

## TP de Noyau temps réel

# TP FreeRTOS

Application de FreeRTOS et de ses fonctionnalités  
pour la commande d'une STM32

Professeur encadrant :  
*Nicolas PAPAZOGLOU*



Ecole Nationale  
Supérieure  
de l'Electronique  
et de ses Applications

Année 2022 - 2023

# Table des matières

<b>Résumé</b>	<b>1</b>
1 (Re)prise en main . . . . .	2
1.1 Premiers pas . . . . .	2
2 FreeRTOS, tâches et sémaphores . . . . .	2
2.1 Tâche simple . . . . .	2
2.2 Sémaphores pour la synchronisation . . . . .	3
2.3 Notification . . . . .	4
2.4 Queues . . . . .	5
2.5 Réentrance et exclusion mutuelle . . . . .	5
3 On joue avec le Shell . . . . .	6
4 Debug, gestion d'erreur et statistiques . . . . .	8
4.1 Gestion du tas . . . . .	8

## Résumé

L'objectif de ce TP sur cinq séances est de mettre en place quelques applications sous FreeRTOS en utilisant la carte STM32f746-Discovery conçue autour du microcontrôleur STM32F746ng.

Pour implémenter le code des différentes applications de ce TP, nous utilisons le logiciel STM32CubeIDE. Notre code est accessible depuis notre GitHub, [https://github.com/ClementChapuis/TP\\_FreeRTOS](https://github.com/ClementChapuis/TP_FreeRTOS) et est actualisé régulièrement au fil des séances de TP.

# 1 (Re)prise en main

## 1.1 Premiers pas

Pour cette première application, nous commençons par suivre scrupuleusement les différentes étapes explicitées dans l'énoncé dans le but de créer notre nouveau projet.

Ensuite, nous répondons aux différentes questions qui nous sont posées :

- le fichier main.c se trouve dans le fichier Source
- les commentaires appelés BEGIN et END sont des balises entre lesquelles il nous faudra écrire notre code si nous voulons que celui-ci soit sauvegardé après une modification des périphériques, des timers, etc
- dans HAL\_Delay, il nous faut entrer comme paramètre un temps en ms. Dans HAL\_GPIO\_WritePin nous entrons les paramètres du port GPIO correspondant ainsi que le pin dans lequel nous voulons écrire et l'état du pin que nous souhaitons ( 0 ou 1)
- les ports d'entrées et de sortie sont définis dans le fichier gpio.c

Après cela, différentes manipulations nous sont demandées. Nous commençons par écrire un programme simple permettant de faire clignoter la LED. En effet, il y a plusieurs LED sur notre carte STM32F7, mais seule une d'entre elles nous est accessible : il s'agit de la LED connectée à la broche PI1.

Voici une capture de notre code :

```
HAL_GPIO_WritePin(pin_led_GPIO_Port, pin_led_Pin, SET);
HAL_Delay(1000);
HAL_GPIO_WritePin(pin_led_GPIO_Port, pin_led_Pin, RESET);
HAL_Delay(1000);
```

FIGURE 1 – Code pour le clignotement de la LED.

Nous souhaitons désormais modifier notre programme afin que la LED s'allume lorsque le bouton USER est appuyé. L'énoncé nous indique que ce dernier est connecté à la broche PI11. Voici une capture de ce nouveau code :

```
if(HAL_GPIO_ReadPin(blue_button_GPIO_Port, blue_button_Pin))
{
    HAL_GPIO_WritePin(pin_led_GPIO_Port, pin_led_Pin, SET);
}
HAL_GPIO_WritePin(pin_led_GPIO_Port, pin_led_Pin, RESET);
```

FIGURE 2 – Code amélioré avec appui bouton.

# 2 FreeRTOS, tâches et sémaphores

## 2.1 Tâche simple

Nous créons désormais un nouveau projet sur CubeIDE.

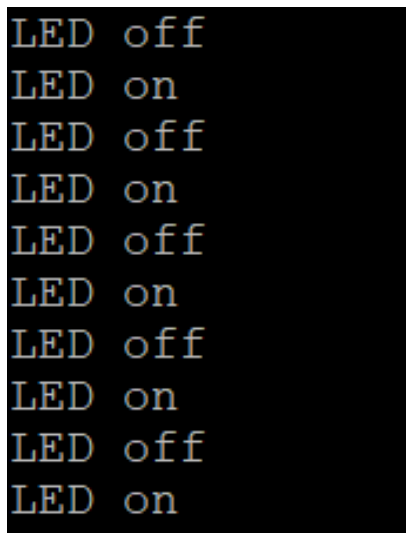
Il nous est demandé à quoi correspond le paramètre TOTAL\_HEAP\_SIZE : il s'agit de la taille totale de la pile, importante à considérer pour ne pas être à court de RAM.

Nous allons maintenant créer une tâche permettant de faire changer l'état de la LED toutes les 100ms. Voici le code que nous avons :

```
void task_led(void * unused)
{
    while(1)
    {
        HAL_GPIO_WritePin(LED_GPIO_Port, LED_Pin, SET);
        printf("LED on\r\n");
        vTaskDelay(delay);
        HAL_GPIO_WritePin(LED_GPIO_Port, LED_Pin, RESET);
        printf("LED off\r\n");
        vTaskDelay(delay);
    }
}
```

FIGURE 3

Et voici notre résultat dans la console :



```
LED off
LED on
LED off
LED on
LED off
LED on
LED off
LED on
LED off
LED on
```

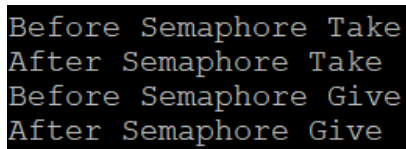
FIGURE 4

La macro `portTICK_PERIOD_MS` peut être utilisée pour déterminer le temps réel en ms à partir de la fréquence d'horloge.

A partir de ce moment, ayant réécrit et modifié plusieurs fois les mêmes tâches de ce projet, nous n'avons pas pris de capture du code. Notre GitHub contient le code que nous avons utilisé pour ces différentes applications. Nous n'avons choisi de vous montrer que les résultats.

## 2.2 Sémaphores pour la synchronisation

Nous créons ensuite deux tâches de priorité différente, `taskGive` et `taskTake`. `TaskGive` donne un sémaphore toutes les 100ms et `TaskTake` prend le sémaphore. Notre objectif est d'afficher du texte avant et après avoir pris le sémaphore. Nous commençons donc par créer nos deux tâches. En donnant la priorité la plus élevée à la tâche `taskTake`, voici ce que nous obtenons dans la console :



```
Before Semaphore Take
After Semaphore Take
Before Semaphore Give
After Semaphore Give
```

FIGURE 5

Et en inversant les priorités :

```
Before Semaphore Take
Before Semaphore Give
After Semaphore Take
```

FIGURE 6

Ensuite, nous mettons en place un mécanisme de gestion d'erreur lors de l'acquisition du sémaphore. L'idée est d'invoquer un reset software au STM32 si le sémaphore n'est pas acquis au bout d'une seconde. Puis, pour valider la gestion d'erreur, nous ajoutons 100ms au délai de TaskGive à chaque itération. Voici ce que cela donne pour la première configuration en terme de priorité des tâches :

```
Before Notify Take
Before Notify Give
After Notify Take
After Notify Give
delay 500

Before Notify Take
Before Notify Give
After Notify Take
After Notify Give
delay 600

Before Notify Take
Before Notify Give
After Notify Take
After Notify Give
delay 700

Before Notify Take
Before Notify Give
After Notify Take
After Notify Give
delay 800

Before Notify Take
Before Notify Give
After Notify Take
After Notify Give
delay 900

Before Notify Take
Before Notify Give
After Notify Take
After Notify Give
delay 1000
```

FIGURE 7

Nous pouvons remarquer sur cette capture que nous sommes déjà passé à l'utilisation de notifications ainsi que d'une queue. En effet, nous avons oublié de faire la capture avant ça.

## 2.3 Notification

Nous modifions notre code afin d'avoir le même fonctionnement que précédemment, mais avec des notifications :

```

After Notify Take
Before Notify Take
After Notify Give
Before Notify Give
After Notify Take
Before Notify Take
After Notify Give
Before Notify Give
After Notify Take

```

FIGURE 8

## 2.4 Queues

Maintenant, nous souhaitons réaliser une queue afin d’être en mesure de stocker et d’afficher la valeur du timer. Voici notre résultat après modification du code :

```

Before Notify Take
Before Notify Give
After Notify Take
After Notify Give
delay 500

Before Notify Take
Before Notify Give
After Notify Take
After Notify Give
delay 600

Before Notify Take
Before Notify Give
After Notify Take
After Notify Give
delay 700

Before Notify Take
Before Notify Give
After Notify Take
After Notify Give
delay 800

Before Notify Take
Before Notify Give
After Notify Take
After Notify Give
delay 900

Before Notify Take
Before Notify Give
After Notify Take
After Notify Give
delay 1000

```

FIGURE 9

## 2.5 Réentrance et exclusion mutuelle

Nous récupérons le projet voulu par l’énoncé, le compilons et l’exécutons. Nous remarquons cependant qu’au lieu d’avoir un enchaînement de “je suis la tâche1 ...” “je suis la tâche 2 ...”, les

lignes se superposent. Cela est dû au fait que la tâche 2 est plus prioritaire que la tâche 1. Il y a donc une préemption et la tâche 2 va continuer d'écrire alors que la tâche 1 n'a pas fini.

Pour pallier à ce problème, nous choisissons une solution utilisant un sémaphore Mutex. Voici notre code dans ces conditions :

```
void task1(void * pvParameters)
{
    int delay = (int) pvParameters;

    for(;;)
    {
        xSemaphoreTake(sem, portMAX_DELAY);
        printf("Je suis la tâche 1 et je m'endors pour %d ticks\r\n", delay);
        vTaskDelay(delay);
        xSemaphoreGive(sem);
    }
}

void task2(void * pvParameters)
{
    int delay = (int) pvParameters;

    for(;;)
    {
        xSemaphoreTake(sem, portMAX_DELAY);
        printf("Je suis la tâche 2 et je m'endors pour %d ticks\r\n", delay);
        vTaskDelay(delay);
        xSemaphoreGive(sem);
    }
}
```

FIGURE 10

Et voici ce que nous obtenons :

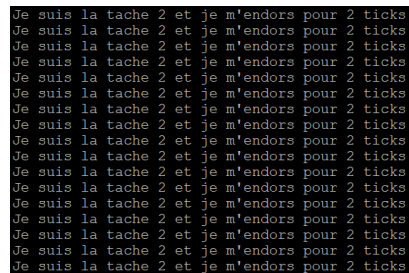


FIGURE 11

Nous avons un résultat qui n'est pas satisfaisant, il y a donc une erreur de programmation.

### 3 On joue avec le Shell

Dans cette nouvelle application, nous reprenons le shell sur lequel nous avons travaillé en TD.

L'énoncé nous indique qu'il y a cependant une subtilité :

*seules les interruptions dont la priorité est supérieure à la valeur configLIBRARY\_MAX\_SYSCALL\_INTERRUPT\_PRIORITY (définie à 5 par défaut) peuvent appeler des primitives de FreeRTOS. On peut soit modifier ce seuil, soit modifier la priorité de l'interruption de l'USART1 (0 par défaut). Dans l'exemple montré en Figure 1, la priorité de l'interruption de l'USART1 est fixée à 5.*

Si nous ne respectons pas ces priorités, le shell n'est plus utilisable.

Comme demandé dans l'énoncé, nous écrivons une fonction led(), callable depuis le shell, permettant de faire clignoter la LED (PI1 sur la carte). Un paramètre de cette fonction configure la



période de clignotement. Une valeur de 0 maintient la LED éteinte. Le clignotement de la LED s'effectue dans une tâche.

La difficulté réside dans le fait de faire communiquer \*proprement\* la fonction led avec la tâche de clignotement.

Pour cela, nous devons créer une tâche pour notre shell. Nous devons aussi modifier le fichier des drivers du shell afin d'appeler la tâche de celui-ci avec une sémaphore binaire. Par système d'interruption, cela nous permet de continuer à taper dans la console alors qu'une autre tâche est réalisée. Nous écrivons le code suivant pour la tâche du shell :

```
void shell(void * pvParameters)
{
    shell_init(&h_shell);
    shell_add(&h_shell, 'f', fonction, "Une fonction inutile");
    shell_add(&h_shell, 'l', led, "Clignotement d'une led");
    shell_add(&h_shell, 's', spam, "Fonction spam");
    shell_run(&h_shell);
}
```

FIGURE 12

et le code suivant dans le fichier des drivers :

```
uint8_t drv_uart1_receive(char * pData, uint16_t size)
{
    HAL_UART_Receive_IT(&huart1, (uint8_t*) (pData), size);
    xSemaphoreTake(sem, portMAX_DELAY);

    return 0;    // Life's too short for error management
}
```

FIGURE 13

```
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
{
    if(huart==&huart1)
    {
        static BaseType_t xHigherPriorityTaskWoken;
        xHigherPriorityTaskWoken = pdFALSE;
        xSemaphoreGiveFromISR(sem, &xHigherPriorityTaskWoken);
        portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
    }
}
```

FIGURE 14

Puis, nous créons une fonction led. Celle-ci fait appel à une tâche permettant de faire clignoter la led de notre STM par l'utilisation d'un autre sémaphore binaire. Voyez plutôt :

```

void clignotement(void * pvParameters)
{
    xSemaphoreTake(sem_led, portMAX_DELAY);
    while(1)
    {
        if(periode ==0)
        {
            HAL_GPIO_WritePin(LED_GPIO_Port, LED_Pin, RESET);
        }else{
            HAL_GPIO_TogglePin(LED_GPIO_Port, LED_Pin);
            vTaskDelay(periode);
        }
    }
}

void led(h_shell_t * h_shell, int argc, char ** argv)
{
    periode = atoi(argv[1]);
    if(periode !=0)
    {
        xSemaphoreGive(sem_led);
    }
}

```

FIGURE 15

Puis, il nous est demandé de réaliser une tâche spam, en se servant du même principe. Finalement, l'encadrant nous a indiqué que cette consigne était caduque, mais voici tout de même le code que nous avons réalisé pour cette nouvelle application :

```

void task_spam(){
    xSemaphoreTake(sem_spam, portMAX_DELAY);
    for(int i = 0; i <= repetition; i++)
    {
        printf("%s\r\n", msg);
        vTaskDelay(100);
    }
}

void spam(h_shell_t * h_shell, int argc, char ** argv)
{
    strcpy(msg, argv[1]);
    repetition = atoi(argv[2]);
    xSemaphoreGive(sem_spam);
}

```

FIGURE 16

## 4 Debug, gestion d'erreur et statistiques

### 4.1 Gestion du tas

La zone réservée à l'allocation dynamique s'appelle le tas. Celui-ci est géré par FreeRTOS.