TP Noyaux Temps Réel

Partie 0

1. On trouve le main.c dans le fichier source.
2. BEGIN et END correspondent à des balises où il nous faudra écrire notre code (car sinon il ne sera pas sauvegardé après un modification des périphériques, timers, …).
3. Nous devons entrer dans HAL_Delay un paramètre de temps en ms. Ensuite on mettra les paramètres du port GPIO dans HAL_GPIO_Write_PIN: le pin ainsi que l'état dans lequel on souhaite le mettre (0 ou 1).
4. Ils sont définis dans le fichier gpio.c.

Partie 1

1. TOTAL_HEAP_SIZE défini la taille totale de la pile.
2. Nous définissons une tâche qui s'endort périodiquement. On créé donc la fonction CodeTache ci-dessous :

```
void CodeTache2 (void* pvParameters){
    int duree = (int) pvParameters;
    char* s = pcTaskGetName(xTaskGetCurrentTaskHandle());

    while(1){

        printf("Je suis la tache %s et je m'endors pour %d periodes \r\n", s, duree);
        vTaskDelay(duree);
```

Une fois exécutée elle va s'endormir pendant 2000 ticks ce qui correspond à 2 secondes. Nous observons le résultat suivant:

```
shell (CONNECTED)
Je suis la tache Task2 et je m'endors pour 2000 periodes
Je suis la tache Task2 et je m'endors pour 2000 periodes
Je suis la tache Task2 et je m'endors pour 2000 periodes
Je suis la tache Task2 et je m'endors pour 2000 periodes
Je suis la tache Task2 et je m'endors pour 2000 periodes
Je suis la tache Task2 et je m'endors pour 2000 periodes
Je suis la tache Task2 et je m'endors pour 2000 periodes
```

3. Nous voulons avoir deux tâches dont l'une qui s'endort périodiquement et qui devra débloquer l'autre à son réveil. Pour se faire, nous créons deux tâches en plus d'un sémaphore binaire qui nous permettra de synchroniser le tout.

```
xReturned = xTaskCreate(codeTache1, "Task1",STACK_SIZE, (void*) DELAY_1, 1, &xHandle1);
xReturned = xTaskCreate(codeTache2, "Task2",STACK_SIZE, (void*) DELAY_2, 1, &xHandle2);
```

```
void codeTache1(void* pvParameters){
    while(1){
        xSemaphoreTake(sem,portMAX_DELAY);
        printf("Tache 1 \r\n");
    }

    }

void codeTache2(void* pvParameters){

    while(1){
        xSemaphoreGive(sem);
        printf("Tache 2 \r\n");
        vTaskDelay(DELAY_1);
```

Lorsqu'elle s'exécute, la tâche 1 voudra prendre le sémaphore si ce dernier est plein, ainsi elle s'exécutera. Si ce n'est pas le cas, il faudra attendre que la tâche 2 s'exécute pour donner le sémaphore. Lorsque la tâche 2 s'exécute, elle donne le sémaphore de manière périodique ce qui synchronise la tâche 1.

```
l (CONNECTED)
Tache 1
Tache 2
Tache 1
Tache 2
Tache 1
Tache 2
Tache 1
Tache 2
```

Les commandes printf peuvent être trop longues: en effet il faut un temps non négligeable à la carte pour exécuter cette commande, les deux phrases peuvent donc se superposer (ce problème est résolu Q4). Nous n'avons donc affiché dans la console que la "Tache 1" et "Tache 2".

4. Dans ce code on a deux tâches qui sont créés avec les code pour chaque tâches suivants :

```
void vTask1(void* pvParameters){
    int delay = (int) pvParameters;

    while(1) {
        printf("Je suis la tache 1 et je m'endors pour %d ticks\r\n", delay);
        vTaskDelay(delay);
    }
}

void vTask2(void * pvParameters){
    int delay = (int) pvParameters;

    while(1) {
        printf("Je suis la tache 2 et je m'endors pour %d ticks\r\n", delay);
        vTaskDelay(delay);
    }
}
```

Une fois le code exécuté on obtient ceci :

```
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
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
```

On note que les lignes “je suis la tâche1 …” “je suis la tâche 2 …ne s’alternent pas correctement et se superposent. En effet la tâche 1 est moins prioritaire que la tâche 2 ce qui déclenche une préemption; or la tâche 2 va continuer d’écrire bien que la tâche 1 n'a pas fini. Pour remédier à ce problème, nous allons utiliser un sémaphore pour réaliser une exclusion mutuelles : On va modifier les fonctions vTask1 et vTask2 de la manière suivante :

```
semMutex = xSemaphoreCreateMutex();
```

```
void vTask1(void * pvParameters) {
    int delay = (int) pvParameters;

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

void vTask2(void * pvParameters) {
    int delay = (int) pvParameters;

    while(1) {
        xSemaphoreTake(semMutex, portMAX_DELAY);
        printf("Je suis la tache 2 et je m'endors pour %d ticks\r\n", delay);
        xSemaphoreGive(semMutex);
        vTaskDelay(delay);
```

Chaque tâche doit acquérir le sémaphore d'exclusion mutuelle avant d'exécuter le printf et libérer le sémaphore une fois terminée. Ainsi, lorsque la tâche 1 s'exécute, elle bloque le sémaphore, empêchant ainsi la tâche 2 de continuer tant que le printf de la tâche 1 n'est pas terminé.

Nous testons ensuite le nouveau code :

```
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
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
```

Le problème est résolu: les tâches ne se préemptent plus.

5. On lit dans la documentation Freertos que débloquer une tâche RTOS avec une notification est beaucoup plus rapide et efficace que de débloquer une tâche avec un sémaphore binaire Unblocking.

Partie 2

Nous créons le projet et le compilons.
Nous essayons de récupérer les arguments de la fonction.Nous comprenons que “argc” permet de savoir combien d'arguments sont donnés en entrée et “argv” est un pointeur de pointeur nous donnant accès aux valeurs d’entrée de la fonction. On code donc:

```
int fonction(int argc, char ** argv) {
    printf("Fonction bidon\r\n");

    // TODO: Ajouter l'affichage des arguments
    printf("argc = %d\r\n",argc);
    for (int i=0; i<argc; i++){
        printf("argument numero %d = %s\r\n",i, argv[i]);
    }

    return 0;
```

On note qu'un seul printf permet de récupérer la valeur de argc, cependant afin de récupérer toutes les données d’entrée il faut parcourir l’ensemble des valeurs de argv pour.
Une fois notre solution codée, nous avons pu la tester pour vérifier qu’elle fonctionnait comme nous le souhaitions.

```
Fonction bidon
argc = 6
argument numero 0 = f
argument numero 1 = Mais
argument numero 2 = ou
argument numero 3 = est
argument numero 4 = Lary
argument numero 5 = ?
>
```

3. On part du main.c qui fait ses initialisations habituelles, puis elle crée une tâche (liée au code **vTaskshell**).
Une fois la tâche créée :
On entre dans la fonction Init qui fait un printf (affichant “===== *Monsieur Shell v0.2* =====”) pour nous permettre de repérer facilement le début des lignes qui vont être reçues. Et qui crée une fonction **help**, décrivant toutes les autres (elle y compris).

On a ensuite une fonction **shell_add**. Dans cette fonction on commence par s’assurer que notre itérateur de tableau (qui est en variable globale) est en dessous d’un certain seuil. Le table en question est décrite juste après, **shell_func_list**. Ce tableau est d’une taille prédéfinie (à 64), on va pouvoir ainsi écrire 64 fonctions, et si on dépasse on déclenche une erreur.

Enfin, la fonction shell_run est l'endroit où se déroule la majeure partie du code. Elle est constituée d'une boucle infinie où l'on lit d'abord une valeur fournie. Si cette valeur est la touche 'ENTER', on effectue un retour à la ligne (/r). Si c'est la touche 'RETOUR ARRIÈRE', on effectue un retour en arrière (/b). Dans le cas contraire, il y a une erreur pour les autres caractères, à condition qu'il y ait encore de la place. Cela permet de créer un tampon qui contiendra tout ce qui a été saisi. Ce tampon est ensuite transmis à la fonction shell_exec. Dans cette fonction, on parcourt tous les éléments de la table (dont on connaît déjà le nombre) et pour chaque fonction de la table, on vérifie si le caractère transmis correspond. Si la fonction n'est pas trouvée, on affiche "c: no such command", sinon, on construit les argc et argv et l'appel de la fonction est effectué via la commande "return shell_func_list[i].func(argc, argv);" où l'on appelle la fonction et lui donne les paramètres créés.

4. La méthode présente un inconvénient car elle implique que le processeur vérifie constamment s'il a reçu quelque chose, ce qui entraîne une utilisation élevée de la puissance de traitement (en raison de la fonction UART_Receive). La fonction UART_READ est bloquante car elle boucle en continu, ce qui empêche les tâches moins importantes de s'exécuter.

5. Pour contourner cette difficulté, il est nécessaire d'utiliser une méthode différente en utilisant un sémaphore binaire et la fonction HAL_UART_ReceiveIT plutôt que HAL_UART_Receive. Il faut d'abord définir le sémaphore et le donner lors de la réception de données sur l'USART:

```
char uart_read() {
    char c;
    HAL_UART_Receive_IT(&huart1, (uint8_t*)(&c), sizeof(char));
    xSemaphoreTake(sem1,HAL_MAX_DELAY);
    return c;
```

En utilisant cette approche, lorsque l'on envoie des données à la console, l'interruption associée à l'UART se déclenche et appelle la fonction correspondante, qui donne le sémaphore. Ensuite, on retourne à la fonction uart_read. Ainsi, on récupère le sémaphore et on renvoie le caractère qui vient d'être envoyé.

Notre objectif est de créer une fonction led() qui permette de faire clignoter la LED à la fréquence souhaitée. Pour ce faire, on écrit une fonction led qui prend en compte l'argument lié à la fréquence, et utilise la fonction atoi pour convertir le caractère en un nombre entier. La fonction se présente donc comme suit :

```
int led(int argc,char ** argv){
    int f=atoi(argv[1]); // On récupère la fréquence souhaitée
    if (xHandle!=NULL ) { // Si une tache est déja écrite on l'efface
        vTaskDelete(xHandle);
        xHandle = NULL; // On réinitialise la pile
    }
    if (f!=0){ //si la fréquence est différente de 0
        xTaskCreate(TaskLed,"Clignotement", STACK_SIZE,(void *) f, 1, &xHandle);
        // Alors on crée la tâche de clignotement de led qui a pour pvParameters la fréq f
    }
    if (f==0){ // Si la fréquence est nulle
        HAL_GPIO_WritePin(LED_GPIO_Port, LED_Pin, 0); // On configure la sortie du pin à 0
    }
    else return 0;
```

La fonction déclenche l'exécution de la tâche suivante:

```c
void TaskLed(void * pvParameters){
    int freq = (int) pvParameters; // On récupère la fréquence de clignotement voulue
    while(1){
        HAL_GPIO_TogglePin(LED_GPIO_Port, LED_Pin); // On fait clignoter la led
        vTaskDelay(1000/freq);                      // à la fréquence voulue
```

Ainsi on peut maintenant faire varier la période de fonctionnement de la led et pour l'arrêter il suffit de rentrer la fonction “led 0”