Chapitre 9

FreeRTOS - RTOS Fonctionnement

9.1 Introduction

Un système d'exploitation temps réel, en anglais RTOS pour real-time operating system, est un système d'exploitation pour lequel le temps maximum entre un stimulus d'entrée et une réponse de sortie est précisément déterminé.

Ces systèmes d'exploitation multitâches sont destinés à des applications temps réel : systèmes embarqués (thermostats programmables, contrôleurs électroménagers, téléphones mobiles, robots industriels, vaisseaux spatiaux, systèmes de contrôle commande industriel, matériel de recherche scientifique).

Un RTOS facilite la création d'un système temps réel, mais ne garantit pas que le résultat final respecte les contraintes temps réel, ce qui exige le développement correct du logiciel. Un RTOS n'a pas nécessairement pour but d'être performant et rapide, mais un RTOS fournit des services et des primitives qui, si elles sont utilisées correctement, peuvent garantir les délais souhaités. Un RTOS utilise des ordonnanceurs spécialisés afin de fournir aux développeurs des systèmes temps réel les outils et les primitives nécessaires pour produire un comportement temps réel souhaité dans le système final.

9.2 FreeRTOS

FreeRTOS est un système d'exploitation temps réel (RTOS) faible empreinte, portable, préemptif et Open source pour microcontrôleur. Il a été porté sur plus de 40 architectures différentes. Le noyau FreeRTOS a été développé à l'origine par Richard Barry vers 2003, puis développé et maintenu par la société de Barry, Real Time Engineers Ltd. En 2017, Real Time Engineers Ltd. a transmis la propriété du projet FreeRTOS à Amazon Web Services. Barry continue à travailler sur FreeRTOS au sein d'une équipe AWS. Il est aujourd'hui parmi les plus utilisés dans le marché des systèmes d'exploitation temps réel.

FreeRTOS est disponible gratuitement sous une licence MIT depuis le 29 novembre 2017. Les versions précédentes étaient disponibles sous licence GPL modifiée et utilisable sans paiement de redevances, cette licence n'oblige pas les développeurs à publier le code de leurs logiciels applicatifs mais impose de garder le noyau de FreeRTOS Open Source.

Une licence commerciale avec le support ad hoc (OpenRTOS) est également proposée par la société High Integrity Systems.

Le nombre de tâches exécutées simultanément et leur priorité ne sont limités que par le matériel. L'ordonnancement est un système de file d'attente basé sur les Sémaphores et les Mutex. Il est basé sur le modèle Round-Robin avec gestion des priorités. Conçu pour être très compact, il n'est composé que de quelques fichiers en langage C, et n'implémente aucun pilote matériel.

Les domaines d'applications sont assez larges, car les principaux avantages de FreeRTOS sont l'exécution temps réel, un code source ouvert et une taille très faible. Il est donc utilisé principalement pour les systèmes embarqués qui ont des contraintes d'espace pour le code, mais aussi pour des systèmes de traitement vidéo et des applications réseau qui ont des contraintes temps réel.

Sa conception est volontairement minimaliste pour qu'il puisse être installé sur de petits systèmes. La taille de l'image binaire du noyau varie 6 KB et 12 KB (4,3 KB compilé sur ARM7). Le kit minimal ne compte que quelques fonctions pour gérer les tâches et la mémoire. De plus, les files d'attente, les sémaphores et les mutex sont implémentés.

Le noyau de la version 7.3.0 n'est composé que de cinq fichiers codés en langage C. Des sections en Assembleur permettent d'assurer la compatibilité de FreeRTOS avec les différentes architectures matérielles.

Un nombre illimité de tâches peut être exécuté simultanément et sans contrainte

9.3 Caractéristiques

- Fournit une solution unique et indépendante pour de nombreuses architectures et outils de développement différents,
- Connu pa sa fiabilité. La fiabilité est assurée par les activités entreprises par le projet associé SafeRTOS.
- Riche en fonctionnalités et fait toujours l'objet d'un développement actif et continu.
- Un minimum de ROM, de RAM et de coûts de traitement. Typiquement, une image binaire de noyau RTOS est de l'ordre de 6 K à 12 K octets.
- Très simple, le cœur du noyau du RTOS est contenu dans seulement 3 fichiers C. La majorité des nombreux fichiers inclus dans le téléchargement du fichier .zip ne concernent que les nombreuses applications de démonstration.

— Gratuit pour une utilisation dans des applications commerciales (voir les conditions de licence pour plus de détails).
— Une licence commerciale, un support professionnel et des services de portabilité sont disponibles sous la forme d'OPENRTOS du partenaire WITTENSTEIN high integrity systems.
— Possède une option de conversion vers SafeRTOS, qui comprend des certifications pour les secteurs médical, automobile et industriel.
— Bonne implantation avec une base d'utilisateurs importante et en constante augmentation.
— Comporte un exemple pré-configuré pour chaque port. Pas besoin de chercher comment configurer un projet - il suffit de télécharger et de compiler.
— Très évolutif, simple et facile à utiliser.
— FreeRTOS offre une alternative de traitement en temps réel plus petite et plus facile pour les applications où eCOS, Linux embarqué (ou Real Time Linux) et même uCLinux ne conviennent pas, ne sont pas appropriés, ou ne sont pas disponibles.

9.4 Architectures matérielles supportées

Les architectures matérielles supportées
— Altera : Nios II
— ARM architecture : ARM7, ARM9, ARM Cortex-M0, ARM Cortex-M3, ARM Cortex-M4, ARM Cortex-M7
— Atmel : Atmel AVR, AVR32, SAM3, SAM7, SAM9
— Cortus : APS3
— Espressif Systems : ESP32, ESP8266
— Fujitsu : MB91460 series, MB96340
— Freescale : Coldfire V1, Coldfire V2, HCS12, Kinetis
— Intel : x86, 8052
— PIC : PIC18, PIC24, dsPIC, PIC32
— Renesas : 78K0R, RL78, H8/S, RX600, RX200, SuperH, V850
— STMicroelectronics : STM32, STR7
— Texas Instruments : MSP430, Stellaris, Tiva C
— Xilinx : MicroBlaze
— IBM : PPC405, PPC404
— NXP : LPC2000, LPC1000, LPC4300
— Infineon : TriCore, XMC4000
— Microsemi : SmartFusion

— Cypress : PSoC
— Energy Micro : EFM32

9.5 Architecture

FreeRTOS a été conçu pour être très léger, de 6 ko à 12 ko pour une image binaire classique du noyau RTOS. Le noyau lui-même n'est composé que de trois fichiers source écrit en langage C.

9.5.1 L'ordonnanceur

L'ordonnancement des tâches a pour but principal de décider parmi les tâches qui sont dans l'état `prêt`, laquelle exécuter. Pour faire ce choix, l'ordonnanceur de FreeRTOS se base uniquement sur la priorité des tâches.

Les tâches en FreeRTOS se voient assigner à leur création, un niveau de priorité représenté par un nombre entier. Le niveau le plus bas vaut zéro et il doit être strictement réservé pour la tâche Idle. L'utilisateur a la possibilité de surcharger ce niveau avec une priorité personnalisée en modifiant la constante : `tskIDLE_PRIORITY`. Le nombre maximum de niveaux de priorités est défini par la constante : `tskMAX_PRIORITIES`. Plusieurs tâches peuvent appartenir à un même niveau de priorité.

Dans FreeRTOS Il n'y a aucun mécanisme automatique de gestion des priorités. La priorité d'une tâche ne pourra être changée qu'à la demande explicite du développeur.

Les tâches sont de simples fonctions qui généralement s'exécutent en boucle infinie et qui suivent la structure générique suivante :

```
void vATaskFunction( void *pvParameters )
{
    for( ;; )
    {
        -- Ajouter le code de votre tâche ici --
    }
}
```

Dans le cas d'un micro-contrôleur possédant un seul cœur, il y aura à tout moment une seule tâche en exécution. L'ordonnanceur garantira toujours que la tâche de plus haute priorité pouvant s'exécuter sera sélectionnée pour entrer dans l'état d'exécution. Si deux tâches partagent le même niveau de priorité et sont toutes les deux capables de s'exécuter, alors les deux tâches s'exécuteront en alternance par rapport aux réveils de l'ordonnanceur (round robin).

Afin de choisir la tâche à exécuter, l'ordonnanceur doit lui-même s'exécuter et préempter la tâche en état d'exécution. Afin d'assurer le réveil de l'ordonnanceur, FreeRTOS définit une interruption périodique nommé la **tick interrupt**. Cette interruption s'exécute infiniment selon une certaine fréquence qui est définie dans le fichier `FreeRTOSConfig.h` par la constante :

```
configTICK_RATE_HZ  /* Fréquence d'exécution de la "tick interrupt" en Hz.*/
```

Cette constante décrit alors la période de temps allouée au minimum pour chaque tâche ou expliqué autrement, l'intervalle séparant deux réveils de l'ordonnanceur.

FreeRTOS est donc un RTOS qui utilise un ordonnancement préemptif pour gérer les tâches. Néanmoins il peut aussi utiliser optionnellement (si la directive lui est donnée) l'ordonnancement coopératif. Dans ce mode d'ordonnancement, un changement de contexte d'exécution a lieu uniquement si la tâche en exécution permet explicitement à une autre tâche de s'exécuter (en appelant un Yield() par exemple) ou alors en entrant dans un état de blocage. Les tâches ne sont donc jamais préemptées. Ce mode d'ordonnancement simplifie beaucoup la gestion des tâches, malheureusement il peut mener à un système moins efficace et moins sûr.

FreeRTOS peut aussi utiliser un ordonnancement hybride, utilisant l'ordonnancement préemptif et l'ordonnancement coopératif. Dans ce mode, un changement de contexte d'exécution peut aussi avoir lieu lors de l'événement d'une interruption

La famine

Dans FreeRTOS, la tâche de plus haute priorité prête à s'exécuter sera toujours sélectionnée par l'ordonnanceur. Ceci peut amener à une situation de *famine*. En effet, si la tâche de plus haute priorité n'est jamais interrompue, toutes les tâches ayant une priorité inférieure ne s'exécuteront jamais. FreeRTOS n'implémente aucun mécanisme automatique pour prévenir le phénomène de *famine*. Le développeur doit s'assurer lui-même qu'il n'y ait pas de tâches monopolisant tout le temps d'exécution du microcontrôleur. Pour cela, il peut placer des évènements qui interrompront la tâche de plus haute priorité pour un temps déterminé ou jusqu'à l'avènement d'un autre événement et laissant ainsi le champ libre aux tâches de priorités inférieures pour s'exécuter.

Afin d'éviter la famine, le développeur peut utiliser l'ordonnancement à taux monotones (RMS). C'est une technique d'assignation de priorité qui attribue à chaque tâche une unique priorité selon sa fréquence d'exécution. La plus grande priorité est attribuée à la tâche de plus grande fréquence d'exécution et la plus petite priorité est attribuée à la tâche de plus petite fréquence. L'ordonnancement à taux monotones permet de maximiser l'ordonnançabilité des tâches mais cela reste difficile à atteindre du fait de la nature des tâches qui ne sont pas totalement périodiques.

La tâche Idle

Un microcontrôleur doit toujours avoir quelque chose à exécuter. En d'autres termes, il doit toujours y avoir une tâche en exécution. FreeRTOS gère cette situation en définissant la tâche `Idle` qui est créée au lancement de l'ordonnanceur. La plus petite priorité du noyau est attribué à cette tâche. Malgré cela, la tâche `Idle` peut avoir plusieurs fonctions à remplir dont :

- Libérer l'espace occupé par une tâche supprimée.
- Placer le microcontrôleur en veille afin d'économiser l'énergie du système lorsqu'aucune tâche applicative n'est en exécution.
- Mesurer le taux d'utilisation du processeur.

9.5.2 La description des tâches

Dans FreeRTOS, chaque tâche est décrite par un TCB (Task control Block) qui contient toutes les informations nécessaires afin de spécifier et de représenter une tâche.

Variables	Description
`pxTopOfStack`	Pointeur vers le dernier élément placé en haut de la pile.
`xGenericListItem`	Elément (`xListItem`) de la liste utilisé pour placer la TCB dans une liste d'ét
`xEventListItem`	Elément (`xListItem`) de la liste utilisé pour placer la TCB dans une liste d'év
`uxPriority`	Priorité de la tâche
`pxStack`	Pointeur vers le début de la pile du processus
`pxEndOfStack`	Pointeur vers la fin de la pile du processus
`uxTCBNumber`	Taille de la pile en nombre de variables
`pcTaskName`	Nombre s'incrémentant à chaque fois qu'une TCB est créée (utilisé pour le dé
`uxBasePriority`	La dernière priorité assignée à la tâche
`ulRunTimeCounter`	Calcule le temps passé par la tâche dans un état d'exécution
`pxTaskTag`	Permet d'ajouter un Tag à une tâche. Le Tag s'utilise pour réaliser des Log vi
`uxCriticalNesting`	Permet de sauvegarder la profondeur d'imbrication des sections critiques de ce

Table 9.1 – Task Control Block

Les tâches sous FreeRTOS peuvent exister sous 5 états : `supprimé`, `suspendu`, `prêt`, `bloqué` ou `en exécution`.

Dans FreeRTOS, il n'y a aucune variable pour spécifier explicitement l'état d'une tâche, en contrepartie FreeRTOS utilise des listes d'états. La présence de la tâche dans un type de listes d'états détermine son état (prêt, bloqué ou suspendu). Les tâches changent

souvent d'état, l'ordonnanceur n'aura alors qu'à déplacer la tâche (l'élément (`xListItem`) appartenant à cette même tâche) d'une liste d'états à une autre.

A la création d'une tâche, FreeRTOS crée et remplit la TCB correspondant à la tâche, puis il insère directement la tâche dans une **Ready List** (Liste contenant une référence vers toutes les tâches étant dans l'état `Prêt`).

FreeRTOS maintient plusieurs `Ready List`, une liste existe pour chaque niveau de priorité. Lors du choix de la prochaine tâche à exécuter, l'ordonnanceur analyse les "Ready list" de la plus haute priorité à la plus basse.

Plutôt que de définir explicitement un état `en exécution` ou une liste associée à cet état, le noyau FreeRTOS décrit une variable `pxCurrentTCB` qui identifie le processus en exécution. Cette variable pointe vers la TCB correspondant au processus se trouvant dans l'une des `Ready list`.

Une tâche peut se retrouver dans l'état `bloqué` lors de l'accès à une file en lecture/écriture dans le cas où la file est vide/pleine. Chaque opération d'accès à une file est paramètrée avec un timeout (`xTicksToWait`), Si ce timeout vaut 0 alors la tâche ne se bloque pas et l'opération d'accès à la file est considérée comme échouée. Dans le cas où le timeout n'est pas nul, la tâche se met dans l'état `bloqué` jusqu'à ce qu'il y ait une modification de la file (par une autre tâche par exemple). Une fois l'opération d'accès à la file possible, la tâche vérifie que son timeout n'est pas expiré et termine avec succès son opération.

Une tâche peut être volontairement placée dans l'état `suspendu`, elle sera alors totalement ignorée par l'ordonnanceur et ne consommera plus aucune ressource jusqu'à ce qu'elle soit retirée de l'état et remise dans un état `prêt`.

Le dernier état que peut prendre une tâche est l'état `supprimé`, cet état est nécessaire car une tâche supprimée ne libère pas ses ressources instantanément. Une fois dans l'état `supprimé`, la tâche est ignorée par l'ordonnanceur et une autre tâche nommée `IDLE` est chargée de libérer les ressources allouées par les tâches étant en état `supprimé`.

La tâche `IDLE` est créée lors du démarrage de l'ordonnanceur et se voit assigner la plus petite priorité possible ce qui conduit à une libération retardée des ressources lorsque aucune autre tâche est en exécution

9.5.3 Structure des données

Les Listes

Les listes sont les structures de données les plus utilisées dans FreeRTOS. Elles sont utilisées pour organiser et ordonnancer les tâches, ainsi que pour l'implémentation des files.

FreeRTOS définit plusieurs structures afin de représenter les listes:

— La structure `xLIST` représente l'en-tête des listes qui sont créées et utilisées par l'ordonnanceur. Par exemple, les listes des tâches prêtes (une liste par niveau de priorité), la liste des tâches bloquées, etc. Cette structure contient d'une certaine façon les autres structures de listes utilisées par FreeRTOS.
La structure `xLIST` est définie comme suit :

```
typedef struct xLIST
{
  volatile unsigned portBASE_TYPE uxNumberOfItems; /* Le nombre d'éléments dans
                                                      cette liste. */

  volatile xListItem * pxIndex;     /* Pointeur utilisé pour parcourir cette
                                       liste,il pointe successivement sur les
                                       éléments (xLIST_ITEM) contenus dans
                                       cette liste. */

  volatile xMiniListItem xListEnd; /* Elément marquant la fin de cette liste.
                                      Elle contient pour cela la valeur maximale
                                      de la variable xItemValue dans la liste. *
} xList;
```

— La structure `xLIST_ITEM` représente les éléments d'une liste (de Type `xLIST`). Chaque élément d'une liste est liée à une tâche et les différentes variables de la structure `xLIST_ITEM` servent à organiser les tâches et à les lier entre elles afin de former une liste doublement chaînée.
La structure `xLIST_ITEM` est définie comme suit :

```
struct xLIST_ITEM
{
  portTickType xItemValue;  /* Une valeur attribuée à cet élément,
                               cette valeur est utilisée afin de trier
                               la liste (de type (xLIST) contenant cet
                               élément (xLIST_ITEM))dans un ordre
                               décroissant. */

  volatile struct xLIST_ITEM * pxNext;  /* Pointeur vers le prochain
                                           élément (xLIST_ITEM) dans
                                           la liste (xLIST). */
```

```
    volatile struct xLIST_ITEM * pxPrevious; /* Pointeur vers le précédent
                                                élément (xLIST_ITEM) dans
                                                la liste (xLIST). */

    void * pvOwner;                          /* Pointeur vers l'objet
                                                contenant cet élément,
                                                cet objet est, dans la
                                                plupart des cas, le TCB
                                                d'une tâche. */

    void * pvContainer;                      /* Pointeur vers la liste
                                                (xLIST) dans laquelle cet
                                                élément est contenu. */

};
```

— La structure `XMiniListItem` est une version réduite de `XLIST_ITEM`. Elle ne dispose pas des variables `pvOwner` et `pvContainer`. Elle représente un élément marquant la fin de cette liste.

Les files

Les files sont les mécanismes principaux qui permettent de faire communiquer et synchroniser les tâches entre elles.

La structure basique d'une file est décrite comme suit :

```
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 (ordonnées par niveau
                                  de priorités) qui sont bloquées en
                                  attente d'écriture sur la file. */

    xList xTasksWaitingToReceive;  /* Liste des tâches (ordonnées par
                                      niveau de priorités) qui sont
                                      bloquées en attente de lecture
                                      depuis la file .*/

    volatile unsigned portBASE_TYPE uxMessagesWaiting; /* Nombre d'éléments
                                                          actuellement contenus
                                                          dans la file. */

    unsigned portBASE_TYPE uxLength;  /* Taille de la file définit comme le
                                         nombre d'éléments maximum qu'elle
                                         peut contenir et non le nombre d'octets. */

    unsigned portBASE_TYPE uxItemSize;   /* Taille de chaque élément (en octet)
                                            que la file pourra contenir. */

} xQUEUE;
```

Une file contient un nombre fini d'éléments de taille fixe. La taille physique d'une file est déterminée par le nombre maximum d'éléments qu'elle peut contenir (`uxLength`) multiplié par la taille en octet de chaque élément (`uxItemSize`).

Dans FreeRTOS, l'écriture ou l'envoi de données dans une file est effectuée en copiant octet par octet et quel que soit son type car la durée de vie de l'élément sauvegardé est bien souvent inférieure à la durée de vie de la file. A l'instar de l'opération d'écriture, la lecture ou réception de données est également effectuée en copiant octet par octet la donnée qui sera supprimée de la file.

Les files sont des objets indépendants, il n'y a aucune assignation ou appartenance à une tâche. Elles sont les structures qui permettent aux tâches de communiquer entre elles. De ce fait, elles peuvent avoir plusieurs tâches en lecture et en écriture simultanément.

Les opérations d'écriture ou de lecture sur les files peuvent être bloquantes ou non-bloquantes. Les opérations non-bloquantes retournent directement le statut réussi ou échoué sur la file. Les opérations bloquantes sont paramétrées avec un timeout qui leur

permet de se bloquer et qui détermine le temps maximum pendant lequel elles peuvent rester dans cet état.

9.5.4 La gestion des ressources

FreeRTOS utilise les files comme moyen de communication entre les tâches. Cependant, les files gèrent aussi la synchronisation et la concurrence entre les tâches17. De ce fait, cette structure est l'élément de base pour la gestion des ressources dans FreeRTOS.

FreeRTOS synchronise les tâches en utilisant principalement deux mécanismes : les Sémaphores et les Mutex.

Sémaphores

FreeRTOS permet la création et l'utilisation de Sémaphores à plusieurs éléments. Les sémaphores sont implémentés sous forme de file tel que le nombre d'éléments du sémaphore représente le nombre d'éléments maximum que la file peut contenir. La file représentant le sémaphore ne sauvegarde aucune donnée, elle s'occupe uniquement d'enregistrer le nombre de ses entrées actuellement occupées. La taille des éléments de la file est donc nulle (`uxItemSize=0`). Un accès au Sémaphore ne fait qu'augmenter ou diminuer le nombre d'entrées occupées dans la file et aucune copie d'élément n'est effectuée.

L'API offerte par FreeRTOS pour la gestion des sémaphores fait une différence entre les sémaphores à N éléments et les sémaphore binaires. Les Sémaphores binaires peuvent être vus comme des files ne pouvant contenir qu'un seul élément. Un Sémaphore binaire ne pourra donc être pris qu'une seule fois avant qu'il ne devienne indisponible contrairement au Sémaphore à n éléments qui pourra être pris à plusieurs reprises ce qui permet par exemple, de définir un nombre de ressources disponibles ou alors de compter le nombre d'événements qui doit être encore exécuté.

Mutex

Un Mutex est utilisé afin de protéger une ressource partagée.

L'implémentation des Mutex dans FreeRTOS est similaire à celle des Sémaphores binaires (sous la forme d'une file) sauf que la tâche qui prend le Mutex doit obligatoirement le rendre. Cela peut être vu comme l'association d'un jeton à une ressource, une tâche prend le jeton et utilise la ressource puis rend le jeton à la fin, au même moment aucun autre jeton supplémentaire ne pourra être associé à la tâche.

Une autre différence majeure entre les Mutex et les Sémaphores binaires dans FreeRTOS est le système d'héritage de priorité. Quand plusieurs tâches demandent à prendre un Mutex, la priorité du détenteur du Mutex est fixée momentanément à la valeur de la

plus haute priorité parmi les tâches qui attendent sa libération. Cette technique a pour effet de prévenir les phénomènes à risques d'inversion de priorité même si cela ne garantit pas une sécurité infaillible face à ces phénomènes.

9.5.5 La gestion des interruptions

Une interruption est un mécanisme purement matériel qui est implémenté et lancé par ce dernier. FreeRTOS ne fait que fournir des méthodes servant à la gestion des interruptions et il peut également lancer des interruptions par appel à une instruction matérielle.

FreeRTOS n'impose aux développeurs aucune stratégie spécifique pour la gestion des interruptions mais il offre plusieurs moyens pour que la stratégie choisie puisse être implémentée et maintenue facilement.

Les routines d'interruptions (ISR) sont des fonctions exécutées par le microcontrôleur lui-même et qui ne peuvent être gérées par FreeRTOS, ce qui peut poser certains problèmes. Pour ces raisons, les routines d'interruptions ne peuvent pas utiliser les fonctions habituelles de l'API FreeRTOS que toute autre tâche basique peut utiliser. Néanmoins FreeRTOS définit un groupe de fonctions spécialement conçues pour les ISR, par exemple, une ISR utilisera la fonction `xSemaphoreGiveFromISR()` plutôt que `xSemaphoreGive()`, de la même manière, elle utilisera la fonction `xQueueReceiveFromISR()` plutôt que `xQueueReceive()`.

Afin de spécifier une politique de gestion des interruptions et de gérer l'accès aux fonctions du noyau spécifiques aux ISR, FreeRTOS définit des constantes dans le fichier de configuration `FreeRTOSConfig.h`:

- `configKERNEL_INTERRUPT_PRIORITY` : Définit le niveau de priorité de l'interruption temporelle (ang. Tick interrupt) qui est une interruption périodique utilisée pour lancer l'ordonnanceur à chaque intervalle de temps.
- `configMAX_SYSCALL_INTERRUPT_PRIORITY` : Définit le plus haut niveau de priorité pour lequel les fonctions spécifiques aux ISR de FreeRTOS peuvent être utilisées.

La définition de ces deux constantes permet de spécifier une politique de gestion des ISR selon leur niveau de priorités :

- une ISR ayant un niveau de priorité compris entre `configKERNEL_INTERRUPT_PRIORITY` et `configMAX_SYSCALL_INTERRUPT_PRIORITY` pourra utiliser les fonctions de l'API spécifique aux ISR, pourra préempter une tâche, mais pas le noyau ni une tâche exécutant une section critique. Elle ne pourra pas être préemptée par l'ordonnanceur car l'interruption du tick timer a une priorité plus basse.
- une ISR ayant un niveau de priorité strictement supérieur à `configMAX_SYSCALL_INTERRUPT_PRI` pourra préempter l'ordonnanceur, même lors de l'exécution de sections de code critique. En contrepartie elle n'aura plus accès à aucune des fonctions de l'API de FreeRTOS.

— une ISR n'utilisant aucune des fonctions de l'API de FreeRTOS peut utiliser n'importe quel niveau de priorité.

La définition des deux constantes précédentes ajoute aussi aux ISR la particularité d'imbrication d'interruptions (ang. Interrupt Nesting). L'imbrication d'interruptions est la possibilité qu'une deuxième interruption ait lieu au même moment que le traitement d'une autre interruption par une ISR. Cette deuxième interruption peut préempter la première si elle dispose d'une priorité plus élevée.

Il est à noter que les priorités des interruptions sont définies par l'architecture du microcontrôleur. Ce sont des priorités matérielles qui n'ont aucune relation avec les priorités logicielles que l'on peut attribuer aux tâches grâce à FreeRTOS.

Les interruptions différées

Comme spécifié précédemment, les ISR sont des sections de code exécutées par le micro-contrôleur et non par FreeRTOS. Ce qui amène à des comportements inattendus du noyau. Pour cette raison, il est nécessaire de réduire au maximum le temps d'exécution d'une ISR. Une stratégie pour réduire le temps d'exécution est d'utiliser les Sémaphores binaires offerts par FreeRTOS. Un Sémaphore binaire peut être utilisé afin de débloquer une tâche à chaque fois qu'une interruption particulière a lieu. Ainsi la partie de code exécutée dans l'ISR pourra être très largement réduite et la gestion de l'interruption reviendra en grande partie à la tâche débloquée. On aura ainsi différé le processus d'interruption vers une simple tâche.

Si l'interruption s'avère être critique, alors la priorité de la tâche de gestion de l'interruption pourra être définie de manière à toujours préempter les autres tâches du système.

Suspension des interruptions

FreeRTOS permet de protéger des sections de code appartenant à une tâche de tout changement de contexte, d'opération de l'ordonnanceur ou même d'une levée d'interruption, ces portions de code sont appelées sections critiques. L'utilisation de sections critiques peut être efficace pour respecter l'atomicité de certaines instructions. Néanmoins ces sections critiques sont à utiliser avec précaution car pendant leur exécution, le système reste statique et totalement inactif envers d'autres tâches critiques qui seraient bloquées ou alors envers des interruptions signalant des événements extérieurs critiques.

Afin de définir une partie de code comme étant une section critique, il suffit de l'englober entre les deux instructions de début et de fin de section critique : `taskENTER_CRITICAL()` et `taskEXIT_CRITICAL()`.

9.5.6 La gestion de la mémoire

Le noyau FreeRTOS doit allouer dynamiquement la mémoire RAM à chaque fois qu'une tâche, une file ou un Sémaphore est créé. L'utilisation des traditionnelles méthodes `Malloc()` et `Free()` est toujours possible mais elle peut poser quelques problèmes car elles ne sont pas déterministes et elles peuvent souffrir d'une mémoire trop fragmentée. De ce fait, deux méthodes reprenant les mêmes prototypes existent dans FreeRTOS : `pvPortMalloc()` et `vPortFree()`.

Chacune d'elles est implémentée de trois manières différentes décrites dans les fichiers `Heap_1.c`, `Heap_2.c` et `Heap_3.c` mais l'utilisateur est libre de définir ses propres fonctions.

La constante `configTOTAL_HEAP_SIZE` définie dans le fichier de configuration `FreeRTOSConfig.h`

La première implémentation

Cette version ne définit pas de méthode pour libérer de l'espace mémoire RAM.

La mémoire est divisée en un tableau de taille `configTOTAL_HEAP_SIZE` (en octets) nommé la *pile FreeRtos*.

Lorsque le noyau a besoin d'allouer de la mémoire, il réserve deux espaces libres pour une même tâche. Le premier est utilisé pour la TCB alors que le second représente la file. Dans la mesure où l'espace mémoire des tâches n'est jamais libéré, la pile se remplit jusqu'à épuisement de l'espace disponible.

La deuxième implémentation

Cette représentation se différencie de la première par le fait qu'elle dispose d'une méthode `vPortFree()` qui libère la mémoire allouée pour une tâche et peut l'allouer de nouveau à une autre tâche. Il faut que la taille du bloc mémoire de la nouvelle tâche soit au maximum égale à la taille du bloc de l'ancienne tâche.

La troisième implémentation

Il ne s'agit ici que d'une redéfinition de `Malloc()` et `Free()` mais où la sécurité a été augmentée en suspendant toutes les tâches pendant la durée de l'opération sur la mémoire.

9.6 Services fournis et domaines d'application

Grâce à son très faible encombrement mémoire et son portage sur de nombreuses architectures matérielles, FreeRTOS est principalement utilisé comme système d'exploitation

embarqué. Il a été implémenté sur des périphériques mobiles et est souvent utilisé sur les processeurs ARM.

Etant un système d'exploitation purement orienté temps réel et livré sans application, il sert souvent de socle pour le développement d'API spécifiques et/ou propriétaires. Il est donc utilisé dans des domaines très variés où la contrainte temps réel est forte comme par exemple: les appareils de surveillance médicale, les outils de contrôle sismique et d'environnement ou encore le pilotage d'appareils et de robots industriels. Ses fonctionnalités de gestion des tâches permettent de garantir l'intégrité des données collectées en temps réel par l'utilisation des plus hauts niveaux de priorités.

Le domaine technique dans lequel ce système temps réel est le plus utilisé est le réseau, et plus particulièrement pour la transmission de données sur des réseaux sans fils. Pour des réseaux à très haute fréquence comme le WPAN, sur des architectures matérielles à très faible capacité mémoire, des phénomènes d'overflow interviennent durant les échanges pour des systèmes à boucle unique. Cependant l'implémentation de ces échanges sous forme de différentes tâches complique le développement. Grâce à sa gestion des priorités et son ordonnanceur multitâche, FreeRTOS permet d'éliminer ce type de problèmes. Chaque tâche ayant un bloc réservé, le phénomène d'overflow est supprimé car il n'y a plus de dépassement mémoire par manque de ressources.

Pour les applications de contrôle comme l'eye-tracking à base de matériel embarqué à ressources limitées, FreeRTOS fournit un core pour le système de base gérant le matériel permettant d'y ajouter des applications spécifiques. Là encore c'est surtout l'ordonnanceur et son système de gestion et de priorité des tâches qui est primordial pour ce type d'application. Le firmware de ce périphérique mobile est composée de 3 couches, la HAL qui gère la couche d'abstraction matérielle, la DEL qui gère les composants additionnels au processeur et la TAL qui est la partie tâches qui gère FreeRTOS. Ce type d'application est destinée à s'adapter à l'utilisateur en fonction de ces réactions perçues au travers des mouvements de ses yeux.

FreeRTOS est très utilisé aussi pour l'implémentation de pile réseau et est souvent associé à uIP. Pour les périphériques mobiles comme les téléphones, on peut même le retrouver pour la gestion de la vidéo. On l'utilise aussi souvent pour des implémentations de couche réseau MAC comme le protocole 802.15.4 très utilisé pour les réseaux de capteurs sans fil. Une des forces de FreeRTOS est aussi son aspect open-source et le fait qu'il permet l'implémentation de couches réseaux IP très légères et facilement portable comme uIP ou lwIP.

Un autre grand domaine d'application est le réseau de capteurs sans fils. Ces systèmes consistent en un ensemble de capteurs transmettant leurs données à un nœud pour éventuellement les envoyer à un système central. Ce type de réseau est présent dans le domaine

médical pour monitorer les patients ou dans l'agriculture pour localiser et surveiller les élevages.

FreeRTOS est le système d'exploitation temps réel de prédilection dans ce domaine car les capteurs consomment très peu d'énergie et disposent de ressources RAM très limitées.

La consommation électrique est aussi un argument en faveur de FreeRTOS. Les périphériques mobiles de toutes sortes ont en commun cette contrainte essentielle, et ce système d'exploitation permet un traitement temps réel tout en garantissant un minimum de consommation électrique.

Avec la convergence des différentes technologies et leur miniaturisation, FreeRTOS permet d'allier des implémentations de piles réseau simples et efficaces aux besoins d'économie d'énergie. Les nouveaux périphériques mobiles comme les téléphones en sont un bon exemple. Cette solution logicielle permet aussi d'abaisser fortement les coût de fabrication.

Ayant été porté sur beaucoup d'architectures et avec le développement de carte notamment à base de FPGA intégrant des cartes réseau, FreeRTOS permet d'avoir des systèmes avec un système adapté permettant de se focaliser sur les objectifs finaux du projet.

Cependant étant sous licence GPL, même s'il permet le développement d'applications propriétaires, son code source doit rester ouvert et il est donc privilégié dans le domaine de la recherche et de l'enseignement. Ses concurrents propriétaires comme QNX sont le plus souvent choisis dans le monde de l'industrie.

Dans le domaine de l'enseignement là aussi, les différentes utilisations sont nombreuses. FreeRTOS est utilisé pour étudier l'implémentation d'ordonnanceur, la gestion de tâches et la programmation modulaire. Il permet également de développer des applications dans le domaine de l'électronique comme la lecture de température et son affichage.

FreeRTOS a aussi fait naître des frameworks grâce à son code ouvert, sa faible taille, ses possibilités de mise à l'échelle ainsi que son extensibilité. Certains de ces frameworks sont utilisés dans le secteur automobile.