ANNEXE : Description et principe de notre OS

On reprendra certains concepts de freeRTOS mais la cible cortex M3 étant connue, la création dynamique de tâches proscrite, on pourra procéder à de nombreuses simplifications :

L'OS maison comportera un nombre prédéfini de tâches défini à la compilation par exemple

#define NB_TASK 4 /* par exemple*/

Il faudra penser à tenir compte de la tâche idle (la tâche par défaut quand aucune autre est executée) lors de la création du tableau des TCB des tâches. La tâche Idle utilisera l'index 0 pour que l'OS soit universel.

les registres sont 32bits, les piles des tâches seront des tableaux de type unsigned int qui ne seront pas définis dynamiquement.

exemple : unsigned int stack_tache[64] ; ou uint32 stack_idle[64] ;

On pourra utiliser un tableau de tableau pour définir toutes les piles de toutes les tâches d'un coup et juste conserver la pile de la tâche idle indépendante :

Peut on indifférement déclarer uint32 stack[NB_TASK][64] ou uint32 stack[64][NB_TASK] ?

On les choisira de 64 entiers 32 bits, pour faciliter l'espionnage de la mémoire allouée à une pile,le but est d'avoir des blocs de 256 octets rendant l'identification de la pile en cours facile.

On sait que les piles sont des piles FULL DESCENDING, on devra donc initialiser le pointeur de pile sur la dernière case de chaque tableau lors de la création des tâches.

Dans quelle fonction cela a t il lieu ? Bien entendu, la fonction sera aussi capable d'initialiser le pointeur de pile avec des piles de toute taille.

On utilisera un tableau de TCB, nommé **OS_TCB**, pour gérer les tâches, la structure du TCB sera simplifiée par rapport à celle de freeRTOS.

Notre TCB sera donc une structure de C qui contiendra :

- un pointeur 32bits sur la position courante de la pile de la tâche : **pxTopOfStack**
- une variable 32 bits sur le niveau de priorité pour la tâche: **ulPriority**
- une variable 32 bits sur l'état de la tâche (-2 à 4) : **ulState**
- une variable 32 bits indiquant une heure de rendez-vous : **ulTickRDV**

On n'utilisera pas de listes chaînées pour dire à quelle liste de tâche on est rattaché mais juste l'élément de structure **ulState** dont la valeur indiquera l'état de la tâche :

valeur de **UlState**	**état de la tâche** codé par cette valeur
-2	tâche considérée détruite (on ne pourra plus jamais lui passer la main)
-1	tâche suspendue
0	tâche en attente de délai
1,2,3,4	priorité de la plus basse (celui de la tâche idle qui est donc plus prioritaire que les tâche en attente) à la plus élevée (on peut ne pas se limiter à 4 niveaux si on souhaite plus de niveaux de priorité)

Dans la fonction appelée par interruption qui détermine vers quelle tâche commuter, on cherchera juste dans le tableau des TCB quel est l'index du TCB qui a la priorité la plus haute, pour mettre à jour les variables de l'OS et renvoyer l'adresse du TCB en question (donc l'adresse de la pile de la tâche en question qui est le premier élément de la structure du TCB).

Dans FREERTOS quel est le nom de l'interruption qui gère la commutation de tâche ?

Pour pouvoir récupérer les codes en assembleur de FREE RTOS, il est indispensable de respecter le nom des variables sensibles : On utilisera la variable classique TCB_t * **pxCurrentTCB**

Comment faire pour que le pointeur puisse être déclaré en tant que TCB_t * ,

et soit compatible avec notre structure de TCB ?

Grâce à **pxCurrentTCB, o**n aura ainsi accès à l'adresse du TCB en cours d'execution :

- cela permettra de "ranger la photo de l'état du processeur" quand on abandonne une tâche (sauvegarde de contexte)
- cela permettra simplement de choisir une nouvelle tâche à executer, en actualisant la variable
- cela permettra de "récupérer la photo de l'état du processeur" quand on reprend une autre tâche (restitution de contexte)

Le code de FREE RTOS de **PendSVHandler** appelle depuis l'assembleur une fonction pour selectionner la nouvelle tâche entre la sauvegarde de contexte et la restitution de contexte:

Si on désire nommer cette fonction autrement, quelle ligne de quel code doit on modifier ?

Si on désire nommer les interruptions autrement, quelle ligne de quel code doit on modifier ?

Technique de recherche de la prochaîne tâche à executer :

On doit parcourir tous les descripteurs de tâche et passer la main à la tâche exécutable (donc ni suspendue, ni en attente) de plus haute priorité :

Quelle est la variable de chaque TCB qu'il faut tester et comparer?

Quelles variables locales faut il pour identifier l'index de la tâche à qui passer la main ?

Si on fait une simple boucle for avec un test "supérieur" ou "supérieur ou égal",

que se passe t il en cas de présence de deux tâches exécutables de même priorité ?

Quel type de boucle différente mettre en place pour éviter de ce soucis ?

Une autre optimisation consiste dans l'IT TickHandler à ne solliciter une commutation de tâche en cascade seulement s'il existe plusieurs tâches de même plus haut niveau.

Par test de type OLDTCB!=NEWTCB, on active ou pas un flag **ulFlag_Yield** pour qu'au prochain Tick_Handler (qui gère les délais et les changements de tâche par la commutation de tâche soit redéclenchée ou pas.

La gestion de l'écoulement du temps pour les tâches endormies se fera dans une interruption SysTick_Handler comme elle se fait en FREE RTOS (port.c ligne 411).

Elle consistera à incrémenter le Tick et à regarder pour toutes les tâches en attente de délai si l'on a atteint l'instant de réveil. Une commutation de tâche systématique sera activée si une tâche de niveau supérieur ou égale à la tâche courante est activée ou si **ulFlag_Yield** est levé (indiquant la présence potentielle de deux tâches de même niveau)

L'OS aura plusieures variables clefs :

On utilisera un index **ulIndexCurrentTCB** pour connaître le TCB courant dans la table des TCB,

cette variable est juste une variable de confort pour aider au debbugage...

Quel est le nom et le type de la véritable variable clef liée à la commutation de tâche?

On utilisera la variable **ulTickCount** pour gérer le temps. **De quel type est elle ?**

On utilisera la variable **ulFlag_Yield** pour se souvenir quand le flag est ON qu'il existe potentiellement deux tâches de même niveau de priorité, entre qui partager le temps CPU.

Les tâches devront toutes avoir le même prototype : void ma_tâche(void * tab_param)

comme dans FreeRTOS, 1 seul paramètre sera passé.