Lab n° 3
Real-Time systems [ELEC-H-410]
Realization of an application under FreeRTOS

2021–2022

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`
- Getting started with the Logic Analyzer: `https://learn.sparkfun.com/tutorials/using-the-usb-logic-analyzer-with-sigrok-pulseview`

1 Lab objective

During this lab, you will learn how to write a task under FreeRTOS, to make it periodic and to assign it a priority, in an intelligent way. The hardware will be composed of the PSoC and a logic analyser.

If you are not confident with C programming, read `C_language_for_uC.pdf`.

Principles of the logic analyser are explained in the chapter 9; as well as in Lab 1.

1.1 Creation of a task under FreeRTOS

A task is a succession of instructions doing a specific operation. Contrary to a function, a task cannot return a value. Moreover you do not have any direct influence on the order of execution of the various tasks you create. Indeed, it is the operating system which is given the responsibility to schedule the tasks and thus to choose which task must be carried out at which time on the processor. FreeRTOS is a preemptive RTOS based on fixed priorities that you assign to the tasks. The choice of those priorities is thus critical so that the system behaves as you wish. This is why the second part of this lab will be related to the wise choice of priorities.

First, you will learn how to create a single simple task in FreeRTOS and to initiate the execution of the operating system.

Copy the project **`\ELEC-H-410\lab3\ex1_blinky`** to your computer and open the project with `PSoC Creator`.
In the file main.c you will find the `main` function (see Listing 1) in which are executed :

- the initialisation of FreeRTOS: `freeRTOSInit()`
- the creation of a task : `xTaskCreate()`
- the starting of FreeRTOS: `vTaskStartScheduler()`

This structure cannot vary. The operating system must indeed be initialised before any creation of task and at least one task must have been created before giving control to OS.

For more details on the parameters sent during the creation of the task, refer to the Task Creation section of the documentation of FreeRTOS[1].

Listing 1: Function main.c

```
#include "project.h"
// [...]
```

[1] `https://www.freertos.org/a00125.html`

```
TaskHandle_t taskHandler;

int main(void)
{
    freeRTOSInit();

    // [...]

    // Task creation
    xTaskCreate( task,
                 TASK_NAME,
                 TASK_STACK_SIZE,
                 NULL,
                 TASK_PRIORITY,
                 &taskHandler );

    // [...]

    // Start FreeRTOS
    vTaskStartScheduler();

    for(;;)
    {
        // This line of code is unreachable
    }
}
```

1.2 How to write a task

Adding tasks to the OS is very easy:

- The task must be written like a function which returns nothing (`void`).
- The task must contain an infinite loop.
- A task must always call at least one of the services of FreeRTOS that will make the task "waiting" such as `vTaskDelay()` or `vTaskSuspend()`.

Since FreeRTOS is preemptive, the currently running task has got the highest priority among all "ready" tasks, hence if no event occurs (like an ISR making a higher priority task ready or the current task giving the control back to the scheduler) no other tasks will ever run.

Listing 2: Basic task example

```
void task1( void *data )
{
    // [...] init code, variable declaration

    for(;;)
    {
        // [...] task code
        vTaskDelay(10); // Ask the RTOS to put task1 in "waiting"
                        // state for at least 9 ticks
    }
}
```

1.3 Put a task to sleep for some time

Sometimes, it is necessary to let a task sleep for a while (maybe the job is complete, the task needs some resource/data not available yet to complete its job...)

To let a task sleep in "waiting" state for some time, one might call the `vTaskDelay(uint32_t ticks)` function; The parameter `ticks` is an unsigned 32bit integer[2] which determines the number of ticks during which the task will sleep. The timer creating the periodic interrupts has been configured for a frequency of 1 Hz, hence 1 tick = 1 ms. More precisely the task will sleep at least (`ticks-1`), if you want to be sure to sleep during 1 tick you should specify `ticks=2`. To demonstrate that, draw a chronogram of tick interrupts and imagine where the call `vTaskDelay()` could occur.

Exercise 1. Modify the task in the `blinky` project to lights a LED of the μC board at a frequency of 1 Hz. Then create a second task that will lights another LED at a frequency of 2 Hz.

To change the state of the the LED $D1$ you may use `LED1_Write(1u);`, this is an API[3] automatically generated by PSoC Creator. Peripherals are declared in the TopDesign tab. Check the datasheet of LED1 to see what other API you could use.

1.4 Creation of periodic tasks

In Exercise 1, you have created a periodic task, *i.e.* a task executing forever at regular intervals. In most industrial applications, those tasks are frequent and the periodicity should be realised with a good precision (see example of PI controller in chapter 3 of the course).

Open the project `Periodicity`.

You will find 4 tasks in this example:

- `task1` which should have a period of 10ms;
- `task2` which should have a period of 50ms;
- `task3` which should have a period of 100ms.

Exercise 2. Scheduling verification. We will use the logic analyser to verify if your task is scheduled correctly in the RTOS.

- Switch the logic analyser on and launch the display interface on the PC.
- Start your program `Periodicity` on the microcontroller and launch a first data acquisition with the logic analyser.
- Observe preemptions of certain tasks when a higher priority task is active (see signals `input1`, `input2` and `input3` whose value is 1 when the tasks `task1`, `task2` and `task3` are respectively active, *i.e.*, between its first instruction until its completion).
- Use the logic analyser to measure the real period of real activation of each task. Are they exactly in conformity with the desired periods? Identify 2 causes of these errors.

1.4.1 Use of `xTaskGetTickCount()`

FreeRTOS provides the `xTaskGetTickCount()` function which returns a 32 bit integer (`uint32_t`) representing the number of ticks since the launching of OS.

Exercise 3. Compute after how long this counter will overflow.

Exercise 4. Use `xTaskGetTickCount()` in each task to compensate for the error over the period.

1.4.2 Use of a software timer

It is possible to use software timers in FreeRTOS. Those are used exactly in the same way as hardware timers, except that they are entirely managed by the operating system and that they are synchronised on the ticks of the system. The function `xTimerCreate()` allows to create a software timer (see FreeRTOS documentation for details) and `xTimerStart()` to start it. When a timer expires, it calls a function whose pointer was given in the parameters.

[2] ranging from 0 to $2^{32} - 1$
[3] Application Programming Interface

Open the project `Timer`. You will find the same 4 tasks as in the previous example except that their period are generated by using three software timers.

Functions `vTaskSuspend()` and `vTaskResume()` allow to suspend and restart the execution of a specific task.

Exercise 5. Check with the logic analyser that the periods are strictly respected.

This method for creating periodic task gives very precise results. However, it is rather heavy and should therefore be used when this precision is absolutely required.

Exercise 6. Create a new timer which switches a LED on after 5s. There is no need to write a complete task for this exercise.

1.4.3 Choice of the priorities

As explained earlier, the choice of the priorities of the task is the only tool at our disposal to help the operating system to choose which task must be running at which time. To be convinced of the importance of a judicious choice of these priorities, we will look at a simple example.

Exercise 7. Open the project `Priorities`.

- The task `task1` should run every 1ms
- The task `task2` should run every 100ms
- Check the behaviour of the tasks with the logic analyser.
- Reverse the priorities of `AppTask1` and `AppTask2` and reverify what occurs.
- By comparing the periods of each task and the priorities assigned, which systematic rule of assignment can you deduce?
- How is called this method to assign the priorities?
- What happens when tasks have relative deadlines different from their periods?
- Which scheduling algorithm would you use if you could assign priorities directly to jobs instead of tasks?