FREE-RTOS

I. Généralités

FreeRTOS est un système d'exploitation temps réel pour les systèmes embarqués. De ce fait il possède une emprunte mémoire faible entre 4 et 9Ko ; il est composé de seulement 5 fichiers C. Le nombre de tâche avec cet OS est illimité tant que le Hardware peut les gérer. Il permet de gérer des tâches périodiques et apériodiques.

La figure 1 présente le diagramme des tâches, i.e. les différents états que peuvent prendre les tâches créées par l'utilisateur et les conditions de passage d'un état à l'autre.

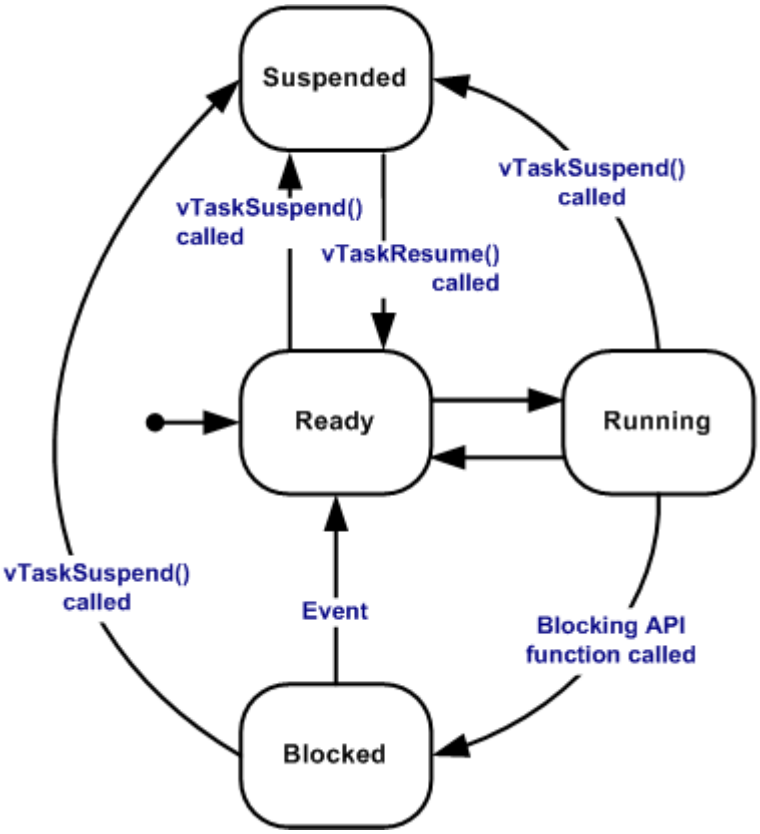


FIGURE 1: DIAGRAMME DES ETATS DES TACHES SOUS FREERTOS

D'après le diagramme de la Figure 1, nous voyons que les tâches peuvent être dans 4 états différents : Suspendu, Prêt, Bloqué ou en Exécution. Il faut rajouter à ces états de base un dernier « état » qui est l'état supprimé.

II. L'ordonnanceur

L'ordonnanceur va définir quelle est la tâche à exécuter parmi celles qui sont dans l'état « Ready » et ceci uniquement sur le niveau de priorité des tâches. De fait plus la tâche dans la file « Ready » a une priorité importante et plus celle-ci a des chances d'être mise en exécution. Le nombre de priorités possibles est défini par la variable tskMAX_PRIORITIES dans le fichier FreeRTOSConfig.h. La valeur de la priorité d'une tâche est statique et seul le développeur peut la modifier en utilisant la fonction vTaskPrioritySet.

Dans le cas ou plusieurs tâches auraient le même niveau de priorité, l'ordonnanceur utilise un algorithme de Round Robin pour faire son choix. Par défaut, l'ordonnancement est préemptif même si deux autres variantes existent (coopératif et hybride). Si jamais une tâche de priorité maximale ne s'arrête jamais alors les autres tâches ne pourront jamais être exécutées car elles ne prendront jamais la main ; ce phénomène est appelé famine. FreeRTOS ne gère pas ce genre de problème donc il faut bien faire attention à l'écriture du code pour éviter ce phénomène par construction.

Comme un microcontrôleur doit toujours avoir quelque chose à exécuter, FreeRTOS créé une tâche IDLE de priorité minimale au démarrage de l'ordonnanceur. Cette tâche IDLE possède des fonctions permettant par exemple de libérer l'espace occupé par des tâches supprimées, mesurer le taux d'utilisation du processeur, mettre en veille le µc lorsqu'aucune tâche n'est en exécution.

III. Les listes

Ces listes sont des outils système interne et sont de type structure de données. Plusieurs structures sont utilisées pour décrire les listes.

- xLIST : représente l'entête des listes créées et utilisées par l'ordonnanceur
    - Nbre d'éléments de la liste
    - Pointeur pour parcourir la liste
    - Element marquant la fin de la liste
- xLIST_ITEM : représente les éléments d'une liste (liste doublement chaînée)
    - Valeur attribuée à cet élément
    - Pointeur vers le prochain élément
    - Pointeur vers l'élément précédent
    - Pointeur vers l'objet contenant cet élément
    - Pointeur vers la liste qui contient l'élément
- xMINI_LIST_ITEM : version réduite de xLISTE_ITEM
    - Elle ne contient que les 3 premiers champs de la liste précédente

La figure 2 présente la façon dont les tâches sont chainées dans FreeRTOS.

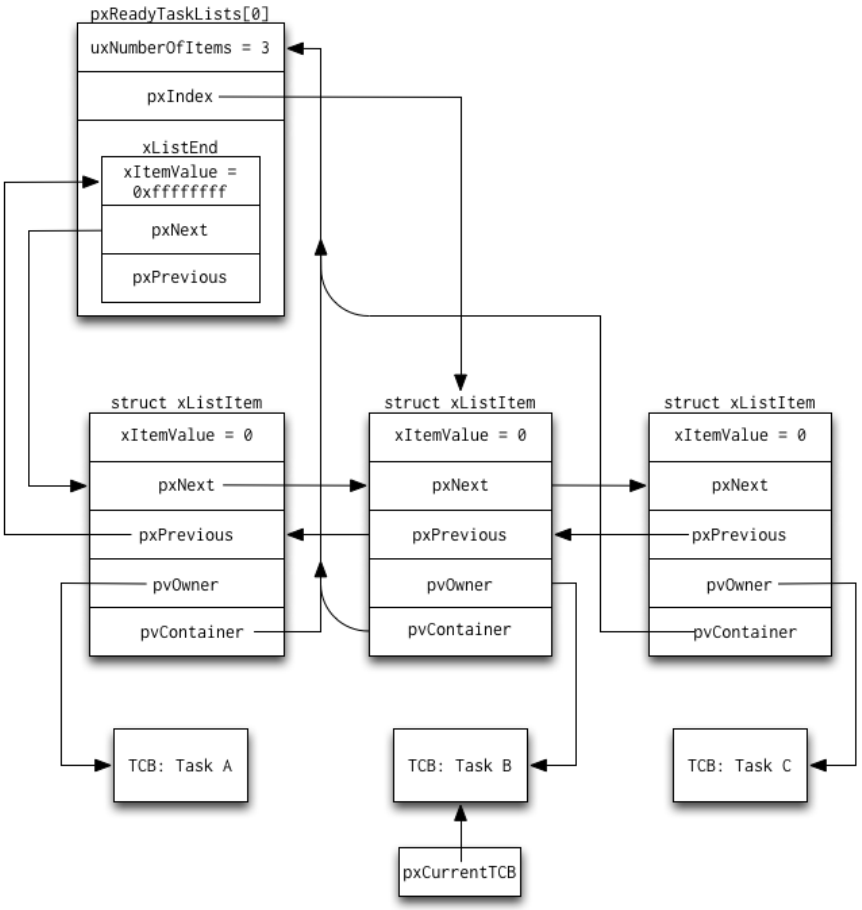


FIGURE 2: CHAINAGE DES TACHES DANS FREERTOS

IV. Les files

Les files sont le principal mécanisme permettant de faire communiquer et de synchroniser les tâches entre elles. Elles sont indispensables afin de construire une application complexe dans laquelle des tâches doivent coopérer. Elles stockent un nombre fini d'élément de taille fixe, elles sont de plus indépendantes des tâches donc peuvent être lues et écrites par différentes tâches. Les lectures/écritures sont réalisées octet par octet.

Typedef struct QueueDefinition
{
*Signed char *pcHead ;* *// pointeur sur l'octet de début de la file en mémoire*
*Signed char *pcTail ;* *// pointeur sur l'octet de fin de la file en mémoire (un octet de plus que nécessaire car utilisé comme un marqueur de fin)*
*Signed char *pcWriteTo ;* *// pointeur sur le prochain octet libre dans la file*
*Signed char *pcReadFrom ;* *// pointeur sur le dernier octet lu de la file*
xList xTasksWaitingToSend ; *// liste des tâches ordonnés par niveau de priorité qui sont bloquées en attente d'écriture sur la file*
xList xTasksWaitingToReceive ; *// liste des tâches ordonnées par niveau de priorités qu sont bloquées en attente de lecture depuis la file*

volatile unsigned portBASE_TYPE uxMessagesWaiting; // nombre d'élément actuellement contenus dans la file
unsigned portBASE_TYPE uxLength ; // taille de la file défini comme le nombre d'éléments max qu'elle peut contenir
unsigned portBASE_TYPE uxItemSize ; // taille de chaque élément (en octet) que la file pourra contenir
} xQUEUE ;

Si plusieurs tâches veulent lire dans une file, c'est la tâche qui a le niveau de priorité la plus élevée qui est choisie. Si plusieurs tâches ont le même niveau de priorité alors c'est celle qui a demandé la lecture en premier qui est servie. Après une lecture, l'élément est supprimé de la file.

*portBASE_TYPE xQueueReceive (xQueueHandle xQueue /*identifiant de la file*/, const void * pvBuffer /*pointeur vers un buffer où la donnée sera copiée*/, portTickType xTicksToWait /*valeur de timeout*/) ;*

Pour l'écriture les règles sont identiques.

Mode FIFO :

*portBASE_TYPE xQueueSend (xQueueHandle xQueue /*identifiant de la file*/, const void * pvItemToQueue /*pointeur vers l'élément à copier*/, portTickType xTicksToWait /*valeur de timeout*/) ;*

*portBASE_TYPE xQueueSendToBack (xQueueHandle xQueue /*identifiant de la file*/, const void * pvItemToQueue /*pointeur vers l'élément à copier*/, portTickType xTicksToWait /*valeur de timeout*/) ;*

Mode LIFO :

*portBASE_TYPE xQueueSendToFront (xQueueHandle xQueue /*identifiant de la file*/, const void * pvItemToQueue /*pointeur vers l'élément à copier*/, portTickType xTicksToWait /*valeur de timeout*/) ;*

Création d'une file :

*xQueueHandle xQueueCreate(unsigned portBASE_TYPE uxQueueLength /*nbre d'éléments*/, unsigned portBASE_TYPE uxItemSize /*Taille d'un élément*/) ;*

A sa création une file est vide.

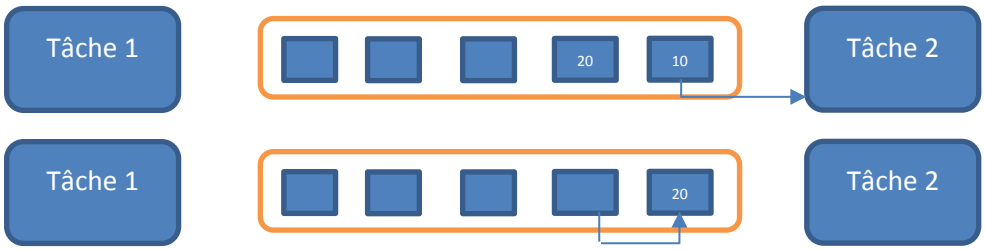
Exemple : création d'une file de 5 éléments entre deux tâches 1 et 2.

La tâche 1 va écrire un élément dans la file

La tâche 1 a écrit une valeur dans la queue et comme la file était préalablement vide alors la première valeur est aussi la dernière

La tâche 1 écrit une seconde valeur dans la file. La nouvelle donnée est écrite à la suite de la présente.

La tâche 2 vient lire un élément de la file ; dès que la lecture est effectuée l'élément lu est supprimé de la file



L'élément restant prend alors la place de l'élément supprimé suite à la lecture.

V. Sémaphores et Mutex

FreeRTOS synchronise principalement les tâches en utilisant un de ces deux mécanismes.

V.I Les sémaphores :

Il existe deux types de sémaphore les sémaphores binaires ou les sémaphores compteurs.

- La manière la plus simple et efficace pour synchroniser des tâches est d'utiliser des sémaphores binaires. En fait ces sémaphores peuvent être vus comme une file à 1 élément.

Création d'un sémaphore :

Void vSemaphoreCreateBinary(xSemaphoreHandle xSemaphore) ;

Prendre un sémaphore :

portBASE_TYPE xSemaphoreTake(xSemaphoreHandle xSemaphore, portTickType xTicksToWait) ;

Libérer un sémaphore :

portBASE_TYPE xSemaphoreGive(xSemaphoreHandle xSemaphore) ;

V.II Les Mutex :

Les Mutex permettent d'éviter les exclusions mutuelles (évite que des ressources partagées soient utilisées en même temps); la libération d'un Mutex ne peut être fait que par la tâche qui l'a pris. Les Mutex héritent de la priorité des tâches qui les créent afin d'éviter des inversions de priorité.

Création d'un Mutex :

xSemaphoreHandle xSemaphoreCreateMutex(void) ;

VI. Gestion des interruptions

FreeRTOS n'impose pas de stratégie spécifique pour la gestion des interruptions. C'est donc à vous de construire la stratégie à l'aide de 2 constantes du fichier FreeRTOSConfig.h :

- configKERNEL_INTERRUPT_PRIORITY
- configMAX_SYSCALL_INTERRUPT_PRIORITY

Les sous routines d'interruption (ISR) sont exécutées par le processeur et donc non vu par l'OS ce qui peut engendrer des comportements inattendus du noyau (**très important pour le temps réel**). Il faut donc réduire au maximum le temps d'exécution d'une ISR. On peut donc utiliser les sémaphores binaires afin de transférer le processus d'interruption vers une simple tâche.

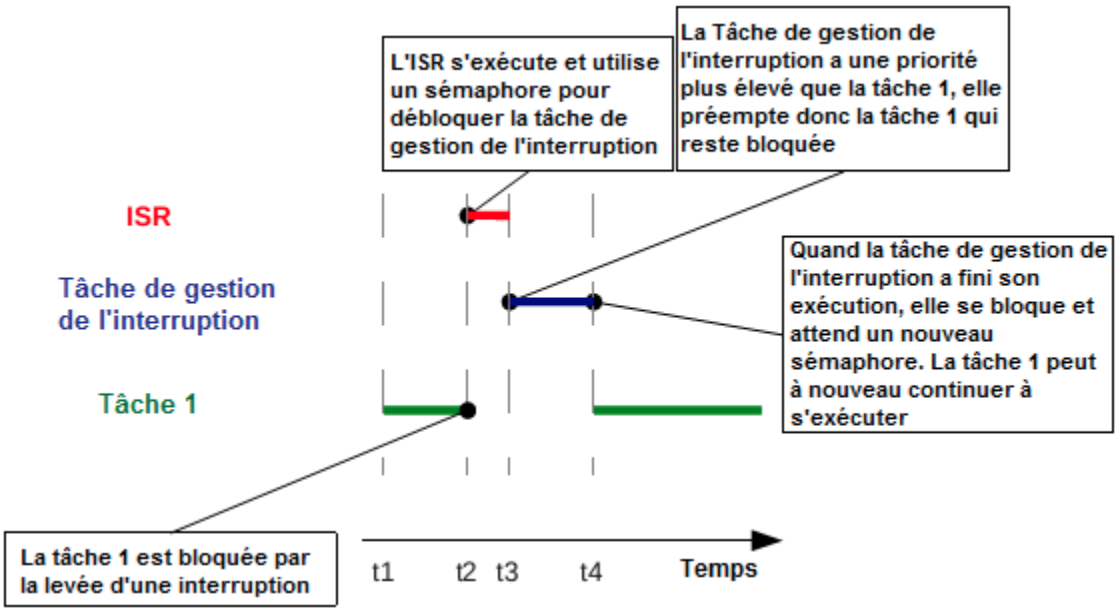


FIGURE 3: GESTION DES INTERRUPTIONS

VII. Les sections critiques

Les sections critiques sont des portions de code qui ont besoin d'être protégés de tout changement de contexte pour éviter d'éventuelles corruptions. Tout le code placé entre taskENTER_CRITICAL() et taskEXIT_CRITICAL() sera donc protégé ce qui signifie que les interruptions et l'ordonnanceur seront désactivés.

L'utilisation de ces portions doit se faire de façon parcimonieuse car durant leur exécution, le système reste « figé » donc aucune autre tâche ne peut s'exécuter et aucune interruption ne peut être vue (sauf les interruptions non masquables).

Une autre méthode moins restrictive peut être utilisée. En effet, il est possible de stopper l'ordonnanceur tout en conservant le mécanisme d'interruption actif.

vTaskSuspendAll() ;

{

// code en section critique

}

xTaskResumeAll() ;

VIII. Gestion de la mémoire

Le noyau de FreeRTOS doit pouvoir allouer dynamiquement de la mémoire à chaque fois qu'une tâche, une file ou un sémaphore est créé. Pour cela deux méthodes existent pvPortMalloc() et le vPortFree(). La quantité de mémoire disponible est définie par la constante configTOTAL_HEAP_SIZE dans le fichier FreeRTOSConfig.h.

Dans de tout petit système embarqué, il est possible d'allouer tous les objets/tâches et de jamais avoir besoin de réallouer ou de dés-allouer la mémoire dans ce cas la méthode pvPortFree ne sert à rien (extrêmement rare).

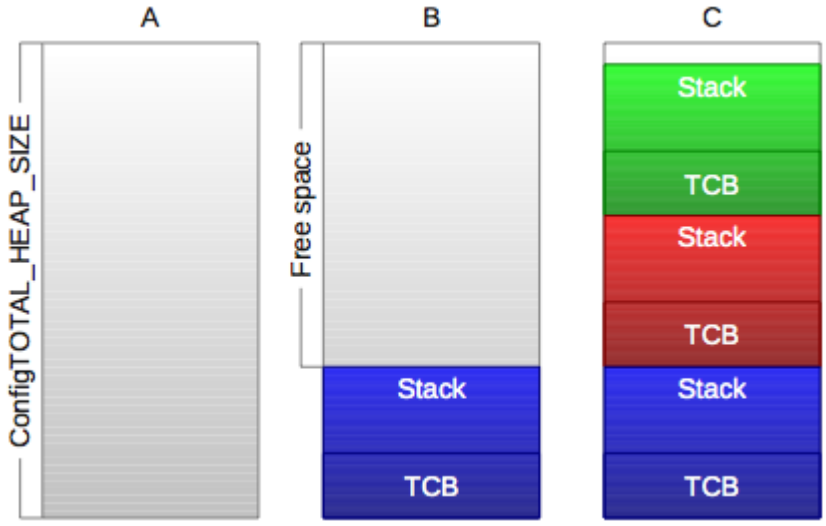


FIGURE 4: GESTION DE LA MEMOIRE DANS DE TOUT PETIT SYSTEME EMBARQUE

Attention une très grande rigueur de programmation est exigée afin de bien gérer la mémoire ; nous sommes sur des systèmes contraints.