
```

4. Run the program.

3.4 Exercise - Display state of the push button

We want to display the state of the push button. To do this, we add a yellow led connected to the *RTC_GPIO14* pin.

1. Duplicate the « lab3-3_advanced_gpio_cfg » folder to « lab3-4_exercise_button_state » for the new lab.
2. Adapt the code.
3. Run the program.

LAB N° 4

Interrupt with a General Purpose Input/Output (GPIO)

Lab Objectives

- Polling mode versus interrupt mode.
- Understanding the Watchdog.
- Understanding the principle of an interrupt.
- Programming an interrupt.

4.1 Polling and Interrupt mode

In order to show the advantages of interrupt mode compared to polling mode, we are going to code an application in both modes. We will count the number of times the push button is pressed and use the state of bit 0 of the counter as an indicator for displaying the yellow LED. This led should flash with each press. We use the push button connected to the *RTC_ GPIO13* input pin and the yellow led connected to the *RTC_ GPIO12* output pin as in the lab Advanced configuration for GPIO (Lab3-3).

4.2 Polling mode (Lab4-1)

1. Duplicate the lab Advanced configuration for GPIO (Lab3-3) to « lab4-1_gpio_polling_mode ».
2. Overwrite the « main.c » file by the provided code of the « lab4-1_main.c » file.
3. Complete the code, add comments to better understanding the *app_ main* function.
   - When is the *count* variable incremented ? Rising edge or falling edge of the push button ? Why ?

- What does the *pressNumber* variable represent ?

4. Complete the program helping you
5. Build.
6. What is a watchdog ? (Web help)

7. Run the program.
8. Is there a problem ? Copy the first line of the error.

9. What is a task watchdog ? (Web help)

10. The solution is to disable the Idle0 task watchdog. Change it from menuconfig. (Web help).
11. Open the *sdkconfig* file and check
    the « CONFIG_ESP_TASK_WDT_CHECK_IDLE_TASK_CPU0 » parameters is right.
12. Run the program and press button until *count* is greater than 200. Is there a new problem ? Why ?

13. How to correct the problem ? Modify the code to correct the problem.

14. Run the program.

4.3 Interrupt mode with GPIO (Lab4-2)

1. Duplicate the lab « lab4-1_gpio_polling_mode » to « lab4-2_gpio_interrupt_mode ».

2. Overwrite the « main.c » file by the provided code of the « lab4-2_main.c » file.
3. Answer the following questions :
   - Study the following function : *gpio_install_isr_service()*. Web help
   - Study the following function : *gpio_isr_handler_add()*. Web help
   - What is the *IRAM_ATTR* attribute on the *gpio_switch_isr_handler()* function ? why do we use this attribute ? Web help
   - Why *count* variable is *volatile* ?
   - Why use a *tmpCount* variable ?
4. Open the *sdkconfig* file and check the « CONFIG_ESP_TASK_WDT_CHECK_IDLE_TASK_CPU0 » parameters is always right.
5. Run the program and press button until *count* is greater than 200. Is there problems ? Why ?
   - First problem.
   - Second problem.
6. To solve one of the problem, we add a delay (100ms) at the beginning of the *gpio_switch_isr_handler()* function. We also disable and enable interrupt to avoid re-entering in the interrupt function when it is not finished.

```
// disable the interrupt
gpio_intr_disable(PIN_PUSH_BUTTON);
// Waiting for signal is 1 again
do {
```

```
    ets_delay_us(1000);
} while(gpio_get_level(PIN_PUSH_BUTTON) == 1);

// Previous code
int pinLedNumber = (int)args;
count++;
gpio_set_level(pinLedNumber, (count % 2));

// re-enable the interrupt
gpio_intr_enable(PIN_PUSH_BUTTON);
```

7. Run the program. Now there is only one problem.
8. The solution is to disable the interrupt watchdog. Change it from menuconfig. (Web help).
9. Open the *sdkconfig* file and check
   the « CONFIG_ESP_INT_WDT » parameters is right.
10. Run the program. No more errors.
11. It is possible to use a *sdkconfig.defaults* in the lab Default configuration as below.
12. Copy the provided « sdkconfig.defaults » file to the project.

```
# ESP32 with 4MB flash size
CONFIG_ESPTOOLPY_FLASHSIZE_4MB=y
CONFIG_ESPTOOLPY_FLASHSIZE="4MB"

# Disabled watchdog for interrupt and task
CONFIG_ESP_TASK_WDT_CHECK_IDLE_TASK_CPU0=n
CONFIG_ESP_INT_WDT=n
```

13. Delete the *sdkconfig* file (old config).
14. Build and run the program. The new *sdkconfig* file is generated taking into account default parameters.

Part III

FreeRTOS

LAB N° 1

Task & Scheduling

Lab Objectives

- Creating, suspending and deleting a task.
- Understanding the scheduling of tasks and priority effect.
- Multi-cores scheduling
- Using Idle task.
- Task handler

1.1 Task scheduling on one core (Lab1-1)

The entry point of an application is the *app_main()* function. This function is actually a task of priority 1. We will usually create the other application tasks from the *app_main()* task. We will create an application with 2 tasks (including *app_main()* task), then 3 tasks running on 1 core in order to understand the behavior of the scheduler.

1.1.1 Preparation

Answer the following questions :

- Study the parameters of the following function : *xTaskCreate()* and *vTaskDelete()*. Web help

- Study the following function : *uxTaskPriorityGet()*. Web help

- Study the following function : *vTaskDelay()*. Web help

- What do they do the following macros in the provided « my_helper_fct.h » file ? *DISPLAY()* and *DISPLAYB()*.

- *Task 1* is an instance of the *vTaskFunction()* in the provided code of the « lab1-1_main.c » file. Is that true ?

- Open the provided « lab1-1_sdkconfig.defaults » file. What does the *CONFIG_FREERTOS_UNICORE* parameter correspond to ? Web help

- What is the tick period in ms ? see the *CONFIG_FREERTOS_HZ* parameter and Web help

1.1.2 Understanding the task scheduling

1. Create the « lab1-1_1_core_sched » lab from « esp32-vscode-project-template » GitHub repository.
2. Overwrite the « main.c » file by the provided code of the « lab1-1_main.c » file.
3. Copy the provided « my_helper_fct.h » file to the « main » folder.
4. Copy the provided « lab1-1_sdkconfig.defaults » file to the project folder and rename « sdkconfig.defaults ».
5. Build and run the program.
6. Trace in the figure 1.1 the behavior of the *app_main()* and *Task 1* tasks until 160 ticks.
7. Why the *app_main()* task does not run ?

8. Create a new instance *Task 2* of the *vTaskFunction()* with the same priority of *Task 1*, i.e. priority=5.

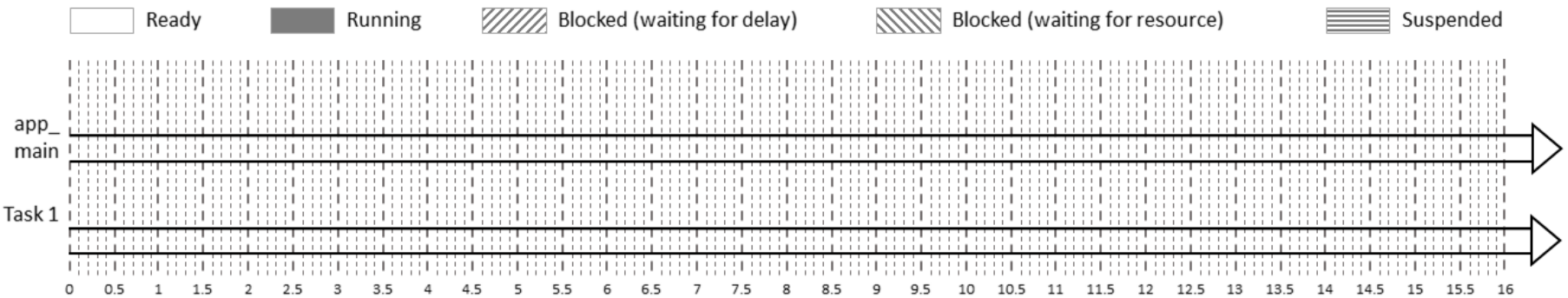


Figure 1.1 – *Scheduling of 2 tasks.*

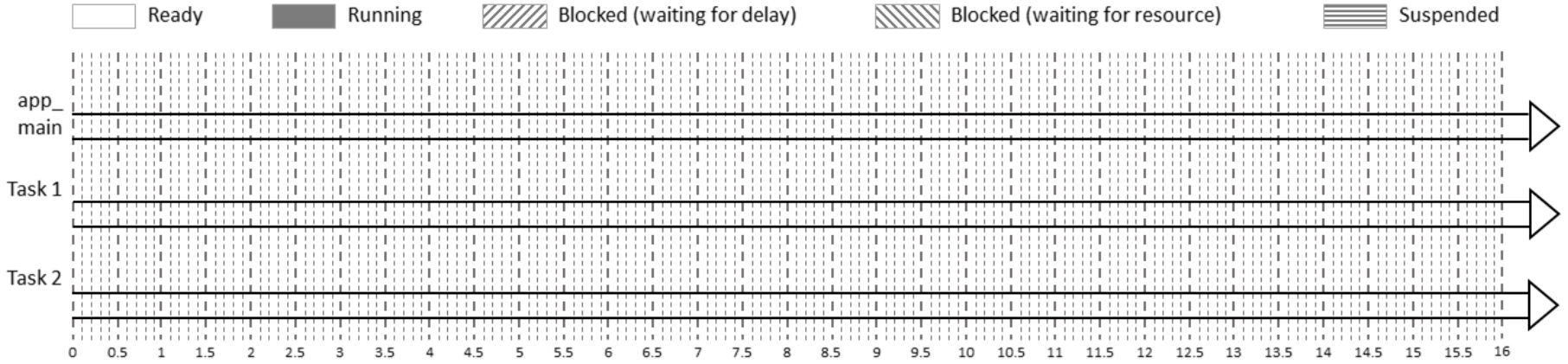


Figure 1.2 – *Scheduling of 3 tasks.*

9. Run the program. Comment on the behavior.

10. We are now going to suspend the scheduler with 2 functions when creating the tasks.

```
void app_main(void) {
  ...
  vTaskSuspendAll();
  // Task creation ...
  ...
  xTaskResumeAll();
  ...
```

11. Run the program.
12. For this scenario, trace in the figure 1.2 the behavior of the *app_main()*, *Task 1* and *Task 2* tasks until 160 ticks.
13. Comment on the behavior. Why the *app_main()* task does not run ?

14. Change the priority=6 for the Task 2. Run the program and comment on the behavior for this scenario.

15. We are now going to add delay on *vTaskFunction()* after the simulation of computation time.

```
void vTaskFunction(void *pvParameters) {
...
for (;; ) {
  DISPLAY("Run computation of %s", pcTaskName);

  /* Delay for simulating a computation */
  for (ul = 0; ul < mainDELAY_LOOP_COUNT; ul++){
  }

  // Add Delay of 300 ms
  DISPLAY("Delay of %s", pcTaskName);
  vTaskDelay(300 / portTICK_PERIOD_MS);
  ...
```

16. Trace in the figure 1.3 the behavior of the *app_main()*, *Task 1* and *Task 2* tasks until 160 ticks.

1.1.3 Using trace information

It is possible to get information about the tasks of the application using the *vTaskList()* function (Web help).

1. Copy the code provided in the « lab1-1_add_vtasklist.c » file to print trace information as below. Note that size of the buffer is approximately 40 bytes per task.

```
/* Buffer to extract trace information */
static char buffer[40*8];

void app_main(void) {
  ...
  /* Print task list */
  vTaskList(buffer);
  printf("---------------------------------------------------------\n"
      );
  printf("task\t\tstate\tprio\tstack\ttsk id\tcore id\n");
  printf("---------------------------------------------------------\n"
      );
  printf(buffer);

  DISPLAY("==== Exit APP_MAIN ====");
  ...
```

2. The 2 first parameters are compulsory to use the *vTaskList()* function. The last parameter is used to print the code id. We active the 3 parameters. 2 solutions :
   - Using the menuconfig command and go to *Component config/FreeRTOS/Enable FreeRTOS trace facility, Enable FreeRTOS stats formatting functions ans Enable display of xCoreID in vTaskList.*
   - Add these lines below in the « sdkconfig.defaults » file (provided in *lab1-1_config_trace_information.txt* file)and delete the « sdkconfig » to take into account the modifications.

```
# Trace facility to extract trace information
CONFIG_FREERTOS_USE_TRACE_FACILITY=y
CONFIG_FREERTOS_USE_STATS_FORMATTING_FUNCTIONS=y
CONFIG_FREERTOS_VTASKLIST_INCLUDE_COREID=y
```

3. Run the program. Copy the trace information. The *stack* column is the amount of free stack space in bytes there has been since the task was created.

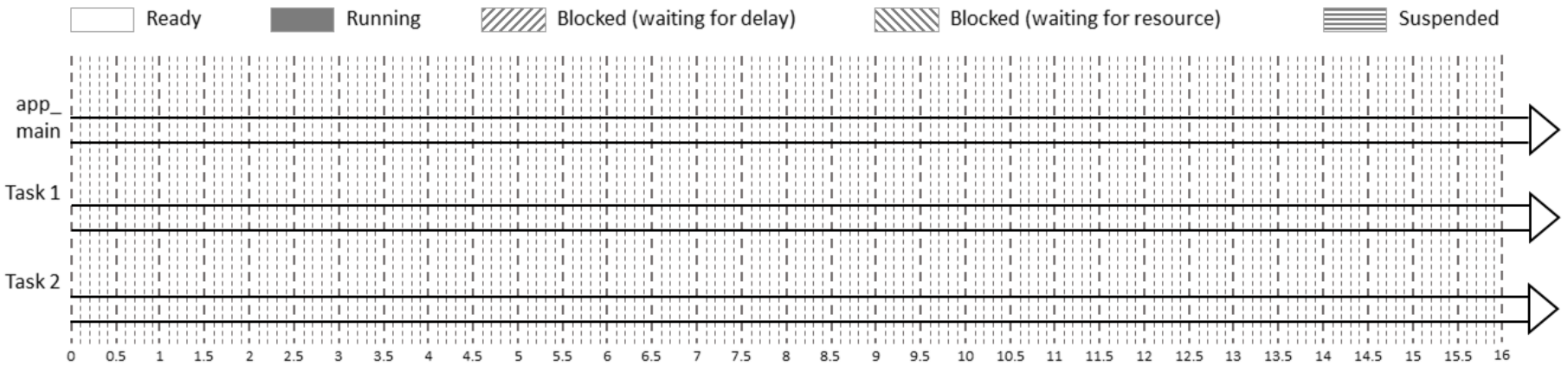


Figure 1.3 – *Scheduling of 3 tasks with task delay.*

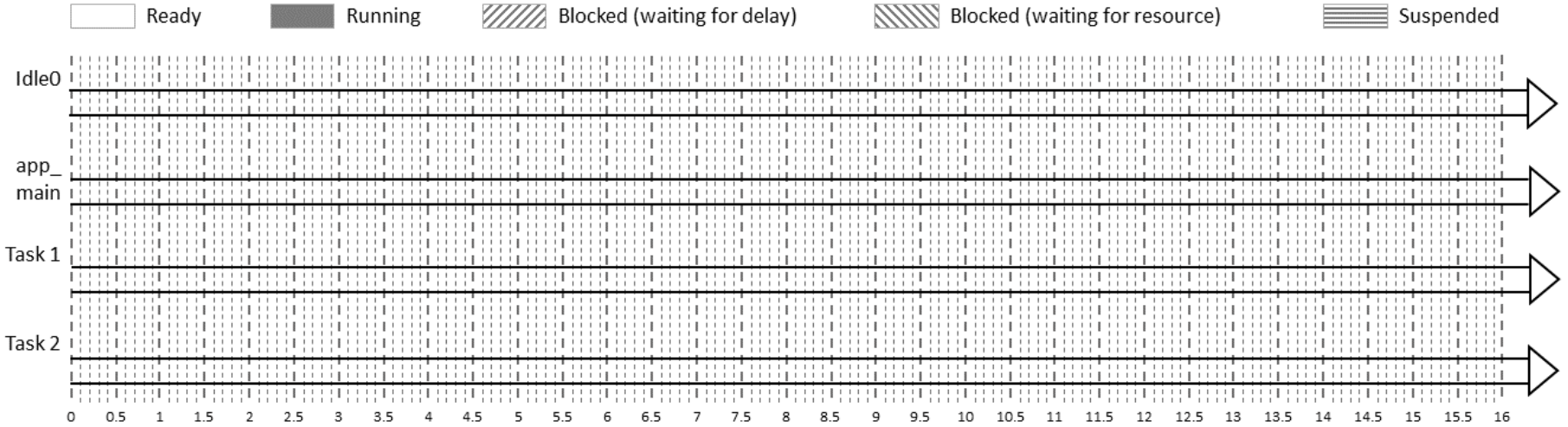


Figure 1.4 – *Scheduling of 3 tasks with task delay.*

4. How many tasks are there ?

5. What is the highest priority task ?

6. What does the *main* task correspond to ?

7. What is the role of the *ipc0* task ? Web help

8. What is the role of the *esp_timer* task ? Web help

9. What is the role of the *Tmr Svc* task ? Web help 1 and Web help 2

10. What is the role of the *IDLE0* task ? What is its priority ? Web help

11. The default stack size is 4000 bytes. Change the constant *STACK_SIZE*=1400 bytes. What is the problem ? Change the size to the nearest hundredth to avoid the problem.

1.1.4 Idle and Tick Task Hooks

An idle task hook is a function that is called during each cycle of the idle task. To use them, we will modify the configuration.

1. Add the two hook functions provided in the « lab1-1_add_idle_hook.c » file.

```
bool vApplicationIdleHook ( void ){
  DISPLAY ("IDLE");
  return true;
}
```

```
void vApplicationTickHook ( void ){
}
```

2. Why are there 2 hook functions ? What is the difference ? Web help 1 and Web help 2

3. The first parameter is compulsory to use the hooks functions. The second parameter adjusts the stack size depending on the behavior of the hook functions. 2 solutions :
   - Using the menuconfig command and go to *Component config/FreeRTOS/Use FreeRTOS legacy hooks*.
   - Add these lines below in the « sdkconfig.defaults » file (provided in *lab1-1_config_idle_hook.txt* file) and delete the « sdkconfig » to take into account the modifications.

     ```
     # allows add hook functions
     CONFIG_FREERTOS_LEGACY_HOOKS=y
     CONFIG_FREERTOS_IDLE_TASK_STACKSIZE=4096
     ```

4. Run the program. What do you see for the *Idle* and *app_main* tasks ?

5. Modify the task delay of vTaskFunction() function to 400ms instead of 100ms. What do you see for *Idle* and *app_main* tasks ? Conclusion ?

6. Trace in the figure 1.4 the behavior of tasks until 160 ticks.

1.2 Task scheduling on two cores (Lab1-2)

the ESP32 has 1 or 2 cores. It is possible to choose on which core the task should be mapped or to let the scheduler choose. At the end of this Lab, we will see another solution to use the *idle* task without using the configuration but an API.

1. Create the « lab1-2_2_cores_sched » lab from « esp32-vscode-project-template » GitHub repository.
2. Overwrite the « main.c » file by the provided code of the « lab1-2_main.c » file.
3. Copy the provided « my_helper_fct.h » file to the « main » folder.
4. Copy the provided « lab1-2_sdkconfig.defaults » file to the project folder and rename « sdkconfig.defaults ».

5. Study the following function : *xTaskCreatePinnedToCore()*. Web help

1.2.1 Task scheduling scenarios

We are going to test different scheduling scenarios according to the cores on which the tasks will be executed. In order to facilitate the different scenarios to be executed, we will use the preprocessor macros as below :

```
//#define DIFF_PRIORITY

#define PINNED_TO_CORE 0x00
//    0x00: Task 1: Core 0, Task 2: Core 0
//    0x01: Task 1: Core 0, Task 2: Core 1
//    0x10: Task 1: Core 1, Task 2: Core 0
//    0x11: Task 1: Core 1, Task 2: Core 1

//#define IDLE_HOOKS

//#define TASK_DELAY
```

You must run the 4 scenarios, copy the trace till 160 ticks (1600 ms) and comment it.

- Uncomment *DIFF_PRIORITY* to have the priority(Task 1) = 5 and priority(Task 2) = 6.
- Scenario nº 1 : Task 1 : Core 0, Task 2 : Core 0

- Scenario nº 2 : Task 1 : Core 0, Task 2 : Core 1

- Scenario nº 3 : Task 1 : Core 1, Task 2 : Core 0

- Scenario nº 4 : Task 1 : Core 1, Task 2 : Core 1.

For the last scenario nº 5, we are going to add a task on *core 0* using task mapping of scenario nº 4.

1. Add a *Task 3* on core 0 with priority = 1.
2. Trace in the figure 1.5 the behavior of the tasks until 160 ticks.

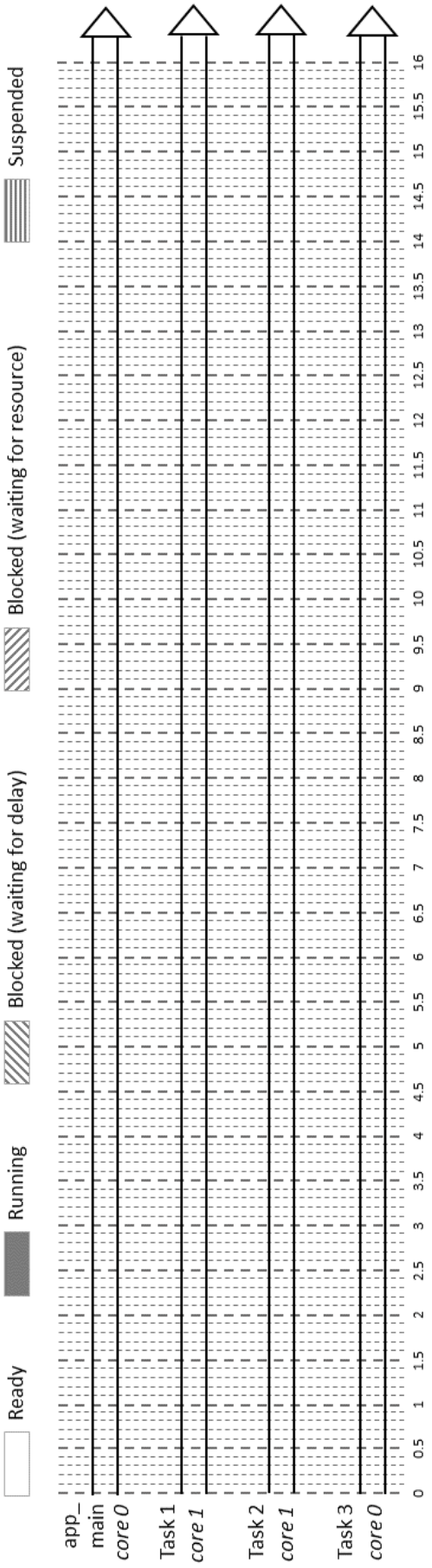


Figure 1.5 – *Scheduling on 2 cores with 4 tasks.*

1.2.2 Idle Task Hook from API (Optional work)

The disadvantage of configuring the hook functions from the configuration menu is the declaration of the 2 functions while generally the *vApplicationIdleHook()* function is the only useful function. Thus, it is possible to use an API which allows adding only the *vApplicationIdleHook()* function.

1. Uncommment the $IDLE\_HOOKS$ preprocessor macro. We use the previous scenario n° 5.
2. Study the following function : *esp_register_freertos_idle_hook_for_cpu()*. Web help
3. Run the program and comment the behavior. What do you notice about the execution of the *IDLE1* task ?
4. Uncommment the $TASK\_DELAY$ preprocessor macro.
5. Run the program and comment the new behavior, in particular the *IDLE1* task.

1.3 Approximated/Exactly periodic task (Lab1-3)

In some applications it is important to have tasks with an exact period. We will compare the use of the *TaskDelay()* function which allows you to wait for a certain time and the *TaskDelayUntil()* function which allows you to adjust the delay according to the time spent.

1. Duplicate the « lab1-2_2_cores_sched » folder to « lab1-3_periodic_task ».
2. Check that you have uncommented the $DIFF\_PRIORITY$, $IDLE\_HOOKS$ and $TASK\_DELAY$ preprocessor macros.
3. Set the $PINNED\_TO\_CORE$ macro to $0x00$ to run all tasks on PRO_CPU (Core 0).
4. Change the behavior of the *vTaskFunction()* task as below.

```
#ifdef TASK_DELAY
  /* Approximated/Periodic Delay */
  #ifdef PERIODIC_TASK_DELAY
    DISPLAY("Periodic Delay of %s", pcTaskName);
    vTaskDelayUntil(&xLastWakeTime, pdMS_TO_TICKS(300));
  #else
    DISPLAY("Approximated Delay of %s", pcTaskName);
    vTaskDelay(pdMS_TO_TICKS(300));
  #endif
#endif
```

5. What does it do the following macro *pdMS_TO_TICKS()* ?

6. Study the *vTaskDelayUntil()* function. Web help

7. Run the program and comment the behavior. What is the interval/period of Tasks 1,2 and 3 ? To compute time, use the tick number from the display of « Run computation of Task X » (X = 1,2 or 3) comment.

8. Uncomment the $PERIODIC\_TASK\_DELAY$ preprocessor macro.
9. Build the project and correct the compilation error if necessary.
10. Run the program at least up to 400 ticks and comment the behavior. What is the interval/period of Tasks 1,2 and 3 ? What is the problem ?

11. Change the *mainDELAY_LOOP_COUNT* constant to *0x1FFFF*. What is the interval/period of Tasks 1,2 and 3 ? Comment the new behavior.

1.4 Task handler and dynamic priority (Lab1-4, Optional work)

During the execution of the application, it is sometimes useful to modify the priority of a task or any other actions. To do this, it is necessary to get the handler of the task on which we have to operate these actions.

1. Create the « lab1-4_task_handler_priority » lab from « esp32-vscode-project-template » GitHub repository.
2. Overwrite the « main.c » file by the provided code of the « lab1-4_main.c » file.

3. Copy the provided « my_helper_fct.h » file to the « main » folder.
4. Copy the provided « lab1-4_sdkconfig.defaults » file to the project folder and rename « sdkconfig.defaults ».
5. Study the *uxTaskPriorityGet()* and *vTaskPrioritySet()* functions. Web help

6. Study the *vTaskGetRunTimeStats()* function. What parameters should be configured in *sdkconfig* file or from graphical menu config ? Web help

7. Explain the use of the handler. In what tasks ? Why ?

8. Run the program. What is the problem ?

9. How are you going to correct the problem ?

10. Run the program after code modification and comment the scenarios.

11. Comment also the statistics.

LAB N° 2

Message Queue & Interrupt service

Lab Objectives

- Using message queue API.
- Using message queue with interrupts

2.1 Single Message Queue (Lab2-1)

We want to create 3 tasks. The functions implementing the tasks will be named *Task1()* (priority=2, periodic : 500ms, running on the *Core 1*), *Task2()* (priority=2, , running on the *Core 0*) and *Task3()* (priority=3, , running on the *Core 0*) respectively. *Task1()* will send a number (using a message queue object) to the *Task2()*. Each task displays string to the terminal with *DISPLAY()* or *DISPLAYI()* macros.

1. Create the « lab2-1_single_msg_queue » lab from « esp32-vscode-project-template » GitHub repository.
2. Copy the provided « my_helper_fct.h » file to the « main » folder.
3. Copy the provided « lab2-1_sdkconfig.defaults » file to the project folder and rename « sdkconfig.defaults ».
4. Append the content of *add_includes.c* file to the start of the *main.c* file.
5. Study the following function : *xQueueSend()*, *xQueueReceive()*. Web help 1, Web help 2

6. Write the 3 tasks with empty body.

7. Declare the 3 tasks on the *app_main()* function (Best practice : use constants for priorities, stack size, ...).
8. Add the message queue (depth is 5) in the program. The type of the message is *uint32_t*.
9. Write the behavior of the tasks as below :
   - Task 3 blocks for 100ms, displays string to the terminal and then simulates a computation of 20ms (2 ticks).

     ```
     // Task blocked during 100 ms
     DISPLAY(...);
     vTaskDelay(pdMS_TO_TICKS(100));
     DISPLAY(...);
     // Compute time : 2 ticks or 20 ms
     COMPUTE_IN_TICK(2);
     ```

   - The *Task 1* should be periodic with a periodicity of 500ms. It posts in a message queue to the *Task 2*, check the result of the post (Failed or posted message), simulates a computation of 40ms (4 ticks) and wait for next period. Note that the write function in the queue should not be blocking.

     ```
     // Post
     uint32_t result = xQueueSend(...);
     // Check result
     if (result) ...
     // Compute time : 4 ticks or 40 ms
     COMPUTE_IN_TICK(4);
     // block periodically : 500ms
     vTaskDelayUntil(...);
     ```

   - Task 2 waits for a message through the message queue, displays the task number and message received and then simulates a computation of 30ms (3 ticks).

     ```
     // Wait for message
     ...
     // display task number and message
     DISPLAY(...);
     // Compute time : 3 ticks or 30 ms
     COMPUTE_IN_TICK(3);
     ```

   - Don't forget to create a message queue.
10. Build and run the program.
11. Trace in the figure 2.1 the behavior of the 3 tasks until 160 ticks.
12. What is the period of the task 2 ?

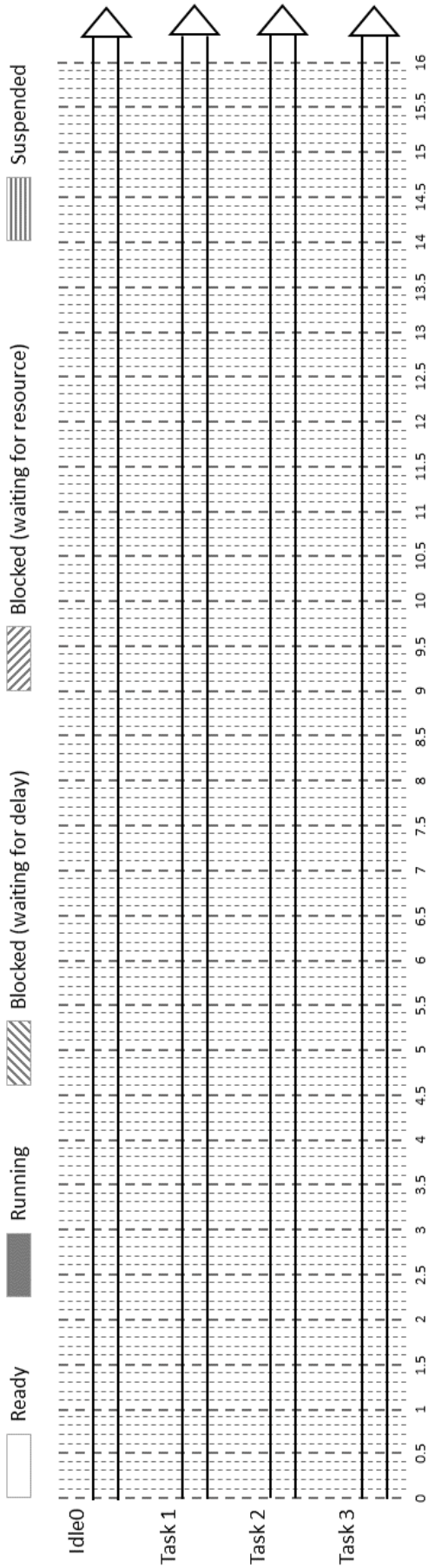


Figure 2.1 – *Message queue and 3 tasks.*

2.2 Message Queue with time out (Lab2-2)

The concept of timeout only applies to blocking system calls. When a task is blocked, it will wake up (go to the ready state) automatically after a period of time called Timeout, even if the expected event has not happened.

- Duplicate the « lab2-1_single_msg_queue » folder to « lab2-2_single_msg_queue_timeout ».
- Force *Task 2* to wake up every 300ms using the Timeout associated with the *xQueueReceive()* function as below.

```
static const char* TAG = "MsgTimeOut";
...
if (xQueueReceive(...)) {
  DISPLAYI(TAG, "Task 2, mess = %d", value);
  COMPUTE_IN_TICK(3);
}
else {
  DISPLAYE(TAG, "Task 2, Timeout!");
  COMPUTE_IN_TICK(1);
}
```

- Build and run the program.
- Trace in the figure 2.3 the behavior of the 3 tasks until 160 ticks.

2.3 Blocking on single Queue (Lab2-3, Optional work)

We want to illustrate the problems of writing/reading queue messages and the impact on the scheduling of tasks. The figure 2.2 illustrates the application. All tasks run on the *Core 0*. The message queue contains items of integer type.

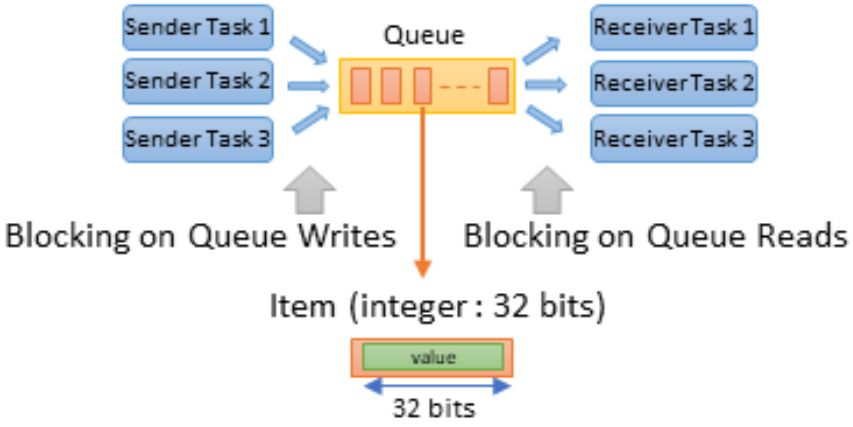


Figure 2.2 – *Blocking on single queue.*

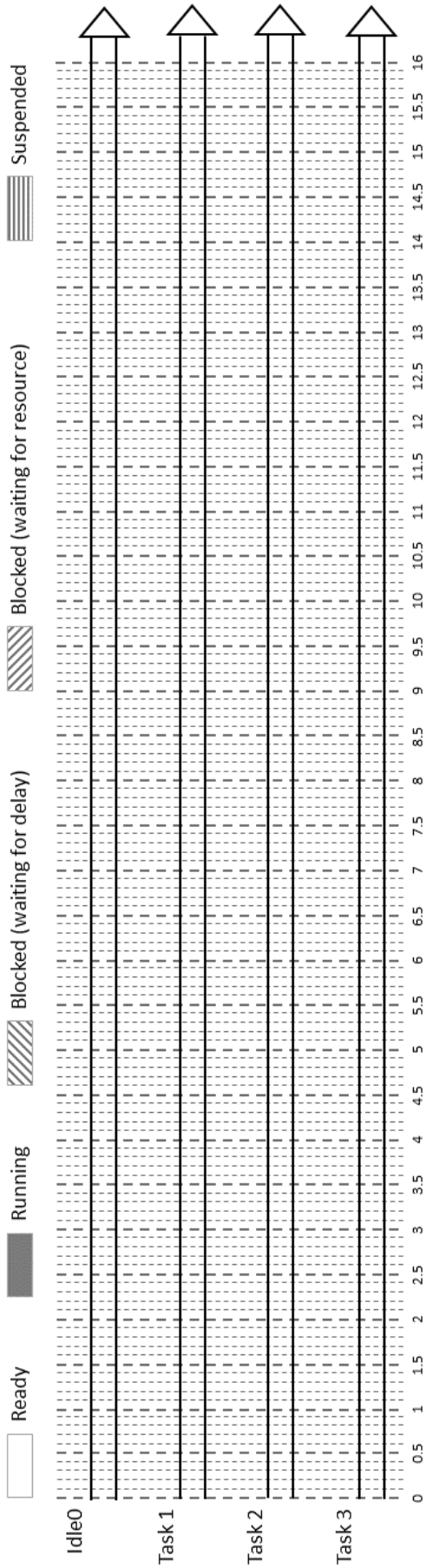


FIGURE 2.3 – *Message queue with timeout and 3 tasks.*

2.3.1 Blocking on Queue Reads

1. Create the « lab2-3_single_msg_queue_blocked » lab from « esp32-vscode-project-template » GitHub repository.
2. Copy the provided « my_helper_fct.h » file to the « main » folder.
3. Copy the provided « lab2-3_sdkconfig.defaults » file to the project folder and rename « sdkconfig.defaults ».
4. Overwrite the « main.c » file by the provided code of the « lab2-1_main.c » file.
5. Complete the code. For the moment, initialize the constants as below :

```
const uint32_t SENDER_TASK_PRIORITY = 3;
const uint32_t RECEIVER_TASK_PRIORITY = 3;
const uint32_t MESS_QUEUE_MAX_LENGTH = 10;
const uint32_t SENDER_TASK_NUMBER = 3;
const uint32_t RECEIVER_TASK_NUMBER = 3;
```

6. Build and run the program.
7. Explain the behavior.

8. Modify the priority of *receiver task* to 5 and run the program.
9. Explain the new behavior.

10. Modify the priority of *sender task* to 5 and the priority of *receiver task* to 3. Run the program.
11. Explain the new behavior.

2.3.2 Blocking on Queue Writes

1. Modify the priority of *sender task* to 5 and the priority of *receiver task* to 3.
2. Modify the capacity of the message queue to 2 (i.e. $MESS\_QUEUE\_MAX\_LENGTH = 2$)

3. Explain the problem.

4. How to correct the problem by adjusting the priority of tasks ?

2.4 Using message queue with interrupts (Lab2-4)

The figure 2.4 illustrates the application we want to achieve. When we press the push button, it will trigger on falling edge an interrupt (*Push_button_isr_handler()*) that will send a message containing the GPIO pin number of push button to the *vCounterTask()* task. If the button is not pressed after 5 seconds, a message is displayed indicating that the button must be pressed to trigger an interrupt. A simple way is to use the timeout of the message queue received function. All tasks run on the *Core 0*. We will take over the lab Interrupt mode with GPIO (Lab4-2) to configure GPIO and to install interrupt service.

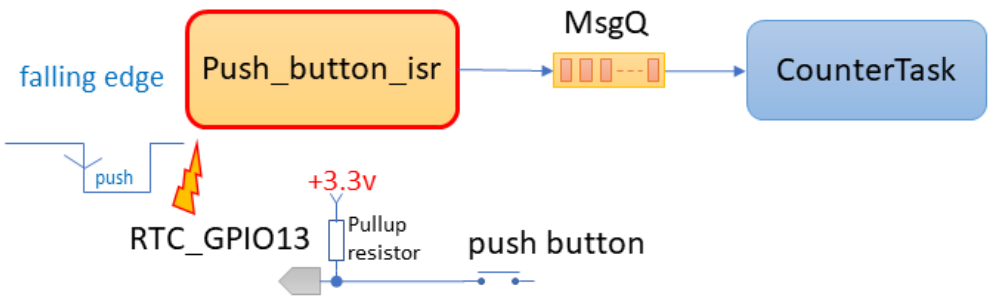


Figure 2.4 – *Interrupt application with message queue.*

1. Create the « lab2-4_single_msg_queue_interrupt » lab from « esp32-vscode-project-template » GitHub repository.
2. Overwrite the « main.c » file by the provided code of the « lab2-4_main.c » file.
3. Copy the provided « my_helper_fct.h » file to the « main » folder.
4. Copy the provided « lab2-4_sdkconfig.defaults » file to the project folder and rename « sdkconfig.defaults ».
5. Study the following function : *xQueueSendFromISR()*. Web help

6. Study the following function : *uxQueueMessagesWaiting()*. Web help

7. Configure the GPIO for the push button (RTC_GPIO13).
8. Write the creation of the message queue (item size = 5) and the *vCounterTask()* task in the *app_main()* function.
9. Complete the body of the *Push_button_isr_handler()* by following the comments. Declare an *isrCount* global variable to count each trigger of interrupt. The argument of the *Push_button_isr_handler()* interrupt is the GPIO pin number of push button.
10. Write the installation of interrupt service in the *app_main()* function.
11. Write the body of *vCounterTask()* task by following the comments in the source code.
12. Perform the wiring on the board. Refer to the documentation *ESP32-PICO-D4_Pin_Layout.pdf*.
13. Build and run the program.
14. When is the message "number of items" displayed ? why ?

15. Explain how to correct the problem ? more than one solution is possible.

16. Change the program according to your solution. Build and run the program.

LAB N° 3

Semaphore & Mutex

Lab Objectives

- Using semaphore API.
- Using Mutex

3.1 Specification of the application

We want to implement the functional structure (cf. figure 3.1) using FreeRTOS.

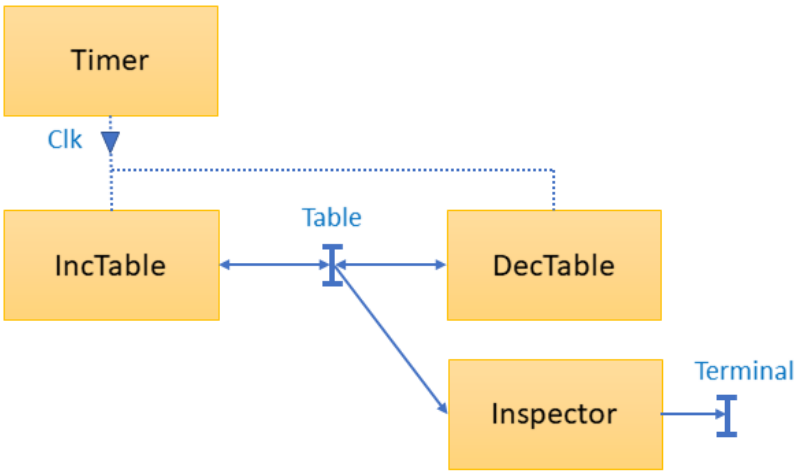


Figure 3.1 – *Functional description.*

The behavioral description of tasks is presented below in algorithm form. *Table* is an array of integer values that can represent a signal, such as a ramp. Its size is a constant (TABLE_SIZE) which can be modified during the tests. During initialization, it takes the

values [0,1,2,..., TABLE_SIZE-1]. Use TABLE_SIZE = 400.
Do not forget to use the *pdMS_ TO_ TICKS(iTime time)* macro to convert time in millisecond to tick number.

3.1.1 Timer task

This is a task that performs a *TaskDelay()* and generates the synchronization (semaphore) via *Clk*. The function *computeTime()* simulates a execution time in millisecond.

Task *Timer* **is**
Properties: Priority = 5
Out **:** Clk is event
Cycle :
waitForPeriod(250 ms);
computeTime(20 ms);
print("Task Timer : give sem");
notify(Clk);
end
end

Algorithm 1: Timer algorithm.

3.1.2 IncTable task

It is a temporary action activated by *Clk*. An *constNumber* is passed by the parameters of the task. Every 5 activations, the task increments by the *constNumber* for each element of *Table*. We simulate a computation time by the *computeTime()* pseudo function. Its functional behavior is as follows :

```
Task IncTable is
    Properties: Priority = 4
    In          : Clk is event
    In/Out      : Table is array[0 to TABLE_SIZE-1] of integer
    ActivationNumber := 0;
    Cycle Clk :
        if ActivationNumber = 0 then
            for index :=0 to TABLE_SIZE-1 do
                Table[index] := Table[index] + constNumber;
            end
            computeTime(50 ms);
            ActivationNumber := 4;
        else
            ActivationNumber := ActivationNumber - 1;
        end
    end
end
```

Algorithm 2: IncTable algorithm.

3.1.3 DecTable task

Its functional behavior is as follows :

```
Task DecTable is
    Properties: Priority = 4
    In          : Clk is event
    In/Out      : Table is array[0 to TABLE_SIZE-1] of integer
    Cycle Clk :
        for index :=0 to TABLE_SIZE-1 do
            Table[index] := Table[index] - 1;
        end
        computeTime(50 ms);
    end
end
```

Algorithm 3: DecTable algorithm.

3.1.4 Inspector task

It is a task which constantly checks the consistency of the *Table* and displays an error message when an inconsistency is found in the *Table* (exit on the program, use *exit(1)* function). For this, it takes the first value of the *Table* as a reference (*reference = Table[0]*) and checks each element of *Table* in accordance with its reference (*Table[index] = reference + index*). When the *Table* has been fully browsed, the cycle begins again (a new reference is

taken and the *Table* is checked again). Its functional behavior is as follows :

Task *Inspector* **is**
Properties: Priority = 4
In : Table is array[0 to TABLE_SIZE-1] of integer
ActivationNumber := 0;
Cycle :
print("Task Inspector is checking.");
reference := Table[0];
error := false;
for *index :=1* **to** *TABLE_SIZE-1* **do**
smallComputeTime(100 us);
if $Table[index] \neq (reference + index)$ **then**
error := true;
end
end
print("Task Inspector ended its checking.");
if *error = true* **then**
print(TAG, "Consistency error in the Table variable.");
exit();
end
end
end

Algorithm 4: Inspector algorithm.

3.2 First Task synchronization (Lab3-1)

Firstly, we will only implement the *Timer*, *DecTable* and *IncTable* tasks as well as the *Clk* and *Table* relationships. The *computeTime()* function can be implemented by the *COMPUTE_IN_TICK()* macro (1 tick = 10ms) and the print() function by *DISPLAY()* macro.

3.2.1 Writing the application

1. Create the « lab3-1_one_sem_clk » lab from « esp32-vscode-project-template » GitHub repository.
2. Copy the provided « lab3-1_sdkconfig.defaults » file to the project folder and rename « sdkconfig.defaults ».
3. Copy the provided « my_helper_fct.h » file to the « main » folder.
4. Overwrite the « main.c » file by the provided code of the « lab3-1_main.c » file.
5. Write the program with the behavior of these 3 tasks and 1 semaphore (*xSemClk*) using the algorithms proposed above. All the tasks are created on the *Core_0*. Below is a creation reminder for a semaphore.

```
/* Creating Binary semaphore */
SemaphoreHandle_t xSemClk;
xSemClk = xSemaphoreCreateBinary();
/* Using semaphore */
xSemaphoreGive(xSemClk);
xSemaphoreTake(xSemClk, portMAX_DELAY);
```

6. Build the program without running it.

3.2.2 Scenarios with one clock semaphore

We will perform scenarios described in the Table 3.1 in order to identify problems and improve the program later. The task priority of *Timer* is 5 and run on *Core_ 0*.

Scenario	IncTable task	DecTable task
1	Prio(4),Core(0)	Prio(4),Core(0)
2	Prio(3),Core(0)	Prio(4),Core(0)
3	Prio(4),Core(0)	Prio(4),Core(1)
4	Prio(3),Core(0)	Prio(4),Core(1)

Table 3.1 – *Scenarios for task synchronization.*

Scenario 1 : Run the program, copy the console, trace in the figure 3.2 the behavior of the 3 tasks until 160 ticks and explain the problem.

Scenario 2 : Run the program, copy the console and explain the problem.

Scenario 3 : Run the program, copy the console and explain the problem.

Scenario 4 : Run the program, copy the console and explain the problem.

3.3 Task synchronization with 2 semaphores (Lab3-2)

We identified different issues in the previous section. We will perform again the scenarios described in the Table 3.1.

1. Duplicate the « lab3-1_one_sem_clk » folder to « lab3-2_two_sem_clk ».
2. Correct the clock program to send 2 separate semaphores (*xSemIncTab* and *xSemDecTab*) to *IncTable* and *DecTable* tasks.

Scenario 1 : Run the program, copy the console, trace in the figure 3.3 the behavior of the 3 tasks until 160 ticks and explain the behavior.

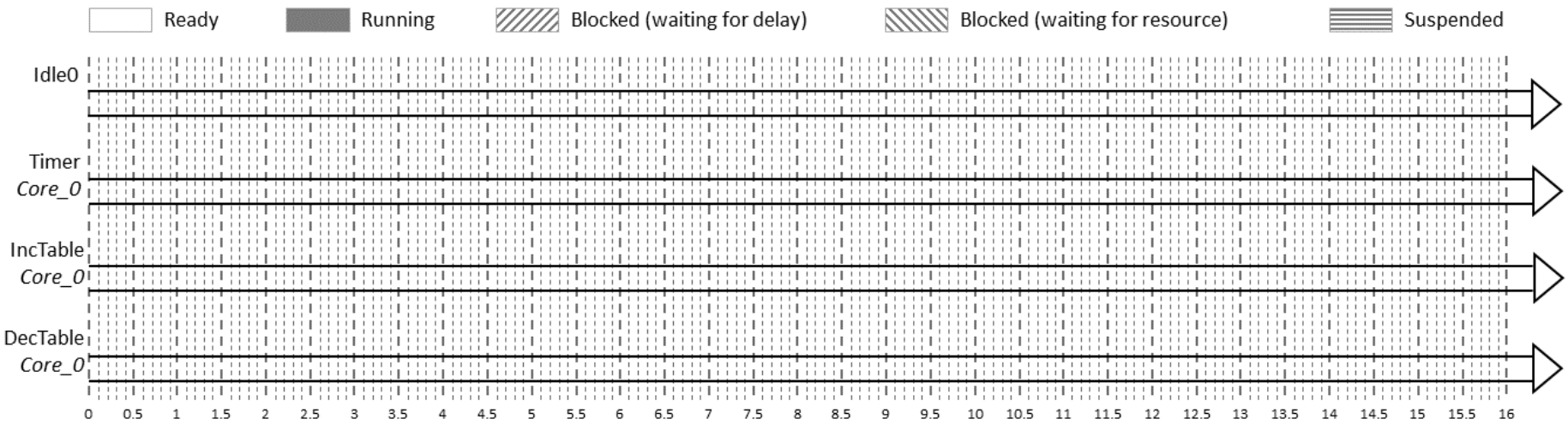


Figure 3.2 – *Scenario 1 : Priority(DecTable=4, IncTable=4), Core(DecTable=0, IncTable=0).*

Scenario 2 : Run the program, copy the console, trace in the figure 3.4 the behavior of the 3 tasks until 160 ticks and explain the behavior.

Scenario 3 : Run the program, copy the console, trace in the figure 3.5 the behavior of the 3 tasks until 160 ticks and explain the behavior.

Scenario 4 : Run the program, copy the console and explain the behavior.

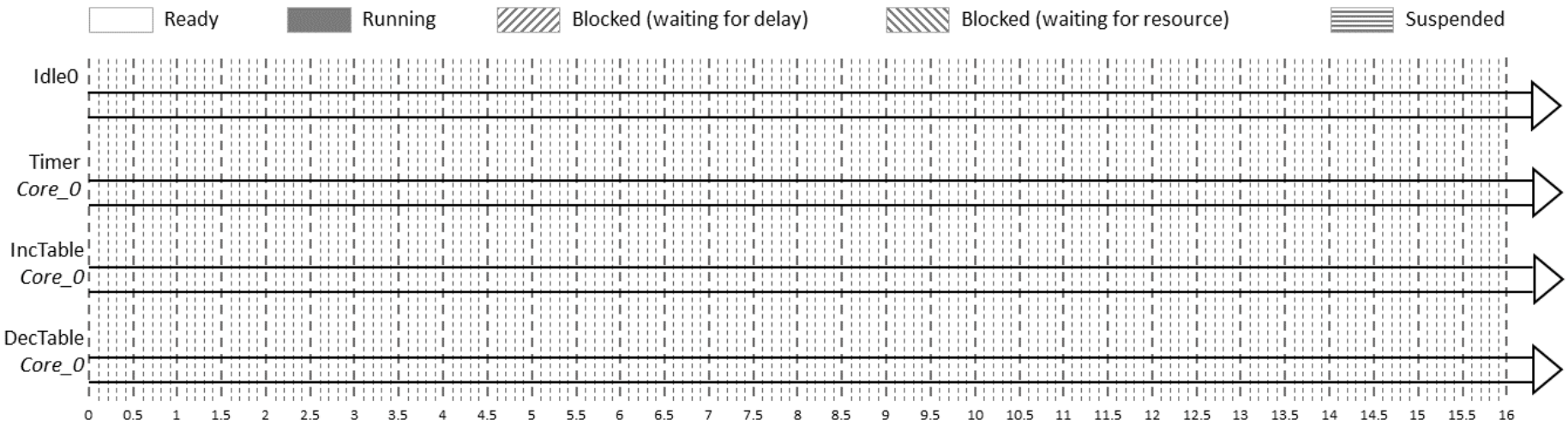


FIGURE 3.3 – *Scenario 1 : Priority(DecTable=4, IncTable=4), Core(DecTable=0, IncTable=0).*

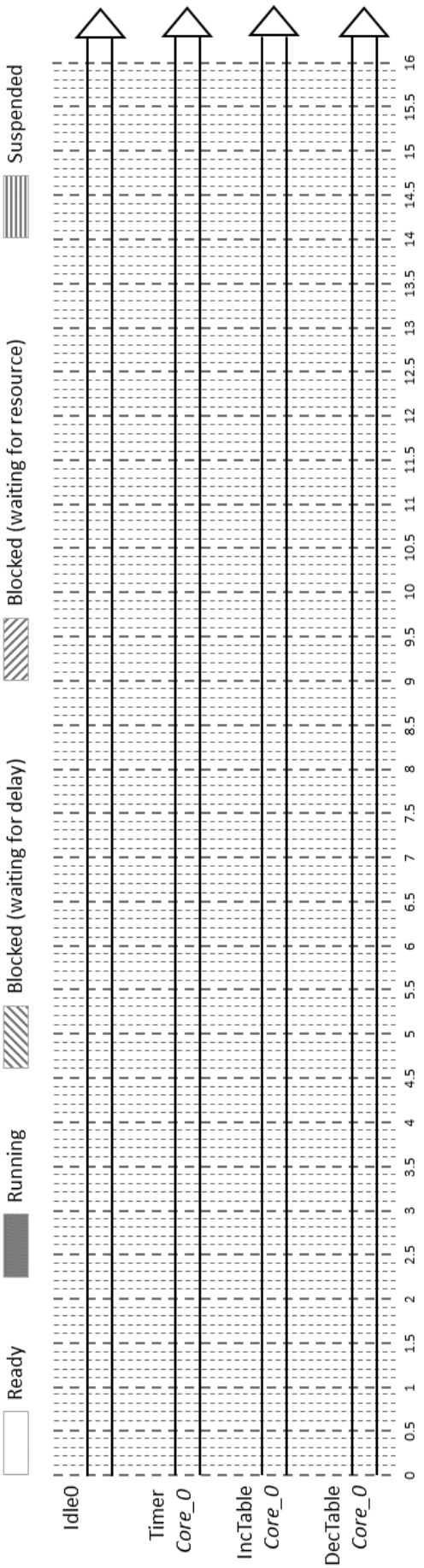


Figure 3.4 – *Scenario 2 : Priority(DecTable=4, IncTable=3), Core(DecTable=0, IncTable=0).*

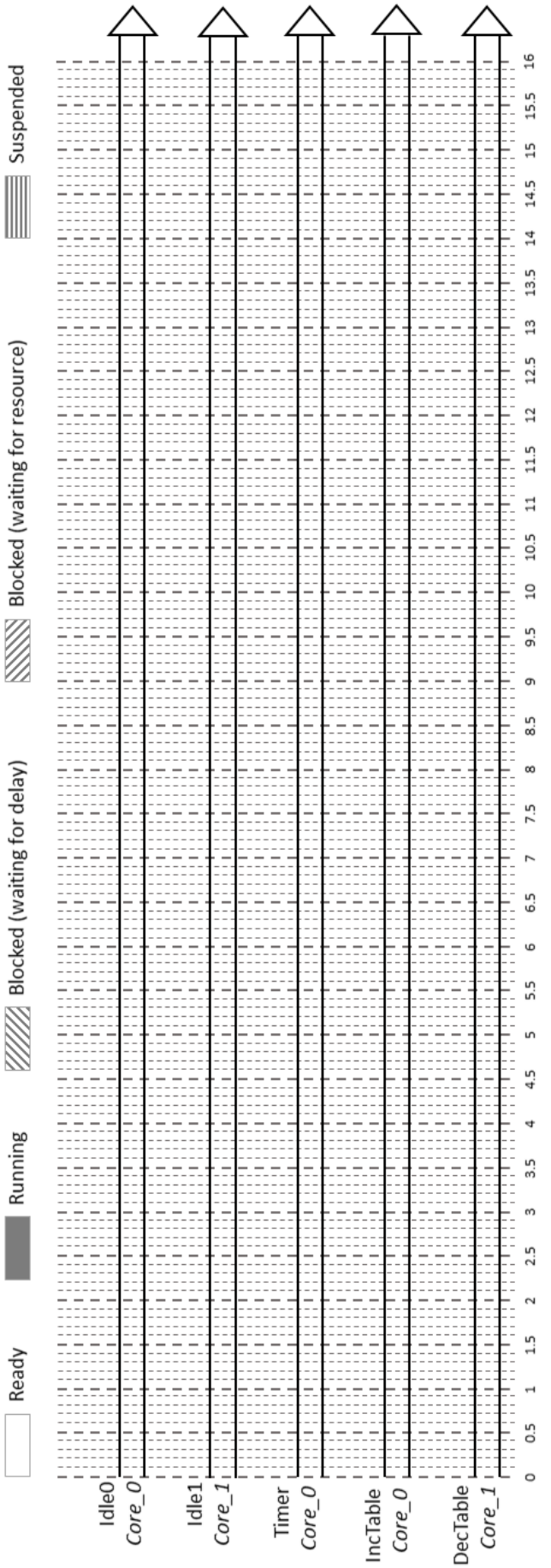


FIGURE 3.5 – *Scenario 3 : Priority(DecTable=4, IncTable=4), Core(DecTable=1, IncTable=0).*

3.4 Mutual Exclusion (Lab3-3)

We now add the *Inspector* task. We will perform the program with the task priorities described in the Table 3.2.

Timer task	IncTable task	DecTable task	Inspector task
Prio(5),Core(0)	Prio(3),Core(0)	Prio(3),Core(0)	Prio(4),Core(0)

Table 3.2 – *Task priorities with mutex.*

1. Duplicate the « lab3-2_two_sem_clk » folder to « lab3-3_mutex ».
2. Add the *Inspector* task.
3. Run the program, copy the console and explain the behavior. What is the problem ?

4. Correct the problem of initialization.
5. Run the program, copy the console and explain the behavior. What is the problem ?

6. Choose a new priority of the *Inspector* task to solve the problem.
7. Run the program, copy the console until 40 ticks, trace in the figure 3.6 the behavior of the 3 tasks until 40 ticks and explain the behavior.

8. Modify the *Inspector* task by adding a *Mutex* to manage access to the critical area. Below is a creation reminder for a Mutex.

```
/* Mutex */
SemaphoreHandle_t xSemMutex;
xSemMutex = xSemaphoreCreateMutex();
/* Using Mutex */
xSemaphoreGive(xSemMutex);
xSemaphoreTake(xSemMutex, portMAX_DELAY);
```

9. Run the program, copy the console, trace in the figure 3.7 the behavior of the 3 tasks until 40 ticks and explain the behavior and the effect of the Mutex.

10. Change the priority of *Inspector* task to 4. Run the program, copy the console until 40 ticks. What is the problem ?

11. We decide to change the *Inspector* task to *Core_ 1*. Run the program, copy the console until 40 ticks. What is the problem ?

12. We now decide to add a delay of 2 ticks after giving the mutex (using *vTaskDelay()* function). Run the program, copy the console, trace in the figure 3.8 the behavior of the 3 tasks and Mutex until 90 ticks and explain the behavior.

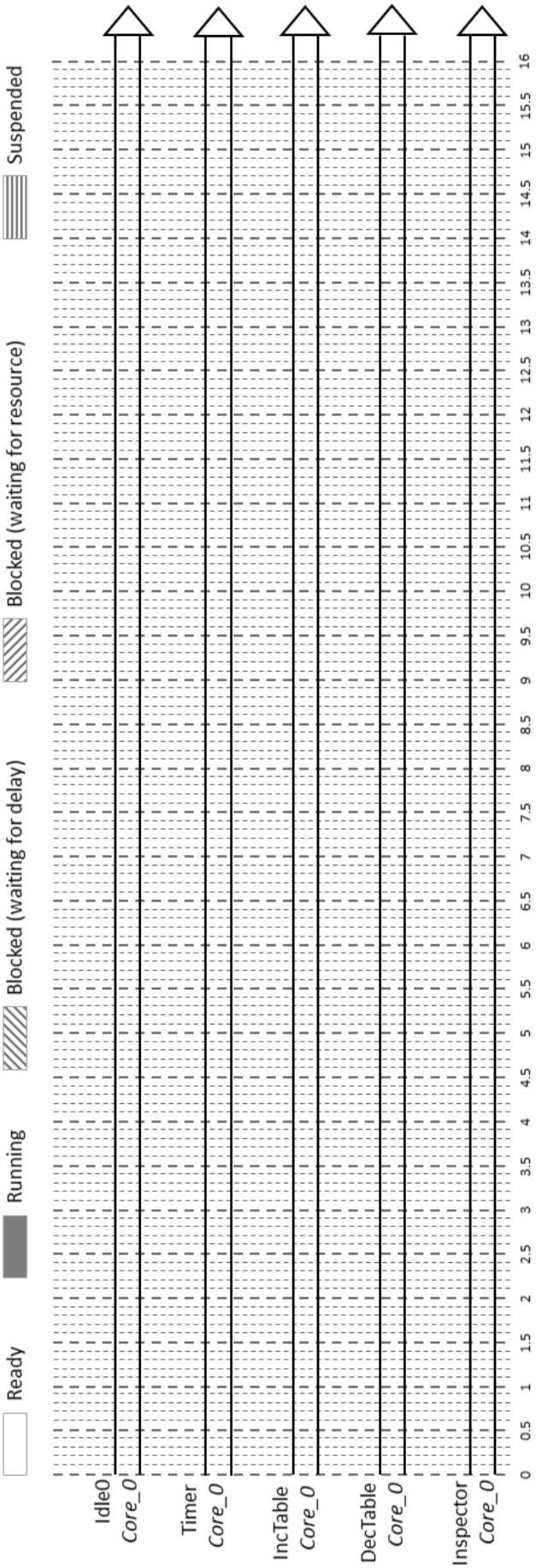


Figure 3.6 – *Program with Task inspector.*

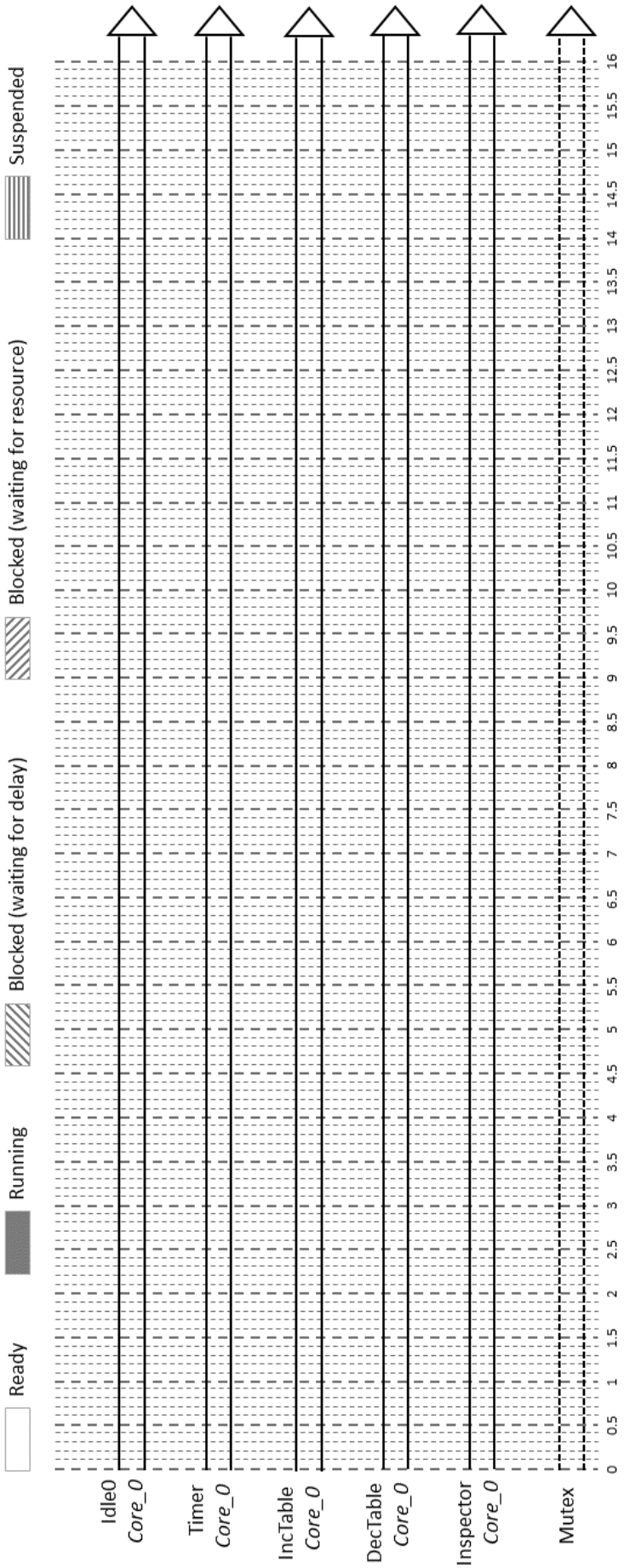


FIGURE 3.7 – *Program with Task inspector on the Core_0 and Mutex.*

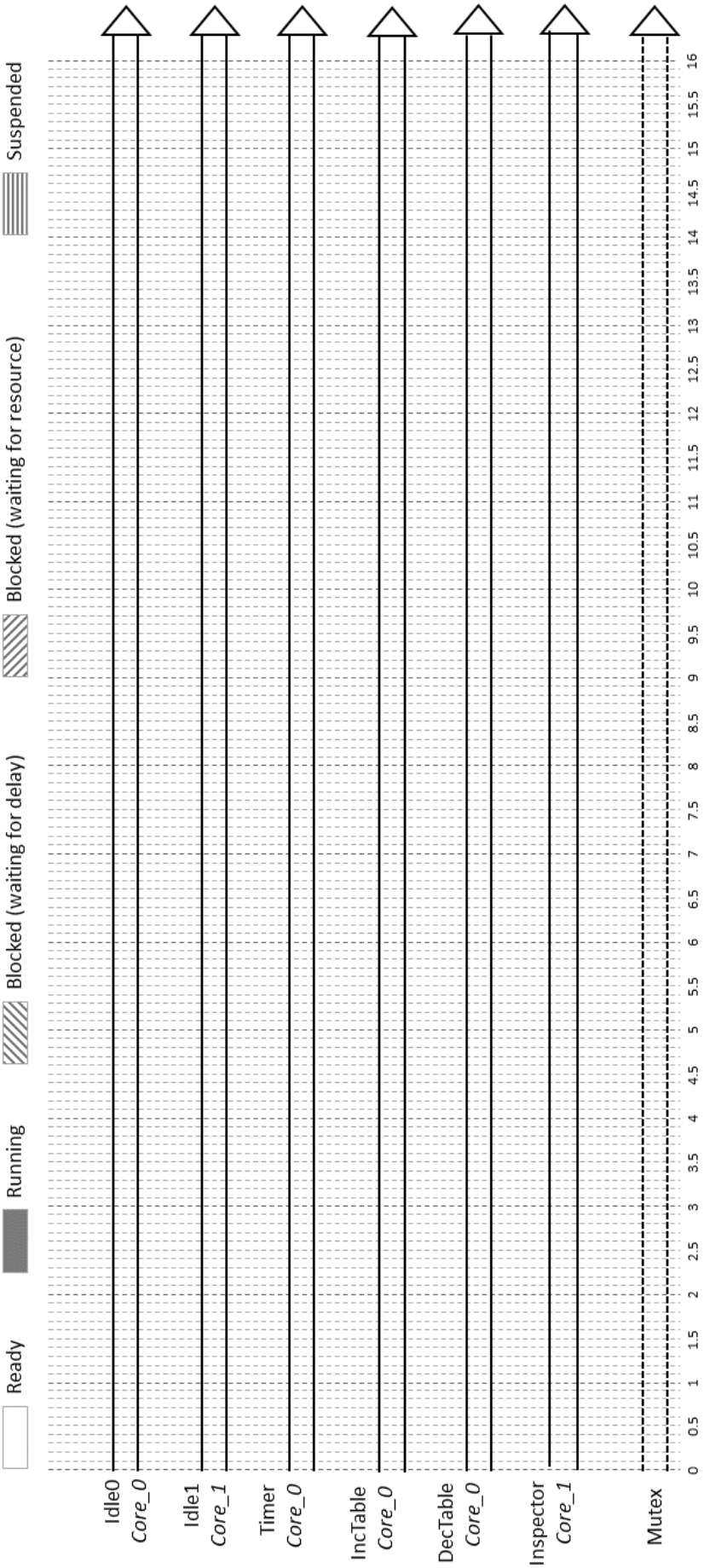


Figure 3.8 – *Program with Task inspector on the Core_1 and Mutex.*

LAB N° 4

Direct task notification

Lab Objectives

- Using task notification instead of semaphore.
- Using event group for a notification.

4.1 Direct task notification (Lab4-1)

A *direct to task notification* is an event sent directly to a task, rather than indirectly to a task via an intermediary object such as a queue, event group or semaphore. The *direct to task notification* is faster than using an intermediary objects and has a RAM Footprint benefits.

- Duplicate the « lab3-2_two_sem_clk » folder to « lab4-1_two_notifications_clk ».
- Answer the following questions :
  — Study the parameters of the *xTaskNotifyGive()* function. Web help

  — Study the parameters of the *ulTaskNotifyTake()* function. Web help

- Replace the two separate semaphores (i.e. notification of the *IncTable* and *DecTable* tasks) by a the *direct to task notification* mechanism. For the *ulTaskNotifyTake()* function, we set the first parameter *xClearCountOnExit* to *TRUE*. Print the return value (pending event counter) of *ulTaskNotifyTake()* function for each task.

- Run the program and explain the behavior.

- In the *Timer* task, add a new *give()* notification to *IncTable* task. So, you have 2 notifications in a row to the *IncTable* task.
- Run the program and explain the behavior.

- We set now the first parameter *xClearCountOnExit* to *FALSE* for the *ulTaskNotify-Take()* function. What is the difference of behavior regarding to the previous question ?

4.2 Direct task notification with a event value (Lab4-2)

To illustrate the principle, we are going to modify the algorithm of the *Timer* task. Its functional behavior is as follows :

Task *Timer* **is**

Properties: Priority = 5, ACTION = eSetBits
Out **:** Clk is event

Cycle :

```
waitForPeriod(250 ms);
computeTime(20 ms);
print("Task Timer : Notify Give (count=%d)", count);
// IncTable task notifications
notify(incTableHandler, (0x01 « count), eSetBits);
notify(incTableHandler, (0x02 « count), ACTION);
// DecTable task notification
notify(decTableHandler, (0x01 « count), eSetValueWithoutOverwrite);
// counter modulo 4
count = (count + 1)
```

end

end

Algorithm 5: New Timer algorithm.

1. Duplicate the « lab4-1_two_notifications_clk » folder to « lab4-2_two_notifications_clk2 ».
2. Answer the following questions :
   - Study the parameters of the *xTaskNotify()* function. Web help
   - Study the parameters of the *xTaskNotifyWait()* function. Web help
3. Modify the code of *Timer* task.
   - The *ACTION* property can be a constant value and can take the *eSetBits*, *eSetValueWithoutOverwrite* or *eSetValueWithOverwrite* value.
   - We set by default the *ACTION* property to *eSetBits*.
   - What is the best value of the first parameter (*ulBitsToClearOnEntry*) of the *xTaskNotifyWait()* function ?

- We set the second parameter (*ulBitsToClearOnExit*) of the *xTaskNotifyWait()* function to 0.

4. Run the program, copy the console until 107 ticks and explain the behavior, in particular the pending counter value in each 2 tasks.

5. We set the *ACTION* property to *eSetValueWithoutOverwrite*. Run the program, copy the console until 107 ticks and explain the behavior.

6. We set the *ACTION* property to *eSetValueWithOverwrite*. Run the program, copy the console until 107 ticks and explain the behavior.

LAB N° 5

Application

Lab Objectives

- Use the knowledge previously seen on freertos services
- Using GPIO
- Using Interrupt

5.1 Specification of the application

We want to implement the functional structure depicted in figure 5.1.

5.1.1 Scan task

The task is activated on *ScanH* event. It acquires the value of the *Value* variable, compares it to high and low thresholds. *Value* being a theoretical output of a Digital/Analog converter not available for this Lab. It is possible to simulate its evolution by a random generation made by *ScanH* when activated.
The *rand()* function allows you to generate a number between 0 and RAND_MAX. So you just need, by rule of three, to bring the result between 0 and 100 and to set the low threshold at 15 and the threshold at 85.
If one of the thresholds is exceeded, a message of *High alarm* or *Low alarm* type is sent to the *Mess* queue, accompanied by the threshold value of *Value* variable. Otherwise, the value of *Value* variable is assigned to *SampleValue*.

```
static const uint32_t VALUE_MAX = 100;
static const uint32_t LOW_TRIG = 15;
static const uint32_t HIGH_TRIG = 85;
```

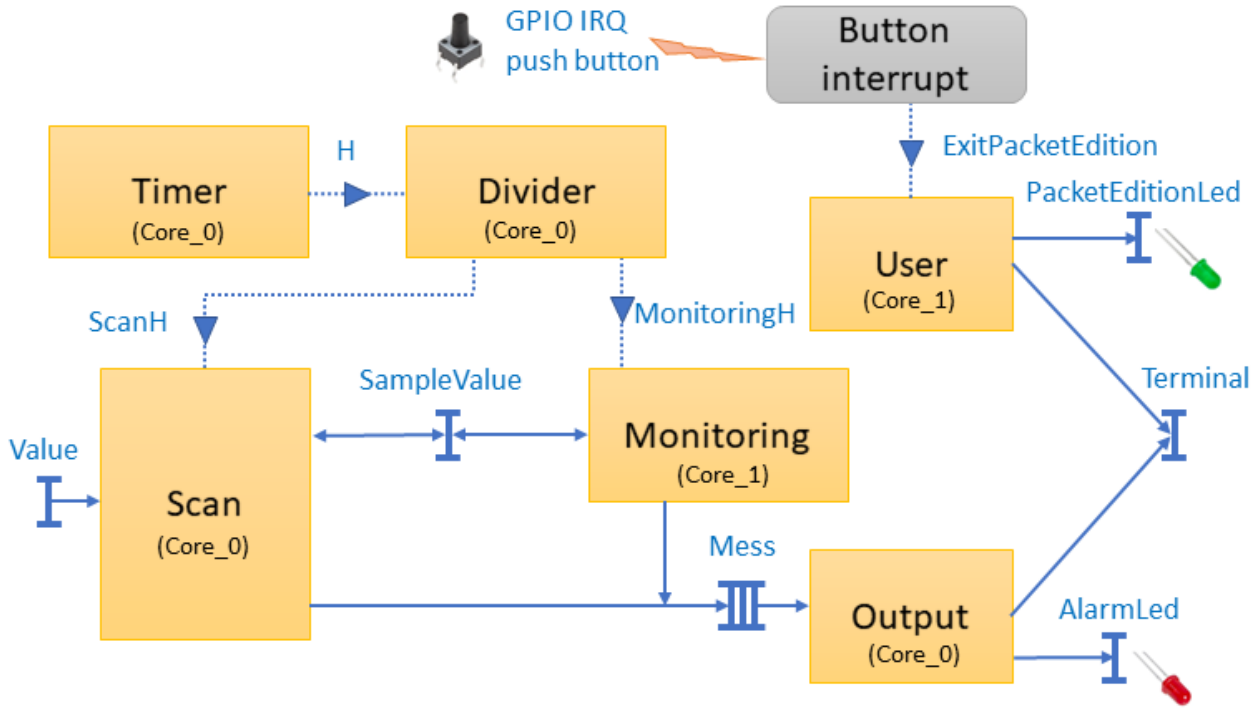


Figure 5.1 – *Functional description.*

```
int iValue = rand() / (RAND_MAX / VALUE_MAX);
if (iValue < LOW_TRIG) {
    // Low Alarm
}
else if (iValue > HIGH_TRIG) {
    // High Alarm
}
else {
    // No Alarm
}
```

5.1.2 Monitoring task

The task is activated on *MonitoringH* event. It sends messages of *Monitoring* type in the *Mess* queue. These messages contain the value of the variable *SampleValue*. Thus, the structure of *Mess* queue can be described as follows :

```
typedef enum typeInfo { Alarm, Monitoring };
typedef struct
{
  enum typeInfo xInfo;
  int xValue;
} typeMessage;
```

5.1.3 Timer task

The *Timer* task generates a periodic *H* event of 10ms.

5.1.4 Divider task

The *Divider* task performs a division of *H* event by 5 to generate *ScanH* event and a division by 18 to generate *MonitoringH* event.

5.1.5 User task

The task is activated by the keyboard. It allows the user to print a packet of characters on the terminal (called *Packet Edition*). These characters are edited by packet, the end of a packet corresponding to the character @ (for example, a packet is « `it is an example@` » as depicted Listing 5.1). When entering a character packet (*Packet Edition* mode), the *Output* function should not display its messages on the terminal and the *PacketEditionLed* is set to 1 (the green led is ON). After entering the character @ (end of *Packet Edition* mode), it can display as many messages as necessary, until the user enters the next character. The characters can be entered using the *getch()* function. An example of use is depicted below :

```
/* Scan keyboard every 50 ms */
while (car != '@') {
  car = getch();
  if (car != 0xff) {
    printf("%c", car);
    }
  vTaskDelay(pdMS_TO_TICKS(50));
}
```

Another solution for exiting *Packet Edition* mode is to press the push button that triggers an interrupt (named *Button interrupt*). This interrupt sends an event (semaphore) to the *User* task so that it can exit *Packet Edition* mode.

5.1.6 Output task

The task displays a formatted message of the *Mess* queue as depicted in the terminal in Listing 5.1. When the message is an alarm, the *AlarmLed* must be set to 1 (the red led is ON).

Listing 5.1 – *Console example of the application*

```
13:0> Monitoring: Value = 69
203:0>  Monitoring: Value = 55
393:0>  Monitoring: Value = 20
583:0>  Monitoring: Value = 84
613:0>  Alarm: Value = 6
733:0>  Alarm: Value = 87
773:0>  Monitoring: Value = 85
```

```
853:0>  Alarm: Value = 98
913:0>  Alarm: Value = 8
963:0>  Monitoring: Value = 15
973:0>  Alarm: Value = 90
1033:0> Alarm: Value = 10
1093:0> Alarm: Value = 92
1153:0> Monitoring: Value = 50
1213:0> Alarm: Value = 87
1343:0> Monitoring: Value = 69

User mode
hello !@
End User mode
2393:0> Alarm: Value = 3
2393:0> Monitoring: Value = 15
2393:0> Alarm: Value = 9
```

5.2 Implementation of the application (Lab5)

1. Create the « lab5_application » lab from « esp32-vscode-project-template » GitHub repository.
2. Copy the provided « my_helper_fct.h » file to the « main » folder.
3. Overwrite the « main.c » file by the provided code of the « lab5_main.c » file.
4. Create a « sdkconfig.defaults » file with right parameters.
5. Write the program for *Timer*, *Divider*, *Scan* and *Monitoring* tasks. The capacity of *Mess* queue is 5. The tasks are mapped on different cores referenced in the figure 5.1.
6. Build to check that is no compilation error.

In order to validate the behavior of the application and to clearly highlight the management of *Mess* queue, we can proceed as below :

1. Validate the functional structure with only the *Timer*, *Divider*, *Scan* and *Monitoring* tasks. Check that after a number of message submissions corresponding to the maximum capacity of the *Mess* queue, the *Monitoring* and *Scan* tasks are blocked.
2. Wire the 2 leds on the board.
3. Add the *Output* task. We will check that the *Monitoring* and *Scan* tasks are no longer blocked. Only the *Output* task can be blocked on an empty *Mess* queue. Check the behavior of the red led.
4. Add the *User* task without using interrupt. Note that the *getch()* function also uses the terminal as the *printf()* function. You must protect the simultaneous write on the terminal. Check the application and green led behaviors.
5. Wire the push button on the board.

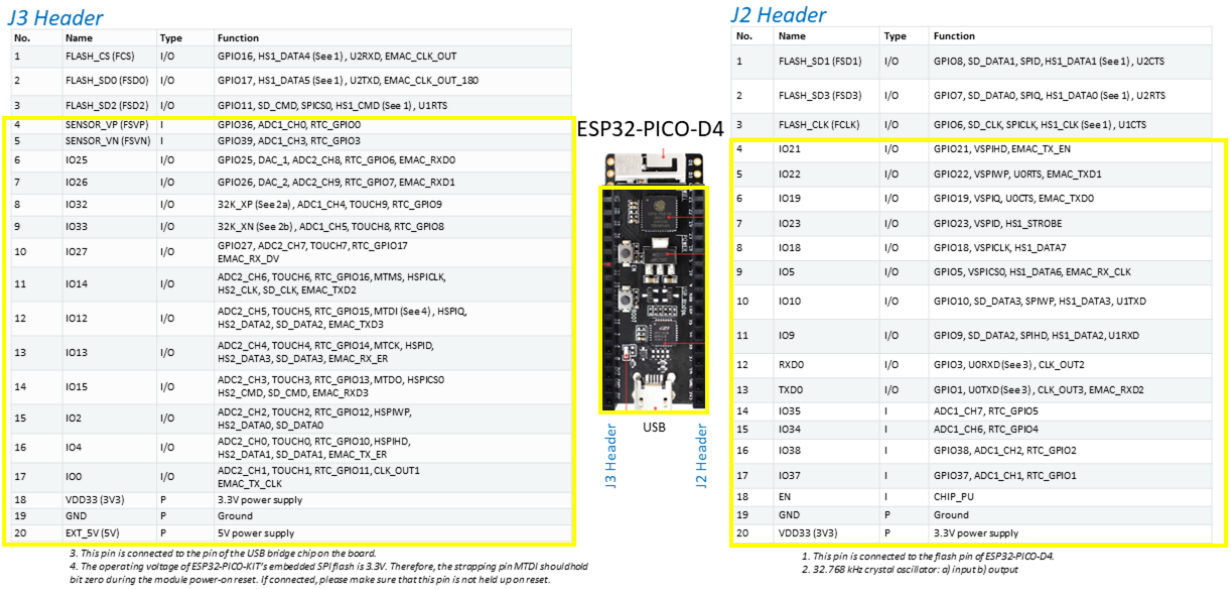
6. Add the *button interrupt* function and check the behavior. The name of the *give()* function is different when called from an interrupt (Web help).

Part IV

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.

ESP32-PICO-D4

USB

J3 Header

J2 Header

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