

TP FREERTOS

0. (Re)prise en mai

0.1. Premiers pas

Créez un projet pour la carte STM32F746G-DISCO. À la fin de la création du projet, une pop-up vous demande "Initialize all peripherals with their default Mode ?" : Répondez "No".

Une fois le projet créé, dans l'onglet "Pinout & Configuration" :

- Configurez le quartz de l'horloge :
System Core > RCC > High Speed Clock (HSE) = Crystal/Ceramic Resonator
Les broches concernées sont les broches PH0 (OSC_IN) et PH1 (OSC_OUT).
- Configurez le système de programmation et de debug :
System Core > SYS > Debug = Serial Wire
Les broches concernées sont les broches PA14 (SYS_JTCK-SWCLK) et PA13 (SYS_JTMS-SWDIO).

(Notez l'utilisation des broches sur le microcontrôleur)

On réalise les différentes configurations dites dans l'énoncé.

0.1.1. Où se situe le fichier main.c ?

Le fichier main.c se situe dans les dossiers Core > Src > main.c.

0.1.2. À quoi servent les commentaires indiquant BEGIN et END (appelons ça des balises) ?

Les commentaires indiquant BEGIN et END nous permettant de savoir où écrire nos bouts de code. Si on l'écrit au mauvais endroit, le code sera effacé lors de la régénération du code : cela peut mener à de nombreux problèmes.

On pourrait compiler et programmer la carte, mais il ne se passerait rien de très palpitant.

Pour rendre ce TP plus excitant, faisons clignoter une LED ! Ce n'est pas écrit dans la doc, mais elle est connectée à la broche PI1. Deux fonctions à utiliser :

- HAL_Delay
- HAL_GPIO_WritePin

0.1.3. Quels sont les paramètres à passer à HAL_Delay et HAL_GPIO_WritePin ?

Maintenez Ctrl et cliquez sur le nom d'une fonction pour voir son code.

La fonction `HAL_Delay()` prend en paramètre un temps en milliseconde.

La fonction `HAL_GPIO_WritePin(GPIO_TypeDef * GPIOx, uint16_t GPIO_Pin, GPIO_PinState PinState)` prend en paramètre :

GPIOx: where x can be (A..G) to select the GPIO peripheral for STM32G4xx family

GPIO_Pin: specifies the port bit to be written. This parameter can be any combination of `GPIO_PIN_x` where x can be (0..15).

PinState: specifies the value to be written to the selected bit. This parameter can be one of the `GPIO_PinState` enum values:

- `GPIO_PIN_RESET`: to clear the port pin
- `GPIO_PIN_SET`: to set the port pin

0.1.4. Dans quel fichier les ports d'entrée/sorties sont-ils définis ?

Le fichier où les ports d'entrée/sorties sont définis est le fichier `gpio.c`.

0.1.5. Écrivez un programme simple permettant de faire clignoter la LED.

```
while (1)
{ HAL_GPIO_WritePin(LED_GPIO_Port, LED_Pin, GPIO_PIN_SET);
  HAL_Delay(1000);
  HAL_GPIO_WritePin(LED_GPIO_Port, LED_Pin, GPIO_PIN_RESET);
  HAL_Delay(1000);
  /* USER CODE END WHILE */

  /* USER CODE BEGIN 3 */
}
```

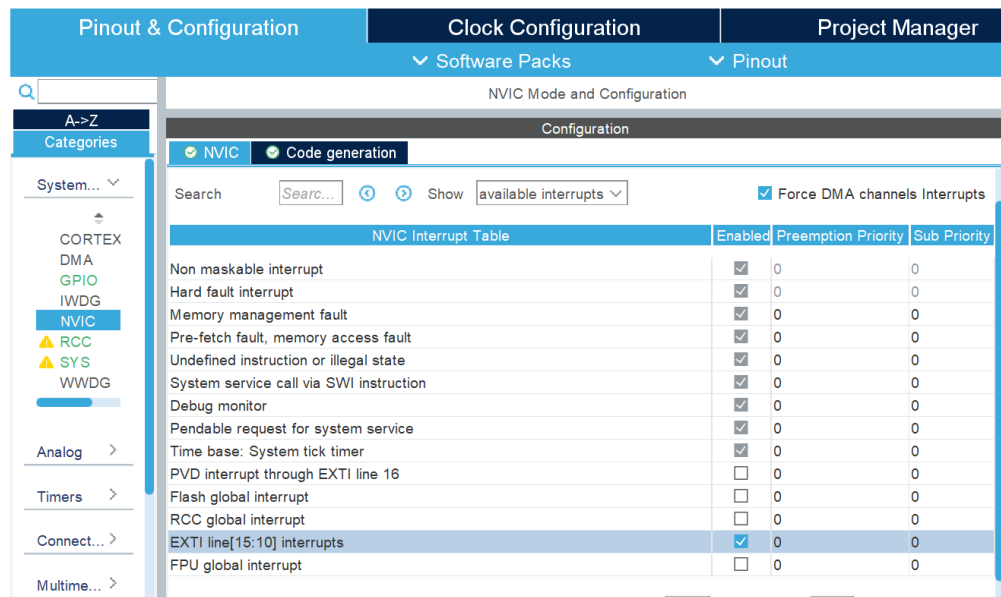
Nous avons utilisé la fonction `HAL_Delay()` car c'est ce qui est demandé dans le sujet mais ce n'est pas propre du tout. Le mieux serait de configurer des timers.

0.1.6. Modifiez le programme pour que la LED s'allume lorsque le bouton USER est appuyé. Le bouton est sur PI11.

On active le pin PI11 et on renomme le pin en `BLUE_BUTTON` pour que le code soit plus facile par la suite.



Puis on active les interruptions correspondantes dans le NVIC.



On sauvegarde et on régénère le code. On va utiliser la fonction `HAL_GPIO_EXTI_Callback` et la redéfinir dans notre fichier `main.c`. Cela va nous permettre d'exécuter la fonction lorsqu'on appuiera sur le bouton bleu.

```

44 /* USER CODE BEGIN PV */
45 int led_switch = 0;
46 /* USER CODE END PV */
47
48 /* Private function prototypes -----*/
49 void SystemClock_Config(void);
50 /* USER CODE BEGIN PFP */
51
52 /* USER CODE END PFP */
53
54 /* Private user code -----*/
55 /* USER CODE BEGIN 0 */
56 void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
57 {
58     if(GPIO_Pin==BLUE_BUTTON_Pin){
59         led_switch=1-led_switch;
60         if(led_switch){
61             HAL_GPIO_WritePin(LED_GPIO_Port, LED_Pin, GPIO_PIN_SET);
62         }
63         else{
64             HAL_GPIO_WritePin(LED_GPIO_Port, LED_Pin, GPIO_PIN_RESET);
65         }
66     }
67 }
68 /* USER CODE END 0 */

```

Grâce à ce code, la led s'allume lorsqu'on appuie sur le bouton bleu et elle s'éteint lorsqu'on appuie à nouveau dessus.

1. FreeRTOS, tâches et sémaphores

On crée le projet en s'aidant de la vidéo de l'énoncé.

1.1. Tâche simple

1.1.0. Créez un projet dans STM32CubeIDE. N'oubliez pas d'activer l'oscillateur externe, l'interface de debug ainsi que l'USART1. La fréquence de

fonctionnement sera configurée à 216MHz. L'interface de Debug sur Serial Wire et l'USART1 sur les broches PB7 et PA9. Activez FreeRTOS. En quoi le paramètre TOTAL_HEAP_SIZE a-t-il de l'importance ?

TOTAL_HEAP_SIZE correspond à la quantité de RAM disponible dans le noyau temps réel.

1.1.1. Observez l'impact de votre configuration sur le fichier FreeRTOSConfig.h.

```
/* Ensure definitions are only used by the compiler, and not by the assembler. */
#if defined(__ICCARM__) || defined(__CC_ARM) || defined(__GNUC__)
    #include <stdint.h>
    extern uint32_t SystemCoreClock;
#endif
#define configENABLE_FPU 0
#define configENABLE_MPU 0

#define configUSE_PREEMPTION 1
#define configSUPPORT_STATIC_ALLOCATION 1
#define configSUPPORT_DYNAMIC_ALLOCATION 1
#define configUSE_IDLE_HOOK 0
#define configUSE_TICK_HOOK 0
#define configCPU_CLOCK_HZ ( SystemCoreClock )
#define configTICK_RATE_HZ ((TickType_t)1000)
#define configMAX_PRIORITIES ( 7 )
#define configMINIMAL_STACK_SIZE ((uint16_t)128)
#define configTOTAL_HEAP_SIZE ((size_t)15360)
#define configMAX_TASK_NAME_LEN ( 16 )
#define configUSE_16_BIT_TICKS 0
#define configUSE_MUTEXES 1
#define configQUEUE_REGISTRY_SIZE 8
#define configUSE_PORT_OPTIMISED_TASK_SELECTION 1
/* USER CODE BEGIN MESSAGE_BUFFER_LENGTH_TYPE */
/* Defaults to size_t for backward compatibility, but can be changed
 * if lengths will always be less than the number of bytes in a size_t. */
#define configMESSAGE_BUFFER_LENGTH_TYPE size_t
/* USER CODE END MESSAGE_BUFFER_LENGTH_TYPE */
```

On retrouve le paramètre de TOTAL_HEAP_SIZE ainsi que des paramètres pour gérer la taille de la pile, des queues, la gestion d'allocation dynamique de mémoire ou encore l'utilisation des sémaphores mutexes.

1.1.2. Créez une tâche permettant de faire changer l'état de la LED toutes les 100ms et profitez-en pour afficher du texte à chaque changement d'état. Quel est le rôle de la macro portTICK_PERIOD_MS ?

```

void vTask_LED( void * unused)
{
    for( ;; )
    {
        // pour faire changer l'état de la LED
        HAL_GPIO_WritePin(GREEN_LED_GPIO_Port, GREEN_LED_Pin, SET);
        printf("La LED est la\r\n");
        // pour appeler la fonction dans 100 ms
        vTaskDelay(100);

        // pour faire changer l'état de la LED
        HAL_GPIO_WritePin(GREEN_LED_GPIO_Port, GREEN_LED_Pin, RESET);
        printf("La LED est pas la\r\n");
        // pour appeler la fonction dans 100 ms
        vTaskDelay(100);
    }
}

```

1.2. Sémaphores pour la synchronisation

1.2.1. Créez deux tâches, **taskGive** et **taskTake**, ayant deux priorités différentes. **TaskGive** donne un sémaphore toutes les 100ms. Affichez du texte avant et après avoir donné le sémaphore. **TaskTake** prend le sémaphore. Affichez du texte avant et après avoir pris le sémaphore.

```

je vais recevoir le semaphore
je vais donner le semaphore
j'ai donné le semaphore
j'ai reçu le semaphore

```

1.2.2. Ajoutez un mécanisme de gestion d'erreur lors de l'acquisition du sémaphore. On pourra par exemple invoquer un reset software au STM32 si le sémaphore n'est pas acquis au bout d'une seconde.

Le mécanisme de gestion d'erreur correspond au `NVIC_SystemReset()`.

1.2.3. Pour valider la gestion d'erreur, ajoutez 100ms au délai de TaskGive à chaque itération.

```

je vais recevoir le semaphore
je vais donner le semaphore
j'ai donné le semaphore
j'ai reçu le semaphore
delay de 700
je vais recevoir le semaphore
je vais donner le semaphore
j'ai donné le semaphore
j'ai reçu le semaphore
delay de 800

```

```
void TaskGive(void * unused){  
  
    for(;;)  
    {  
        printf("je vais donner le semaphore\r\n");  
        xSemaphoreGive(semaphore1);  
        printf("j'ai donné le semaphore\r\n");  
        vTaskDelay(delay);  
    }  
}  
  
void TaskTake(void * unused){  
  
    for(;;)  
    {  
        printf("je vais recevoir le semaphore\r\n");  
        if(xSemaphoreTake(semaphore1,Port_MAX_Delay))  
        {  
            printf("j'ai reçu le semaphore\r\n");  
        }  
        else  
        {  
            NVIC_SystemReset();  
        }  
        delay += 100;  
        printf("delay de %d \r\n",delay);  
    }  
}
```

Ici on observe la gestion d'erreur dans le taskTake, ainsi que les 100ms en plus ajouté au delay, ainsi lorsque le delay dépasse la valeur max (ici 1000), le processeur va reset et le programme va recommencer.

1.2.4. Changez les priorités. Expliquez les changements dans l'affichage.

1.3. Notification

1.3.1. Modifiez le code pour obtenir le même fonctionnement en utilisant des task notifications à la place des sémaphores.

Pour cette partie le code se trouve dans le document appelé **FREERTOSNotif**.

```
je vais recevoir le semaphore  
je vais donner le semaphore  
j'ai donné le semaphore  
j'ai reçu le semaphore  
delay de 700  
je vais recevoir le semaphore  
je vais donner le semaphore  
j'ai donné le semaphore  
j'ai reçu le semaphore  
delay de 800
```

1.4. Queues

1.4.1. Modifiez **TaskGive** pour envoyer dans une queue la valeur du timer. Modifiez **TaskTake** pour réceptionner et afficher cette valeur.

Avec la notification et la queue nous avons cela:

```
void task_give(void * unused){
    char msg[LONG_MAX];
    for(;;){
        printf("Before Notify Give\r\n");
        //xSemaphoreGive(sem1);
        xTaskNotifyGive(hTake);
        printf("After Notify Give\r\n");
        sprintf(msg, "delay %d \r\n",delay);
        xQueueSend(q_delay,(const void *)msg,portMAX_DELAY);

        vTaskDelay(delay);
    }
}

void task_take(void * unused){
    char delayBuff[LONG_MAX];
    for(;;){
        printf("Before Notify Take\r\n");
        /*if (xSemaphoreTake(sem1,PORT_MAX_DELAY)){
            printf("After Semaphore Take\r\n");*/
        if (ulTaskNotifyTake(pdTRUE, PORT_MAX_DELAY)){
            printf("After Notify Take\r\n");
        }
        else{
            NVIC_SystemReset();
        }
        delay += 100;
        xQueueReceive(q_delay, (void *)delayBuff, portMAX_DELAY);
        printf("%s \r\n",delayBuff);
    }
}
```

```
je vais recevoir le semaphore
je vais donner le semaphore
j'ai donné le semaphore
j'ai reçu le semaphore
delay de 300
je vais recevoir le semaphore
je vais donner le semaphore
j'ai donné le semaphore
j'ai reçu le semaphore
delay de 400
```

1.5. Réentrance et exclusion mutuelle

1.5.1. Récupérez le projet à l'adresse suivante : https://github.com/lfiack/tp_freertos_reentrance.

1.5.2. Importez le projet, compilez, exécutez.

1.5.3. Observez attentivement la sortie dans la console. Expliquez d'où vient le problème.

En observant la sortie dans le dossier on observe:

```
Je suis la tache 1 et je m'endors pour 2 ticks
Je suis la tache 2 et je m'endors pour 2 ticks
Je suis la tache 2 et je m'endors pour 2 ticks
Je suis la tache 1 et je m'endors pour 2 ticks
Je suis la tache 2 et je m'endors pour 2 ticks
Je suis la tache 2 et je m'endors pour 2 ticks
```

Cela vient du fait que la fonction printf prend trop de temps à s'exécuter. En effet, la fonction printf s'arrête en plein milieu de son exécution car elle est interrompue par la tâche 2 qui est plus prioritaire. Donc la tâche 2 s'exécute puis la tâche 1 reprend mais la valeur du buffer à changer. C'est pourquoi la fin du printf de la tâche 1 est identique à la fin du buffer de la tâche 2.

1.5.4. Proposez une solution en utilisant un sémaphore Mutex.

```
Je suis la tache 2 et je m'endors pour 2 ticks
Je suis la tache 1 et je m'endors pour 1 ticks
Je suis la tache 2 et je m'endors pour 2 ticks
Je suis la tache 1 et je m'endors pour 1 ticks
Je suis la tache 2 et je m'endors pour 2 ticks
Je suis la tache 1 et je m'endors pour 1 ticks
Je suis la tache 2 et je m'endors pour 2 ticks
```

En utilisant un sémaphore Mutex, le problème est réglé car on attend bien que la tâche en cours d'exécution se finisse avant de passer à la tâche suivante : en effet, le sémaphore étant déjà pris par la tâche 1, la tâche 2 doit attendre que la tâche 1 se finisse et rende son sémaphore pour pouvoir s'exécuter.

2. On joue avec le shell

2.1. Terminer l'intégration du shell commencé en TD.

Pour la suite du TP on utilisera le document appelé **correction** qui reprend le shell du td.

2.2. Que se passe-t-il si l'on ne respecte pas les priorités décrites précédemment ?

Si on ne respecte pas les priorités décrites précédemment, le shell sera plus prioritaire sur les tâches et elles ne s'exécuteront pas car le shell monopolisera l'exécution avec sa boucle "while" dans le shell_run : on ne sortira jamais de cette fonction.

2.3. Écrire 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. Il faut donc trouver un moyen de faire communiquer *proprement* la fonction led avec la tâche de clignotement.

On écrit une fonction led et une tâche taskedled qui nous permet de commander le clignotement de la LED via le shell.

```
> 1 50
:1 50
led clignote a une periode de 50 ms
> 1 0
:1 0
la led est eteinte
```

- 2.4. Écrire une fonction spam(), semblable à la fonction led() qui affiche du texte dans la liaison série au lieu de faire clignoter les LED. On peut ajouter comme argument le message à afficher et le nombre de valeurs à afficher. Ce genre de fonction peut être utile lorsque l'on travaille avec un capteur.

Nous avons essayé à plusieurs reprises de faire fonctionner la fonction et la tâche spam, mais il y a toujours une erreur qui apparaît. Nous avons donc décidé d'avancer plus dans le TP en passant cette question.

3. Gestion du tas

- 3.1. Quel est le nom de la zone réservée à l'allocation dynamique ?

Le tas est la zone réservée à l'allocation dynamique.



- 3.2. Est-ce géré par FreeRTOS ou la HAL ?

C'est géré par FreeRTOS.

- 3.3. Si ce n'est déjà fait, ajoutez de la gestion d'erreur sur toutes les fonctions pouvant générer des erreurs. En cas d'erreur, affichez un message et appelez la fonction Error_Handler();

On a utilisé la fonction "configAssert()" pour la gestion d'erreur.

- 3.4. Notez la mémoire RAM et Flash utilisée, comme dans l'exemple ci-dessous.

Memory Regions		Memory Details				
Region	Start address	End address	Size	Free	Used	Usage (%)
 RAM	0x20000000	0x20050000	320 KB	301,38 KB	18,62 KB	5,82%
 FLASH	0x08000000	0x08100000	1024 KB	999,8 KB	24,2 KB	2,36%

- 3.5. Créez des tâches bidons jusqu'à avoir une erreur.

- 3.6. Notez la nouvelle utilisation mémoire.
- 3.7. Dans CubeMX, augmentez la taille du tas (TOTAL_HEAP_SIZE). Générez le code, compilez et testez.
- 3.8. Notez la nouvelle utilisation mémoire. Expliquez les trois relevés.

On va répondre aux questions précédentes ici. On a pas réussi à faire une erreur, même en ajoutant plusieurs tâches, on a alors décidé de diminuer la valeur du tas TOTAL_HEAP_SIZE à une valeur inférieure à la valeur attribuée aux tâches.

 RAM	0x20000000	0x20050000	320 KB	315,76 KB	4,24 KB	1.33%
 FLASH	0x08000000	0x08100000	1024 KB	993,49 KB	30,51 KB	2.98%

Ainsi en réduisant la valeur de TOTAL_HEAP_SIZE, on alloue plus de place pour la RAM, d'où le pourcentage d'usage qui diminue. L'usage de la flash a augmenté grâce aux tâches bidons créées.