# Labo n° 2
# Real-Time systems [ELEC-H-410]
# Sharing resources under FreeRTOS

2018–2019

## Purpose

Throughout this second lab, you will learn how to manage the dependencies between tasks and how to share resources.

### Useful documentation:

- Official FreeRTOS documentation: `https://www.freertos.org/Documentation/RTOS_book.html`
- Getting Started with PSoC 5LP: `https://www.cypress.com/file/41436/download`
- Video example on how to use the PSoC: `https://www.cypress.com/video-library/PSoC`
- The extension board schematics: `Extension_PSoC.pdf`

## 1 Sharing resources

The simplest way to make two tasks communicate is to use structures of shared data: most of the time they are global variables. It is then necessary to establish a mechanism of protection to control the access to these variables.

Open the project `resource_sharing`.

**Exercise 1.**

- Analyse the code, knowing that each task is executed only **once** (`vTaskSuspend()` is called at the end of the *for* loop), what should be the value of the global variable `cntr` at the end of the execution?
- Compile and execute the code.
- Using a watch window and the debugger, check the value of `cntr`. Is this the expected result? Why? (Use the logic analyser to help)

The **mutex** (see chapter over priority-driven systems) is a convenient way to protect shared data, while still handling interrupts and authorise the preemption of the running task. Note: when a task acquires a resource via `xSemaphoreGive()`, it should not forget to release it via `xSemaphoreTake()`. A pseudo code is given in Listing 1.

Listing 1: Mutex declaration/use

```
// [...]
SemaphoreHandle_t myMutex; // declaration as global variable
myMutex = xSemaphoreCreateMutex(); // creation

void function() {
    // [...]
    xSemaphoreTake( myMutex, portMAX_DELAY );
    // operations on the shared resource ...
    xSemaphoreGive( myMutex );
    // [...]
}
```

The first 2 lines of Listing 1 create a mutex by:

- declaring a handler with the type `SemaphoreHandle_t`. This declaration must be global, so that the mutex is visible from anywhere in the program.

– creating the mutex.

For more details, see the official FreeRTOS documentation.

**Exercise 2.** Modify the code of the `ressource_sharing` project by using a mutex to obtain the expected result for `cntr`.

## 2 Deadlock problem

Improper use of the mutex (or of any other mechanism of synchronisation between tasks) can bring to a situation where no task can be executed anymore, this is called a *deadlock* situation.

Open the `Deadlock` project and launch its execution.

Observe using the logic analyser that no task is executed although the tasks should be periodic.

**Exercise 3.** Find the reason of this deadlock by drawing a time graph of the different calls to the mutexes

**Exercise 4.** Fix this problem using the parameter `xTicksToWait` of `xSemaphoreTake()`.

## 3 Priority inheritance

Mutexes in FreeRTOS uses a particular version of the priority inheritance. If a high priority task blocks while attempting to obtain a mutex that is currently held by a lower priority task, then the priority of the task holding the token is temporarily raised to that of the blocking task. This mechanism is designed to ensure the higher priority task is kept in the blocked state for the shortest time possible, and in so doing minimise the 'priority inversion' that has already occurred[1].

Open the project `prio_inheritance`.

**Exercise 5.** Observe the mechanism with the logic analyser.

## 4 Use of a semaphore

The semaphores (in a flag version) allows a synchronisation between tasks or between an interrupt service routine (ISR) and a task.

In the Listing 2, `Task1` immediately goes to the "waiting" state until another `Task2` has reached a certain point of its execution, then `Task1` resumes its execution.

Listing 2: Semaphore example

```
SemaphoreHandle_t startTask1;
startTask1 = xSemaphoreCreateBinary();

Task1(){
    for(;;){
        xSemaphoreTake( startTask1, portMAX_DELAY ); // Task 1 goes "waiting"
        // [...]
    }
}

Task2(){
    for(;;){
        xSemaphoreGive( startTask1 ); //Task1 goes to "ready"
        // [...]
    }
}
```

[1] for more details see `https://www.freertos.org/Real-time-embedded-RTOS-mutexes.html`

Open the project entitled `synchronisation`.

This project consists in decoding a sound from a waveform and send it to the jack output of the extension board (connect your own headphones !). It makes use of the Digital to Analogue (DAC) block embedded in the Analogue Block Array of the board. You may observe its configuration in the `TopDesign` tab.

The project contains two tasks:

- `decodeTask`, which decodes a sound by batches of 64 samples (long execution time). It uses the `decodeSoundFrame()` function.
- `dacTask` which reads the decoded samples and sends them at 1kHz to the DAC connected to the jack output of the extension board.

To ensure continuous streaming of the sound, two buffers of samples are being used (`buf1[]` and `buf[2]`). When one is being decoded by the `decodeTask` task, the other one is played with the `dacTask`. Of course this only works if both tasks are synchronised. A first semaphore was placed in this code so that the first samples are not sent to the DAC before they actually have been decoded. If you run the code you will see that this does not work.

**Exercise 6.** Modify the code so that a new batch of decoded samples is not written in one of the two buffers before the previous batch has been completely send to the DAC. You may need to add another semaphore.

# 5 Queues

Until now, we saw that the use of global variables made it possible to exchange information between two tasks. This method however presents some disadvantages among which the fact that a task is not automatically informed of a change of the variable. It is thus necessary that the task checks it periodically (polling) or that it announces any change via a mutex or semaphore.

To make this exchange easier, FreeRTOS implements another protocol: **queues**.

A task or an ISR can deposit a message in a queues. In a similar way, one or more tasks can receive a message in this queue.

Note: when we speak about "message" under FreeRTOS, it actually acts as any kind of structure copied in a reserved memory. The type of structure must obviously be known by the transmitting task as well as the receiving task(s).

If a task wishes to receive a message coming from a queue, it is suspended until the arrival of the message, or during a lapse of time defined by the application. Each queues is thus associated to a waiting list containing these suspended tasks. When the message is posted, it's the highest priority waiting task that receives the message[2].

Open the project `keyboard`.

This project contains two tasks :

- `keyboardTask` is the task which manages the keyboard: if a new key is pressed, the `key` variable contains the ASCII code of this key. The task is reactivated every 50ms to detect the actions of the user. It makes use of the `KB_Scan()` API of the block `keyboard_4x3`. This block was custom made for these labs and is part of the `extensionBoardLib` library. Make sure your project includes the library by right clicking on the project tab > Dependencies and verify that the extensionBoardLib is checked. Also, make sure that the keyboard is correctly connected to the extension board, see Appendix A.1.
- `lcdTask` displays a character on the LCD screen. Read the datasheet of the `Character LCD` block to understand how it interact with the LCD screen.

**Exercise 7.** Create a queue allowing to transmit the characters pressed on the keyboard to the task displaying the characters on the screen. The queue should only contain one message.



[2] for more details see `https://www.freertos.org/Embedded-RTOS-Queues.html`

# A Keyboard

## A.1 Electrical connections to the extension board

The keyboard should be connected to the bottom 7 pins starting from the left.