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. The detailed behavior of the tasks is described later.

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 :
   - The *Task 3* blocks for 100ms, displays string to the terminal and then simulates a computation of 20ms.

     ```
     // Task blocked during 100 ms
     DISPLAY(...);
     vTaskDelay(pdMS_TO_TICKS(100));
     DISPLAY(...);
     // Compute time : 20 ms
     COMPUTE_IN_TIME_MS(20);
     ```

   - 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 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 : 40 ms
     COMPUTE_IN_TIME_MS(40);
     // block periodically : 500ms
     vTaskDelayUntil(...);
     ```

   - The *Task 2* waits for a message through the message queue, displays the task number and message received and then simulates a computation of 30ms.

     ```
     // Wait for message
     ...
     // display task number and message
     DISPLAY(...);
     // Compute time : 30 ms
     COMPUTE_IN_TIME_MS(30);
     ```

   - 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 ?

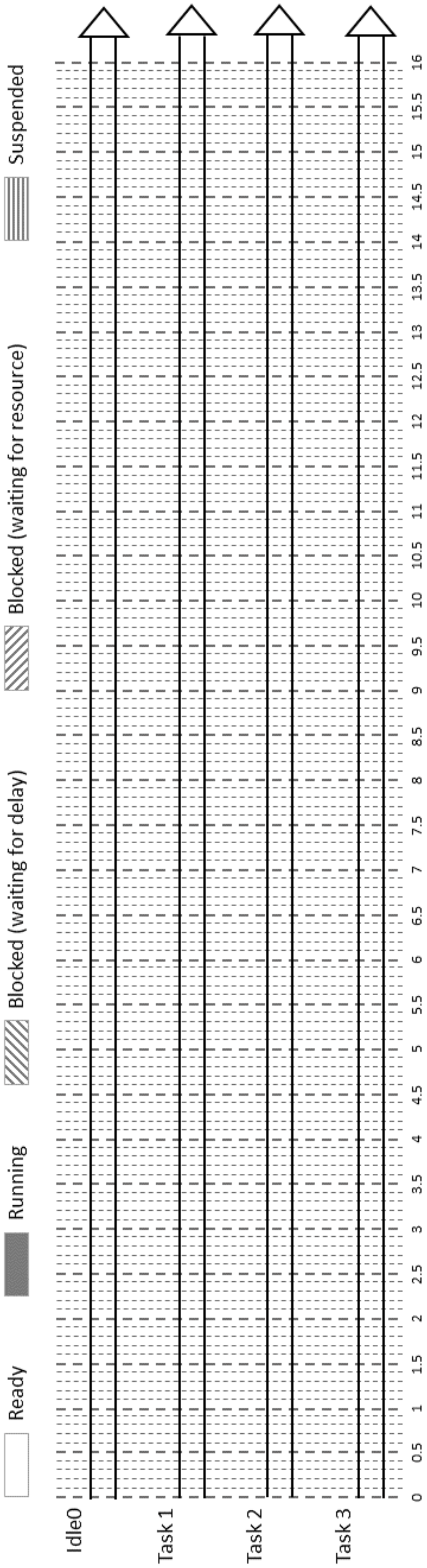

Figure 2.1 – *Message queue and 3 tasks.*

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_TIME_MS(30);
}
else {
  DISPLAYE(TAG, "Task 2, Timeout!");
  COMPUTE_IN_TIME_MS(10);
}
```

- 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.

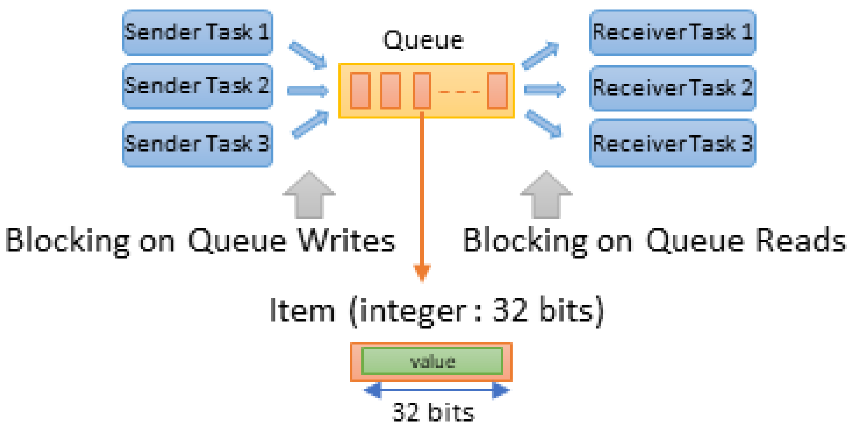

FIGURE 2.2 – *Blocking on single queue.*

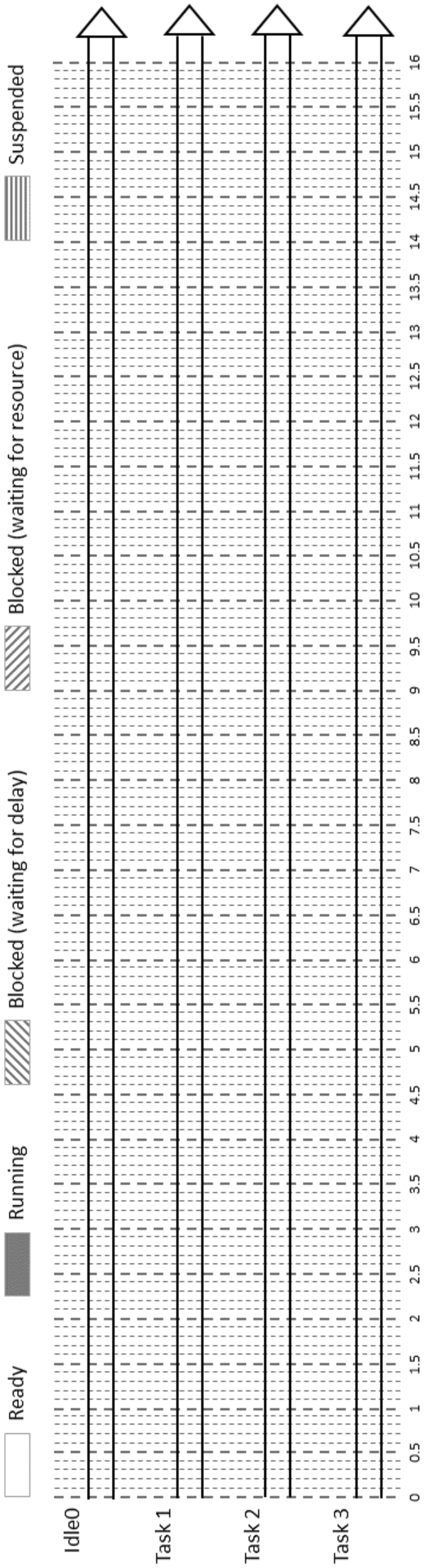

Figure 2.3 – *Message queue with timeout and 3 tasks.*

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*.

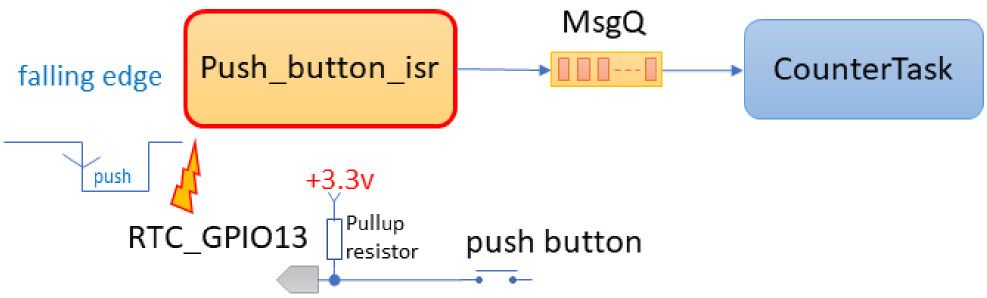

FIGURE 2.4 – *Interrupt application with message queue.*

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) as below. Note that the RTC_GPIO13 is the IO15 pin in the board.

```
/* Config GPIO */
gpio_config_t config_in;
config_in.intr_type = GPIO_INTR_NEGEDGE;  // falling edge interrupt
config_in.mode = GPIO_MODE_INPUT;      // Input mode (push button)
config_in.pull_down_en = false;
config_in.pull_up_en = true;          // Pull-up resistor
config_in.pin_bit_mask = (1ULL<<PIN_PUSH_BUTTON); // Pin number
gpio_config(&config_in);
```

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 as below :

```
/* Install ISR */
gpio_install_isr_service(0);
gpio_isr_handler_add(PIN_PUSH_BUTTON, Push_button_isr_handler, (void
    *)PIN_PUSH_BUTTON);
```

11. Write the body of *vCounterTask()* task by following the comments in the source code.
12. Perform the wiring on the board. The RTC_GPIO13 is the IO15 pin. 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.