Lab 1: reaction test game

In this lab exercise you will learn a bit more about the ESP-IDF and prove what you have learned so far. To be able to go through this lab you will need to have knowledge related to:

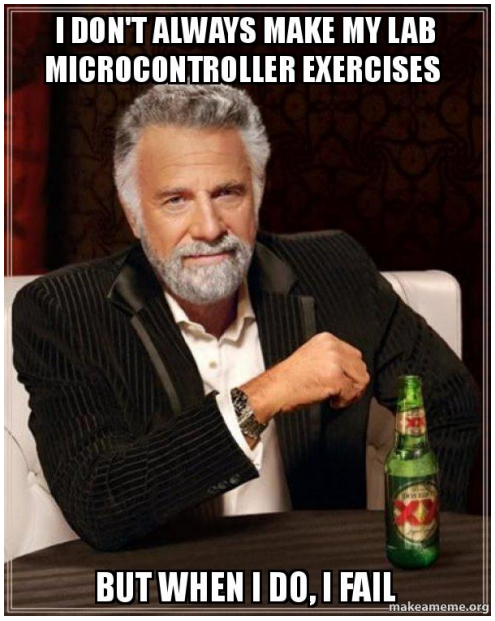
- General C programming.
- Modules, compilation process, preprocessor in C.
- The ESP32 in general and its ESP-IDF software development kit.
- Some basic digital input/output using GPIOs.

In terms of materials you will need:

- A NodeMCU 32S board or equivalent.
- USB cable.
- Two LEDs, better if with different colors.
- A breadboard.
- Two pushbuttons.
- A bunch of wires.

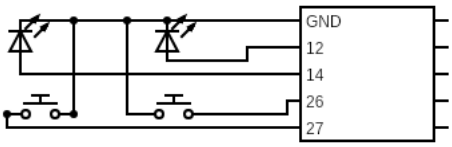
The lab will consist of developing a simple game for the ESP32. The game will work like this: there are 2 players, each controlling a pushbutton. Two blinking LEDs will signal the start of the game and will go OFF. After a random amount of time (between 3 and 5 seconds) both LEDs will be ON until someone presses a button. The first player to press the button will win. The LED closer to the button that was faster will blink to indicate the winner and the game will restart.

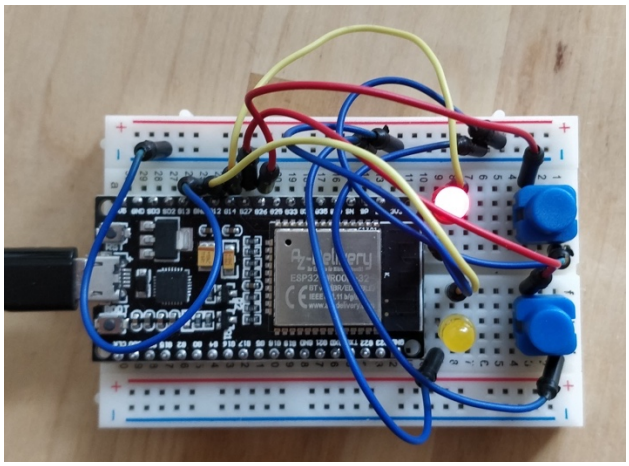
It's not an easy task, but we'll get there step by step. Good luck!



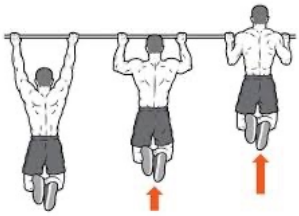
Setting up the hardware

First, let's connect the two LEDs and the two pushbuttons. You will need some wires and, unless you want to use just the pins on one side of the board, cut the board into 2 halves.

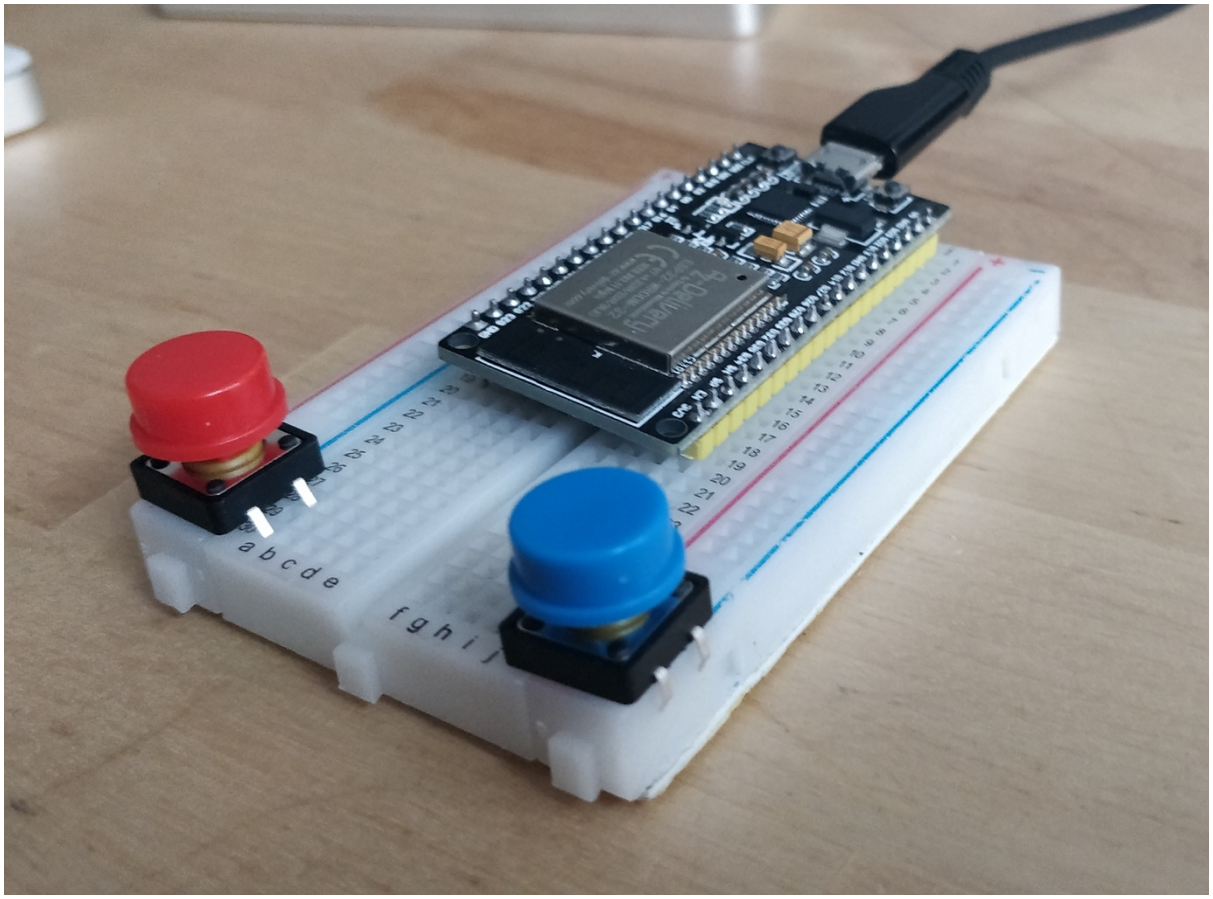




In my example here I am using pins 12 and 14 to power the LEDs and pins 26 and 27 for the pushbuttons. The other terminal of the pushbutton is connected to GND. This means that when the buttons are not pressed, their corresponding pins the pin will be "floating". You know how to solve this, don't you?
Hint:



One note, the space on the breadboard is very small and you may struggle finding space for the buttons, this is how I solved it:



Preparing the code

As in this game there are some long waiting times, we need to deal with the watchdog timer reset. The watchdog reset will become clear later: for now, just think of it as a security mechanism that resets the microcontroller after a while if you don't deal with it.

To keep the watchdog quiet, when you want to pause the execution for a while, instead of using *ets_delay_us()* as used in lectures, we will use *vTaskDelay()*. The function uses a multiplier of a constant portTICK_PERIOD_MS as input. Just use the following example:

```
#include <esp_task_wdt.h>

/*
  Waits for "millis" milliseconds without upsetting the watchdog timer
*/
void waitMs(unsigned int millis)
{
    TickType_t delay = millis / portTICK_PERIOD_MS;
    vTaskDelay(delay);
}
```

Now we need to configure the GPIOs. I want you to practice with C modules, so we'll create a simple module with the following two files:

pins.h

```
#ifndef PINS_H_
```

```h
#define PINS_H_

#include <stdint.h>

/* initialises the 4 pins */
void initPins();

/* switches LED A on if level!=0 or off if level==0*/
void setLEDA(uint8_t level);

/* switches LED B on if level!=0 or off if level==0*/
void setLEDB(uint8_t level);

/* tells if button A is currently being pressed */
uint8_t isButtonAPressed();

/* tells if button A is currently being pressed */
uint8_t isButtonBPressed();

#endif
```

I am already giving you the signature of the functions here, but feel free to change them according to your taste.

The implementation file, pins.c, comes drafted here, you will have to complete it:

```c
#include "pins.h"
#include "driver/gpio.h"

/* initialises the 4 pins */
void initPins()
{
    // init the 2 LEDs pins as output and the 2 buttons' pins as input
    // you will need to use gpio_config()
}

/* switches LED A on if level!=0 or off if level==0*/
void setLEDA(uint8_t level)
{
    if (level)
    {
        // set the pin of LED A to ON
        // you probably need to use gpio_set_level()
    }
    else
    {
        // set the piin of LED A to OFF
    }
}

/* switches LED B on if level!=0 or off if level==0*/
void setLEDB(uint8_t level)
```

```
{
    // same as setLEDA()
}

/* tells if button A is currently being pressed */
uint8_t isButtonAPressed()
{
    // read the value of button A
    // if using a pull-up, the button is pressed when the pin is LOW

    return 0;
}

/* tells if button A is currently being pressed */
uint8_t isButtonBPressed()
{
    // same as with button A, but another pin

    return 0;
}
```

Here it is fundamental that you have a deep look at the ESP-IDF documentation, especially the one related to GPIOs:
https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/peripherals/gpio.html

You will most likely need to use the following functions:

- gpio_config: to configure your 4 pins
- gpio_set_level: to set the LED pins ON or OFF
- gpio_get_level: to read the level of a pin connected to a button

The only trick is really taking care of the fact that the pin connected to the button is floating when the button is not pressed (see suggestion at the beginning!).

To test this, you can start with the following main.c:

```
#include <esp_task_wdt.h>
#include "pins.h"

/*
  Waits for "millis" milliseconds without upsetting the watchdog timer
*/
void waitMs(unsigned int millis)
{
    TickType_t delay = millis / portTICK_PERIOD_MS;
    vTaskDelay(delay);
}

void app_main()
{
    initPins();

    while (1)
```

```
    {
        if (isButtonAPressed())
        {
            setLEDA(1);
            setLEDB(0);
        }
        else if (isButtonBPressed())
        {
            setLEDB(1);
            setLEDA(0);
        }
        else
        {
            setLEDA(1);
            setLEDB(0);
            waitMs(500);
            setLEDA(0);
            setLEDB(1);
            waitMs(500);
        }
    }
}
```

If no button is pressed, you should see the LEDs alternating. If you press (and keep pressed for some seconds) a button, you should the corresponding LED lighting up.

If things go wrong…

And they will! Usually it's not the code the one to blame, but the electronics. Check the connection of each wire and component over and over. Simplify the code to its minimum to understand if connections are working as expected. Use printf() when it makes sense. Check that you are using the right pins! For example, I have wasted 3 hours preparing this lab because I connected ground to the wrong pin (the one that is marked as GND, but it's actually CMD!).

Random number

In the game, you need to wait for a random number of seconds (let's say between 3 and 5), so we will need to produce that random number somehow. Fortunately, the ESP32 has a hardware random number generator embedded. See the documentation at:

https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/system/system.html?#_CPPv410esp_randomv

Now I want you to create a separate module (random.h and random.c files) with just one function, *getRandommsecs*, that gives a random number between min and max milli seconds.

```
/* Returns a random number contained between min and max.
   min: minimum number of ms
   max: maximum number of ms
```

```
*/
int getRandommsecs(int min, int max)
```

Write both the .h and .c files and test them inside this main.c file:

```
#include <esp_task_wdt.h>
#include "pins.h"
#include "random.h"

/*
  Waits for "millis" milliseconds without upsetting the watchdog timer
*/
void waitMs(unsigned int millis)
{
    TickType_t delay = millis / portTICK_PERIOD_MS;
    vTaskDelay(delay);
}

void app_main()
{
    initPins();

    while (1)
    {
        int r = getRandommsecs(100, 500);
        printf("random value: %d\n", r);
        setLEDA(1);
        setLEDB(0);
        waitMs(r);
        setLEDA(0);
        setLEDB(1);
        waitMs(r);
    }
}
```

You should see the LEDs flickering randomly. Also check on the serial console that the values are randomly distributed between min and max.

The only tricky bit here is to transform the 32-bit random value into an integer that ranges from min to max. One possible way to do it, is by converting that random value into a float that ranges 0 to 1 (just divide it by UINT32_MAX), let's call it r, and then return something like min + r * (max – min). Make sure the types are converted properly.
Can you think of any other simpler strategy?

Putting it all together

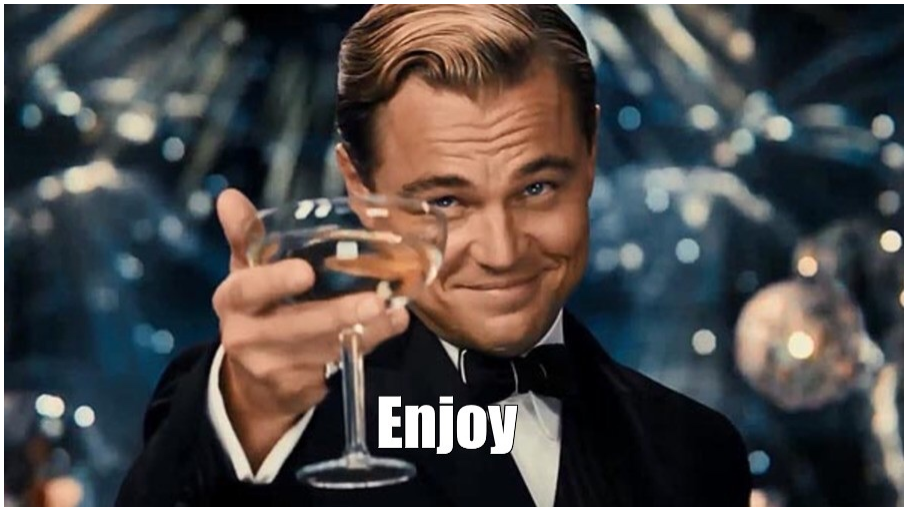
Now that you have all the building blocks for your game, you need to put all of them together.

I will not give you the full solution, but I can hint you at a way for doing it. See the following:

```c
#include <esp_task_wdt.h>
#include "pins.h"
#include "random.h"

/*
  Waits for "millis" milliseconds without upsetting the watchdog timer
*/
void waitMs(unsigned int millis)
{
    TickType_t delay = millis / portTICK_PERIOD_MS;
    vTaskDelay(delay);
}

void app_main()
{
    initPins();

    while (1)
    {
        // signal that the game is about to start
        // you can flash LEDs in a certain way for it

        // switch both LEDs off

        // get a random duration between 3 and 5 seconds

        // wait that random duration

        // switch both LEDs ON

        uint8_t winner = 0;
        while (!winner)
        {
            // check if either button A or B are pressed

            // if any is pressed, set winner to 1 for A or 2 for B
        }

        // if A wins, flash LED A and switch off B
        // if B wins, flash LED B and switch off A

        // switch off both A and B and wait for some time to restart the game
    }
}
```

When solving this, I created a helper function called *flashPin(uint8_t pinA, uint8_t pinA, int ms)* that would flash pin A, or B or both for a given amount of ms. But You can decide yourself how to do it.

Once finished, you should be able to play the game with someone.



Getting evaluated

Once you are finished with the code, and you are sure it is working as it should, zip the content of the src folder and upload it to Canvas under "Grades". I only need the src folder, not all the project! The code must compile and work as expected. I will test it on my computer.

If you use git, you can create a repository on a public server like Github. In this case, you can submit a text file with the link to the repository.

If you are working with other students please remember that you will be assessed individually, so avoid copying the code from other students! It is usually easy for us to spot it because you all have slightly different coding styles. If you cheat you will fail the lab and will be reported.

If 2 or more students work on the same code by sharing it, for example using git, please let us understand who did what. If you use git, we can see who committed what code, so make sure the code is not committed only by one student. If you don't use git, let us know how you have worked and leave some notes in the code using comments so that we understand who did what.

To earn extra points

The code you have just completed is quite simple and has some defects. For example, if any player keeps the button pressed all the time, he or she will win. How can we avoid this happening?

One simple way would be to check if the button is pressed *before* the two LEDs are switched on.

A better way would be to detect any button pressed *during* the waiting time, but how? We need to change the "wait" that waits for the random number of seconds into something like "wait and check". To do that, imagine that you have a function, let's call it *millis*() that returns the number of milliseconds since the board was started (millis() actually exists in Arduino!). Then you could do something like:

```
        // wait that random duration
        unsigned int startMs = millis();
        while (millis() - startMs < waitTime)
        {
            // check if any player is cheating
            // if cheating, exit this loop and let the other player win immediately

            // wait a very little amount of time (like 10ms)
        }
```

This code basically continuously checks if anybody is cheating until enough time has passed. It's a very common strategy to use when you cannot have parallelly execute code so keep it mind!

But, how do you implement millis() ? That's for you to discover. Have a look at this for an answer:
https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/system/esp_timer.html

Other things you can do to earn extra points in your lab are:

- Write good comments in the code, document your functions.
- Manage failure cases, for example, what happens if any of the calls to gpio_config() fails? Would you print something on the serial line? Have a look at these helper functions for error cases: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/system/esp_err.html
- Do not hardcode constants like pin numbers, use #define instead, this makes your code easier to change if, for any reason, you need to switch to another pin.
- Can you find a way to generate a random number between min and max using only integers arithmetic?

In general, also the eye wants its part. The better your code looks like, the easier it would be to earn extra points!