LAB #2—Intro to FreeRTOS

Kizito NKURIKIYEYEZU

October 4, 2021

1 LAB OBJECTIVES

- Introduction to FreeRTOS
- Learn asks states, how to create and use FreeRTOS tasks
- How to use queue for inter-tasks communication
- Critical section protection

2 EXERCISE 1

This exercises consists of three tasks. Task1 flashes the GREEN LED every second. Task2 flashes the RED LED every 200 ms. Task2 is suspended by Task1 after ten seconds, and as a result Task2 stops flashing the RED LED. Task2 is then resumed after 15 seconds at which point Task 2's RED LED starts to normally flash again. Task3 controls a button switch, which, once pressed, will reset the countdown of when Task2 must be suspended (i.e., it will reset the number of elapse second until when the Task2 must be suspended).

To solve this exercise, you can use FreeRTOS API functions vTaskCreate()[1], vTaskDelay()[2], vTaskSuspend()[3], vTaskResume()[4], and macro pdMS_TO_TICKS(). You may use FreeRTOS's vApplicationTickHook()[5] to estimate the elapsed time.

You should start with the code template[6] attached to this lab. The template provide some startup code and shows how to configure key components of the simulator. However, you are expected to remove components that are not relevant to the lab.

[1] `https://www.freertos.org/a00125.html`
[2] `https://www.freertos.org/a00127.html`
[3] `https://www.freertos.org/a00130.html`
[4] `https://www.freertos.org/a00131.html`
[5] `https://stackoverflow.com/a/39205647`
[6] `https://qiriro.com/epe2165/static_files/labs/lab2/Lab2.zip`

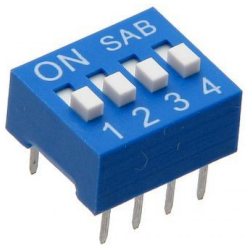

FIGURE 1. A 4-bit DIP Switch

3 EXERCISE 2

3.1 Description

In this exercise, an LED's flashing rate is modified depending the binary value of a 4-bit dual in-line package (DIP) switch. A 4-bit The DIP switch consists of 4 switches (S3,S2,S1 and S0) in a single unit and packaged in a standard dual in-line package (Figure 1). Each switch of the DIP switch can be controlled independently of others. In this exercises, the DIP switch is used as a 4-bit counter (thus, allow to count from 0 to 15) and its counter is used as a delay parameter to change the LED's flashing rate as follows:

- When all the four switches off (i.e., $S_3 = S_2 = S_1 = S_0 = 0$), the LED is off
- When only S_0 is pressed (i.e., $S_3 = S_2 = S_1 = 0, S_0 = 1$, which corresponds to a binary value of 1), the LED flashes every second.
- When only S_1 is pressed (i.e., $S_3 = S_2 = 0, S_1 = 1, S_0 = 0$, which corresponds to a binary value of 3), the LED flashes every half of second.
- When S_0 and S_1 are pressed (i.e., $S_3 = S_2 = 0, S_1 = 1, S_0 = 1$, which corresponds to a binary value of 3), the LED flashes every third of second.
- etc ...
- When all switches are pressed (i.e., $S_3 = S_2 = S_1 =, S_0 = 1$, which corresponds to a binary value of 15), the LED flashes at $T = \frac{1}{15}$ second.

3.2 Implementation details

You should start with the code template[7] attached to this lab. The template provide some startup code and shows how to configure key components of the simulator. However, you are expected to remove components that are not relevant to the lab.

The DIP counter should be stored in a queue instead of a global variable and the LED flashing rate is passed to the task controlling the LED using this queue. The queue length

[7] https://qiriro.com/epe2165/static_files/labs/lab2/Lab2.zip

should be set to 1 since only one value is to be sent via the queue. The program consists of two tasks:

- *vCounterTask(void *pvParameters)*—This tasks keeps track of switches of the DIP switch. It monitors which switches are pressed and converts the corresponding binary value to a digital value. This value is then converted into a time delay (rounded to the nearest millisecond) and sent to the queue. This can be done using the *xQueueSend* API as shown below.

  ```
  xQueueSend(xDelayQueue,&delay_in_ms,pdMS_TO_TICKS(10));
  ```

  In the above pseudocode, the parameter *xTicksToWait* is set to 10ms so that the task will be in Blocked state for 10ms if there is no space available in the queue. You are responsible to properly create and initialize the variables *xDelayQueue*, and *delay_in_ms*.

- *vFlashLedTask(void *pvParameters)*—This task blink the LED depending on the value of the delay it receives from the queue. The flashing rate is read from the queue as follows

  ```
  xQueueReceive(xQueue, &flash_rate_in_ms,0);
  ```

  In the above pseudocode, the parameter *xTicksQueueWait* is set to 0 so the function call will return immediately if there is no data in the queue. The value, *flash_rate_in_ms* read by the queue function is thereafter used in function vTaskDelay() as the new flashing rate of the LED.

Note: All the variables used in the two both tasks should be local variables to their tasks and no global variables should be used in the program. Although using global variables may seem to be an easier option in this case, it has the disadvantages and using global variables is generally considered a poor programming style.

4 EXERCISES 3

4.1 Introduction

In multitasking systems, it is very common for the tasks to cooperate and share the available resources. For example, various tasks may want to share the communications interfaces that can only be used by one task at a time.

Resource sharing such as above is handled in FreeRTOS using mutexes and semaphores.

- A mutex is also called a binary semaphore. Mutexes are used to protect critical sections of a code so that only one task has access to it at any time. The operation of a mutex is such that a mutex must be created before it can be used. A task then takes a mutex before entering a critical section. During this time, no other tasks can have access to the shared resource since the mutex controlling the critical section is not available. When the task finishes its processing within the critical section, it gives the mutex. At this point, the mutex is available for any other tasks who may want to use the critical section. A task that takes a mutex must always give it back; otherwise, no other task can access the critical region.

- Semaphores can be binary or counting. A binary semaphore is similar to a mutex as it has two states. A counting semaphore contains a counter with an upper bound which is defined at the time the semaphore is created. The counter keeps track of limited access to the shared resource (i.e., critical section). When a task wants to use the critical section, it takes the semaphore and, as a result, the counter is decremented if it is not zero. Another task can also take the semaphore where the counter is decremented again. When the count value reaches zero, the resource is not available and further attempts to take the semaphore will fail. When a task releases the semaphore (i.e., gives the semaphore), then the counter value is incremented. Semaphores that are created to manage resources should be created with an initial count value equal to the number of resource that are available.

Notice that although the mutexes and binary semaphores are very similar, there are some important differences between them. Mutexes include priority inheritance, binary semaphores do not. The priority of a task that holds a mutex is raised if another task of higher priority attempts to obtain the same mutex. The task that already holds the mutex is said to inherit the priority of the task attempting to take the same mutex. The inherited priority will be disinherited when the mutex is returned. A binary semaphore does not need to give back the semaphore after it has been taken. Creating a mutex or a semaphore returns a handle to the creator. This handle identifies the created mutex or the semaphore.

This exercise show how a mutex can be created and taken and given by tasks in a multi-tasking environment in order to access a shared critical region.

5 Description

In this exercise, two 4-bit DIP switches are used to control the blinking rate of an LED. The LED is shared among the tasks and is assumed to be in a critical section where the sending tasks must take/give a mutex in order to use the LED.

There are three tasks:

- *vDipSwitch1Task(void *pvParameters)*—This tasks keeps track of switches of the first DIP switch. It monitors which switches are pressed and converts the corresponding binary value to a digital value. This value is then converted into a time delay (rounded to the nearest millisecond). The task then attempts to take the mutex. If the mutex is taken successfully (i.e., pdPASS is returned), then the value is sent to the LED controlling task and used to control its flashing rate as shown in the following pseudo-code.

```
if(xSemaphoreTake(xSemaphore, pdMS_TO_TICKS(1000)) == pdTRUE)
{
  xQueueSend(xDelayQueue,&delay_in_ms,0);
  xSemaphoreGive(xSemaphore);
}
```

This task's control of the LED is displayed every 5 seconds, i.e., the task should give the mutex and waits for 5 seconds.

- *vDipSwitch2Task(void *pvParameters)*—is similar to *vDipSwitch1Task(void *pvParameters)* but controls the second DIP switch. Furthermore, this task control the LED every 2 seconds, i.e., the task should give the mutex and waits for 2 seconds.
- *vFlashLedTask(void *pvParameters)*—this task blinks the LED depending on the values sets on the two switches and the flashing rate is read from the queue.

You should start with the code template[8] attached to this lab. The template provide some startup code and shows how to configure key components of the simulator. However, you are expected to remove components that are not relevant to the lab.

Note: You should minimize the use of global variables. Ideally, all the variables used in the two both tasks should be local variables to their tasks and no global variables should be used in the program. Although using global variables may seem to be an easier option in this case, it has the disadvantages and using global variables is generally considered a poor programming style.

6 EXERCISE 4

6.1 Introduction

Software timers are important parts of any real-time multitasking operating system. The timers are used in tasks to schedule the execution of a function at a time in the future, or periodically with a fixed frequency. Software timers under FreeRTOS do not require any hardware and are not related to hardware timers as they are implemented in software. When a timer expires, the program can be configured to call a function names as the timer's callback function.

Software timers are optional in FreeRTOS and the application programs must be built with the source file *timers.h* included as part of the program. Parameter *configUSE_TIMERS* must be set to 1 in file *FreeRTOSConfig.h* before the software timers can be used.

Two types of software timers are supported by FreeRTOS:

- One-shot timers—These timers are started manually and do not re-start when they complete. The callback function is executed only one when the timer expires.

[8] https://qiriro.com/epe2165/static_files/labs/lab2/Lab2.zip

- Auto-reload timers—These timers re-start each time they expire, thus resulting in repetitive execution of the callback function attached to the timer.

A software timer can be in one of two states: Dormant, and Running. A Dormat timer exists but it is not active. A Running timer is active and it will call its callback function when its period expires.

All software timer callback functions execute in the context of the same RTOS daemon (or “timer service”) task. The daemon task is a standard FreeRTOS task that is created automatically when the scheduler is started. The priority and stack size are set at compile time by the two parameters in file FreeRTOSConfig.h: *configTIMER_TASK_PRIORITY* and *configTIMER_TASK_STACK_DEPTH*. Callback functions must not call to functions that may cause the enter the Blocked state.

A timer must be created before it can be used. Creating a timer does not start it. Timers must be started, stopped, or reset manually by the user programs. Software timer API functions send commands from the calling task to the daemon task on a queue called the “timer command queue”. The timer command queue is a standard FreeRTOS queue that is created automatically when the scheduler is started. The length of the timer command queue is set by the configTIMER_QUEUE_LENGTH compile time configuration constant in FreeRTOSConfig.h.

The daemon task is scheduled like any other FreeRTOS task and it will process commands, or execute timer callback functions, when it is the highest priority task that is able to run. Parameter configTIMER_TASK_PRIORITY controls the timer task priority and is recommended to be set to higher than other tasks to allow the timers to work smoothly.

6.2 Description

In this exercise, you will create a reaction timer. The exercise uses an LED and a push-button switch. The user is expected to press the push-button switch as soon as the LED is turned ON. The time between the LED being turned ON and the user pressing the button is measured and displayed on an LCD in milliseconds. The LED is turned ON again after a random delay, ready for the next measurement.

Algorithm 1: Pseudo-code to control the reaction timer device

```
Input:
  • RESET_SWITCH —when the reset switch is pressed,
    the device turn the LED and clears the LCD display
  • REACTION_TIMER_SWITCH —when pressed, the user's reaction time is displayed on
    the LCD display.
Initialization
    • All pins connected to the switches are initialized as inputs
    • All pins connected to the LED and LCD are initialized as outputs
    • Turn off the LED and clear the LCD display
end
Output: A message on the LCD that shows how many milliseconds a user took to press
        an the REACTION_TIMER_SWITCH
Loop
    if RESET_SWITCH==PRESSED then
        Turn OFF the LED
        Clear the LCD display
    if REACTION_TIMER_SWITCH==PRESSED then
        Wait 5 seconds
        Wait random time between 1 and 10 seconds
        Turn the LED ON
        Save the current time (in terms of tick counts)
        Wait until push-button switch is pressed
        Save the new tick count
        Calculate the elapsed time
        Turn LED OFF
        Display elapsed time in milliseconds on LCD
EndLoop
```

6.3 Implementation details

- Start with the code template[9] attached to this lab. The template provide some startup code and shows how to configure key components of the simulator.
- The code template is configured to use the standard Arduino LCD display[10] but you're free to use any other libraries of your liking.
- The program will consist of only one task
- You can generate the random numbers using the *rand()* as shown in Listing 1

```
#include <stdint.h>
```

[9] https://qiriro.com/epe2165/static_files/labs/lab2/Lab2.zip
[10] https://www.arduino.cc/en/Reference/LiquidCrystal

```
#include <stdlib.h>
#include <time.h>
uint8_t get_random_number(uint8_t min, uint8_t max)
{
  srand(time(0));
  double scaled = (double)random()/RANDOM_MAX;
  return (max - min +1)*scaled + min;
}
```

Listing 1: Function to generate a random number between two integers

- The random number generate from the pseudo-code in Listing 1 is used to create random delay in the program so that the user does not know when the LED will lit again.
- As soon as the LED is turned on, the current tick count is stored in variable. The program then waits until the button is pressed by the user, and then calculates the elapsed time by getting the new tick count and subtracting the old tick count from it. This value is the reaction time of the user in milliseconds, which is then displayed on the LCD.

7 LAB REPORT

Each exercise should be accompanied by **1 one page** lab report. The lab report should be written as if the reader was another member of the class. With your report in hand, one should be able to reproduce what you did and understand it well enough to add to it. The report shall contain the following sections

- **Abstract** —1 paragraph describing what you did. It should explain the purpose of the lab, results, its scope, summary of your solution and a conclusion.
- **Introduction** —What is the lab about project? Why is it interesting? What technical details does your reader need to understand?
- **Circuit description** Draw block diagram of the circuit. Please note that you should note use the Proteus diagram. Instead, you should manually draw a separate block diagram (and not an actual circuit) dthat shows only components and pins that are being used on the MCU. Discuss how various resistors, transistors have been selected. Show any pertinent calculation when necessary.
- **Code description** —Start with a very high level description of your code (block diagrams, perhaps) and how it uses subsystems... Get increasingly detailed in your description until you get to the level of short sections of code that are critical to the proper operation of your project.
- **Discussion and conclusion** —What worked well? What didn't? If something did

not work as expected, what do you think is the reason? How could your solution be improved?

8 Lab grading criteria

- **FUNCTIONALITY (60%)**
  - Code does not compile .... **deduct 60%**
  - The simulator does not run (e.g, missing the .hex file) .... **deduct 60%**
  - The device does not work as intended .... **deduct 20%**
  - If the LED's bounce when the switch is pressed .... **deduct 20%**
  - Any missing/non-implemented specification .... **deduct 20%**
- **CODE QUALITY AND DOCUMENTATION(20%)**
  - Lack of comments, misleading or useless comments .... deduct 5%
  - Poor coding style .... deduct 5%
  - Poor code organization and modularization .... deduct 10%
  - Using magic numbers .... deduct 5%
  - Messy, unreadable Code .... deduct 5%
  - Violates major embedded C coding standard .... deduct 5%
  - Does not use proper naming convention .... deduct 5%
  - Does not use meaningfull variable names .... deduct 5%
  - Use God functions .... deduct 5%
- **LAB REPORT 20%**
  - abstract—conveys a sense of the full report concisely 2%
  - introduction—effectively presents the purposes of the lab 5%
  - circuit description—clear, concise but complete 5%
  - messy circuit diagrams .... deduct 2%
  - lack of justification of component selection (e.g., current limiting, internal or external pull-ups or pull downs) 5%
  - software description 5%
  - Conclusion 3%

- Any cheating or copying someone else's code .... **deduct 100%**
- **TOTAL** **100%**

9 Lab submission

The lab is due on **October 11, 2021**. The submission shall go as follows:

- You should submit the report, all your code file (c or cpp files), the proteus simulation file and the .hex file in one .zip file and submit them through the e-learning platform no later than midnight on October 11, 2021.
- Before submission, please make sure that, when the simulation file is properly linked with the .hex file. **I should be able to run your simulator without compiling your code.**
- Please submit your work before the deadline. I will not accept any submission through my email no matter the reasons