

TP FreeRTOS

Partie 0: Reprise en main

1. Le fichier main.c se situe dans les fichiers Core -> Src.
 2. Les balises servent de repères, si l'on ajoute du code en dehors de ces balises il sera supprimé si l'on utilise le .ioc pour générer du code.
 3. Le code des fonctions ressemble à ceci : `HAL_GPIO_WritePin(GPIOx, GPIO_Pin, PinState)` et `HAL_Delay(Delay)`.
- Il faut donc donner en paramètre un int à `HAL_Delay` qui représente la durée en millisecondes. Et il faut donner le port GPIO puis le Pin sur lequel on souhaite agir à la fonction `HAL_GPIO_WritePin` ainsi que l'état que l'on souhaite donné (1 ou 0).
4. Les ports d'entrées/sorties sont définis dans le fichier gpio.c qui se trouve au même endroit que le fichier main.c.

Partie 1: FreeRTOS, tâches et sémaphores

1. Le paramètre `TOTAL_HEAP_SIZE` représente la taille du tas. Il contient entre autre les piles des tâches il faut donc le choisir suffisamment grand pour qu'il ne soit pas rempli.
2. Pour faire clignoter la led toutes les 100ms on réalise le code suivant :

```
void vTaskLED( void * pvParameters )
{
    for( ;; )
    {
        HAL_GPIO_TogglePin(GPIOI, LED_Pin);
        printf("changement led");
        vTaskDelay(100);
    }
}
```

Le rôle de la macro `portTICK_PERIOD_MS` qui se trouve en paramètre de `vTaskDelay` est de définir le nombre de Tick durant lequel la tâche va s'endormir (ici 1 tick = 1ms).

3. On réalise le code suivant :

```
void vTaskTake( void * pvParameters )
{
    for( ;; )
    {
        printf("Task va prendre le semaphore\r\n");
        xSemaphoreTake(xSemaphore1 ,portMAX_DELAY );
        printf("Semaphore pris: \r\n");
    }
}

void vTaskGive( void * pvParameters )
{
    for( ;; )
    {
        printf("Task va donner le semaphore\r\n");

        xSemaphoreGive( xSemaphore1 );
        printf("Semaphore donné\r\n");
        vTaskDelay(100);
    }
}
```

TaskGive a une priorité de 3 et TaskTake une priorité de 2. On obtient alors ce résultat :

```
Task va prendre le semaphore
Task va donner le semaphore
Semaphore donné
Semaphore pris:
Task va prendre le semaphore
```

C'est le résultat après la première itération. TaskTake veut prendre le sémaphore puis elle bloque en attendant que TaskGive donne le sémaphore qui s'endort ensuite pour 100ms on revient alors à TaskTake qui affiche qu'elle a pris le sémaphore puis rebloque en attendant le prochain. Pour le premier cycle, c'est TaskGive qui est prioritaire on aurait donc : Task va donner -> Sem donné -> Task va prendre -> Sem pris ; puis on arriverait au résultat ci-contre.

4. Pour ajouter un mécanisme de gestion d'erreur il faut réaliser le code suivant :

```
void vTaskTake( void * pvParameters )
{
    for( ;; )
    {
        printf("Task va prendre le semaphore\r\n");
        if(xSemaphoreTake(xSemaphore1,1000) == pdTRUE)
        {
            printf("Semaphore pris: \r\n");
        }
        else{
            printf("reset\r\n");
            NVIC_SystemReset();
            i=0;
        }
    }
}

void vTaskGive( void * pvParameters )
{
    for( ;; )
    {
        printf("Task va donner le semaphore\r\n");

        xSemaphoreGive( xSemaphore1 );
        printf("Semaphore donné\r\n");
        printf("i= %d",i);
        vTaskDelay(i);
        i=i+100;
    }
}
```

On met à 1000 la valeur du paramètre *xTicksToWait* de la fonction *xSemaphoreTake* et on test si elle nous renvoie bien *pdTRUE*. Ce qui veut dire que si la tâche attend plus de 1000ms le semaphore on n'obtiendra pas *pdTRUE* et on réalisera un *SystemReset*. Dans *TaskGive* on augmente le délai de 100ms à chaque cycle. Et voici ce que l'on obtient :

```
Task va prendre le semaphore
Task va donner le semaphore
Semaphore donné
i= 1100Semaphore pris:
Task va prendre le semaphore
reset
Task va donner le semaphore
Semaphore donné
i= 100Task va prendre le semaphore
Semaphore pris:
Task va prendre le semaphore
Task va donner le semaphore
Semaphore donné
i= 200Semaphore pris:
Task va prendre le semaphore
Task va donner le semaphore
```

Lorsque le délai *i=1100ms* et que l'on atteint le *xSemaphoreTake* on effectue bien un reset puis on retrouve le comportement expliqué précédemment. *(Je ne sais pas pourquoi mais si je rajoute \r\n dans le printf du i le code ne fonctionne plus).*

6. Si on échange les priorités voilà ce qui s'affiche :

```
Task va prendre le semaphore
Task va donner le semaphore
Semaphore pris:
Task va prendre le semaphore
Semaphore donné
i= 100Task va donner le semaphore
Semaphore pris:
Task va prendre le semaphore
Semaphore donné
i= 200Task va donner le semaphore
Semaphore pris:
Task va prendre le semaphore
Semaphore donné
```

On a deux fois *Task va prendre le semaphore* avant que *Semaphore donné* s'affiche. C'est normal puisque *TaskTake* étant maintenant prioritaire elle reprend la main dès que la ligne *xSemaphoreGive* est réalisée et effectue donc une nouvelle boucle avant de rendre la main à *TaskGive* qui reprend à la ligne *Semaphore donné*.

7. Pour utiliser des notifications à la place des sémaphores il faut simplement remplacer les lignes xSemaphoreTake/Give par leurs équivalents avec des notifications :

```
void vTaskTake( void * pvParameters )
{
    for( ;; )
    {
        printf("Task va prendre le semaphore\r\n");
        //if(xSemaphoreTake(xSemaphore1 ,1000) == pdTRUE)
        if(ulTaskNotifyTake(pdTRUE, 1000) == pdTRUE)
        {
            printf("Semaphore pris: \r\n");
        }
        else{
            printf("reset\r\n");
            NVIC_SystemReset();
            i=0;
        }
    }
}

void vTaskGive( void * pvParameters )
{
    for( ;; )
    {
        printf("Task va donner le semaphore\r\n");
        //xSemaphoreGive( xSemaphore1 );
        xTaskNotifyGive(xHandle3);
        printf("Semaphore donné\r\n");
        printf("i= %d",i);
        vTaskDelay(i);
        i=i+100;
    }
}
```

On a alors exactement le même comportement que précédemment. Il faut cependant maintenant déclarer le xHandle3 (qui est le Handle de TaskTake) avant le code des tâches et non plus au début de la boucle main.

8. Pour utiliser des queues il faut réaliser le code suivant :

```
void vTaskTake( void * pvParameters )
{
    for( ;; )
    {
        char RxTimerBuff[100];
        printf("Task va prendre le semaphore\r\n");
        //if(xSemaphoreTake(xSemaphore1 ,1000) == pdTRUE)
        if(ulTaskNotifyTake(pdTRUE, 1000) == pdTRUE)
        {
            printf("Semaphore pris: \r\n");
            if(xQueueReceive(xQueue1,(void*)RxTimerBuff, (TickType_t)5))
            {
                printf("timer: %s\r\n", RxTimerBuff);
            }
        }
        else{
            printf("reset\r\n");
            NVIC_SystemReset();
            i=0;
        }
    }
}
```

```
void vTaskGive( void * pvParameters )
{
    char TxTimerBuff[100];
    xQueue1 = xQueueCreate(10, sizeof(TxTimerBuff));
    for( ;; )
    {
        printf("Task va donner le semaphore\r\n");
        //xSemaphoreGive( xSemaphore1 );
        sprintf(TxTimerBuff, "%d",i);
        xQueueSend(xQueue1, (void*) TxTimerBuff,(TickType_t) 0);

        xTaskNotifyGive(xHandle3);
        printf("Semaphore donné\r\n");
        // printf("i= %d",i);
        vTaskDelay(i);
        i=i+100;
    }
}
```

On obtient alors toujours le même résultat mais cette fois ci l'affichage de la valeur du délai ce fait dans la tâche TaskTake :

```
Task va prendre le semaphore
Task va donner le semaphore
Semaphore donné
Semaphore pris:
timer: 500
Task va prendre le semaphore
Task va donner le semaphore
Semaphore donné
Semaphore pris:
timer: 600
Task va prendre le semaphore
```

PARTIE 2 : Shell

1.2. Pour faire apparaitre la liste des arguments il suffit d'ajouter une boucle for :

```
for (int i = 0 ; i < argc ; i++)
{
    printf("arg %d = %s\r\n", i, argv[i]);
}
```

```
> f je suis un test
:f je suis un test
Je suis une fonction bidon
argc = 5
arg 0 = f
arg 1 = je
arg 2 = suis
arg 3 = un
arg 4 = test
```

1.3/4/5. La fonction est exécuté à partir de la tâche vTaskShell qui va dans un premier temps initialisé le shell et afficher le message de départ (==== Monsieur Shell v0.2 ====) puis cette tache ajoute la fonction au shell. On a ensuite la fonction shell_run qui lancera le shell. On peut alors utiliser la fonction en tapant le caractère f suivi ce que l'on veut afficher, il faut ensuite appuyer sur la touche entrée pour transmettre le message via l'uart. Cette méthode pose un problème puisque le processeur doit continuellement regarder si il reçoit quelque chose via l'uart (méthode de pulling) on ne peut donc pas réaliser d'autre taches. Pour pallier à ce problème, il faut utiliser un sémaphore binaire qui sera donné dans la boucle d'interruption de l'uart.


```

void USART1_IRQHandler(void)
{
    /* USER CODE BEGIN USART1_IRQn 0 */
    BaseType_t xHigherPriorityTaskWoken;
    xHigherPriorityTaskWoken = pdFALSE;
    xSemaphoreGiveFromISR(xSemaphoreUart, &xHigherPriorityTaskWoken);
    portYIELD_FROM_ISR(xHigherPriorityTaskWoken);

    /* USER CODE END USART1_IRQn 0 */
    HAL_UART_IRQHandler(&huart1);
    /* USER CODE BEGIN USART1_IRQn 1 */

    /* USER CODE END USART1_IRQn 1 */
}

static char uart_read() {
    char c;

    HAL_UART_Receive_IT(&UART_DEVICE, (uint8_t*)&c, 1);
    xSemaphoreTake(xSemaphoreUart, portMAX_DELAY);
    return c;
}

```

Avec l'ajout de ce sémaphore, la fonction `uart_read` du shell n'est plus bloquante. Elle est maintenant réalisée uniquement lorsque l'on transmet un message, d'autres tâches peuvent donc maintenant être exécutées par le processeur.

2. Si on ne respecte pas les priorités décrites précédemment, on ne pourra pas utiliser les primitives de FreeRTOS dans l'interruption de l'uart (`xSemaphoreGiveFromISR` par exemple).

3. On crée la fonction `led()` comme suit :

```

int fonctionLed(int argc, char ** argv)
{
    argT=atoi(argv[1]);
    printf("fonction led\r\n");
    printf("argc = %d\r\n", argc);
    printf("argT = %d\r\n", argT);
    if(argT==0){
        HAL_GPIO_WritePin(GPIOI, LED_Pin, RESET);
    }
    else {
        xSemaphoreGive(xSemaphoreLed);
    }
    return 0;
}

```

On récupère alors la valeur de la période dans la variable globale `argT`, si elle est égale à 0 on éteint la led sinon on donne un sémaphore binaire qui permettra d'exécuter la tâche suivante :

```
void vTaskLed( void * pvParameters )
{
    xSemaphoreTake(xSemaphoreLed ,portMAX_DELAY);
    for(;;){
        if(argT==0){HAL_GPIO_WritePin(GPIOI, LED_Pin, RESET);}
        else{
            HAL_GPIO_TogglePin(GPIOI, LED_Pin);
            vTaskDelay((1000/argT)/2);
            HAL_GPIO_TogglePin(GPIOI, LED_Pin);
            vTaskDelay((1000/argT)/2);
        }
    }
}
```

Une fois le sémaphore obtenue, on test la valeur de argT et on éteint la led si elle est égale à 0 (c'est le même test que dans la fonction mais il ne marche dans cette dernière, je ne sais pas pourquoi) et sinon on fait clignoter la led avec la période souhaité grâce à des vTaskDelay.

4. Pour la fonction spam on reprend le même principe qu'avant.

```
int fonctionSpam(int argc, char ** argv)
{
    mot=argv[1];
    nombre=atoi(argv[2]);
    printf("fonction Spam: mot nombre\r\n");
    xSemaphoreGive(xSemaphoreSpam);
    return 0;
}
```

On crée une fonction spam qui récupère les arguments et les enregistre dans les variables globales *mot* et *nombre*. Elle donne ensuite un sémaphore binaire qui permettra de réveiller la tâche.

```
void vTaskShell( void * pvParameters )
{
    shell_init();
    shell_add('f', fonction, "Une fonction inutile");
    shell_add('l',fonctionLed,"fonction led");
    shell_add('s',fonctionSpam,"fonction spam");
    shell_run();
}
```

Puis on ajoute la fonction dans le shell, elle sera donc appelée par le caractère s qu'il faudra faire suivre du mot que l'on veut spammer et du nombre de fois à l'afficher.

```
void vTaskSpam( void * pvParameters )
{
    xSemaphoreTake(xSemaphoreSpam ,portMAX_DELAY);
    int i;
    for(i=0;i<nombre;i++){
        printf("%s\r\n",mot);
        vTaskDelay(100);
    }
}
```

Une fois le sémaphore reçu par la tâche elle spam la console avec le mot choisi.

```
===== Monsieur Shell v0.2 =====
> s bonjour 5
:s bonjour 5
fonction Spam: mot nombre
> bonjour
bonjour
bonjour
bonjour
bonjour
```

Une fois que la tâche c'est exécuté le shell plante et on ne peut plus écrire via la liaison uart. Si on utilise le debugger on peut voir où le code s'arrête et on obtient :

```
static void prvTaskExitError( void )
{
    volatile uint32_t ulDummy = 0;

    /* A function that implements a task must not exit or attempt to return to
    its caller as there is nothing to return to. If a task wants to exit it
    should instead call vTaskDelete( NULL ).

    Artificially force an assert() to be triggered if configASSERT() is
    defined, then stop here so application writers can catch the error. */
    configASSERT( uxCriticalNesting == ~0UL );
    portDISABLE_INTERRUPTS();
```

Selon ce qui est expliqué ici, il faudrait terminer la fonction par un vTaskDelete, il faudrait alors remplacer le xSemaphoreGive dans la fonction spam par un xTaskCreate qui recréerait la tâche à chaque fois qu'on utilise la fonction. Je n'ai pas eu le temps de tester cette solution.



Partie 3 : Debug, gestion d'erreur et statistiques

1/2. La zone réservée à l'allocation dynamique est le tas (heap) qui est géré par FreeRTOS.

4/5/6/7/8.

Si on crée des tâches bidon on constate que l'utilisation de la mémoire augmente très légèrement (j'ai oublié de faire un screen avant mais en ajoutant 10 tâches on augmente de à peine de 0.2% je n'ai donc pas réussi à avoir une erreur):

rtos_td_shell.elf - /rtos_td_shell/Debug - 13 janv. 2023 à 15:52:35

Memory Regions		Memory Details				
Region	Start address	End address	Size	Free	Used	Usage (%)
 RAM	0x20000000	0x20050000	320 KB	301,35 KB	18,65 KB	5,83%
 FLASH	0x08000000	0x08100000	1024 KB	990,68 KB	33,32 KB	3,25%

Si on augmente la taille du TOTAL_HEAP_SIZE cette fois ci on constate un vrai changement :

rtos_td_shell.elf - /rtos_td_shell/Debug - Jan 13, 2023, 4:31:58 PM

Memory Regions		Memory Details				
Region	Start address	End address	Size	Free	Used	Usage (%)
RAM	0x20000000	0x20050000	320 KB	188,35 KB	131,65 KB	41.14%
FLASH	0x08000000	0x08100000	1024 KB	989,25 KB	34,75 KB	3.39%

Cela veut donc dire que le tas est stocké dans la mémoire RAM. Si on diminue la taille du tas de telle sorte qu'il soit plein je n'obtiens toujours pas d'erreur mais dans le .ioc on remarque bien une erreur



The screenshot shows the FreeRTOS Heap Usage interface. At the top, there are several tabs: 'Events', 'FreeRTOS Heap Usage' (selected), 'Tasks and Queues', 'Timers and Semaphores', 'Mutexes', 'Advanced settings', 'User Constants', 'Config parameters', and 'Include parameters'. Below the tabs, there is a 'Summary' section. It displays 'HEAP STILL AVAILABLE' as '0 Bytes' and 'TOTAL HEAP USED' as '624 Bytes'.

3.2 Gestion des piles

Si on essaye de remplir la pile d'une tâche avec le code suivant :

```
void vApplicationStackOverflowHook( TaskHandle_t xTask,
                                    signed char *pcTaskName ){
    printf("stack_overflow\r\n");
}

void vTaskBidon1( void * pvParameters )
{
    int a[100000000];
}
```

La fonction vApplicationStackOverflowHook devrait nous afficher un message d'erreur. Ce n'est malheureusement pas le cas et je ne sais pas pourquoi, mais le shell ne fonctionne plus ce qui prouve bien qu'il y a une erreur.