

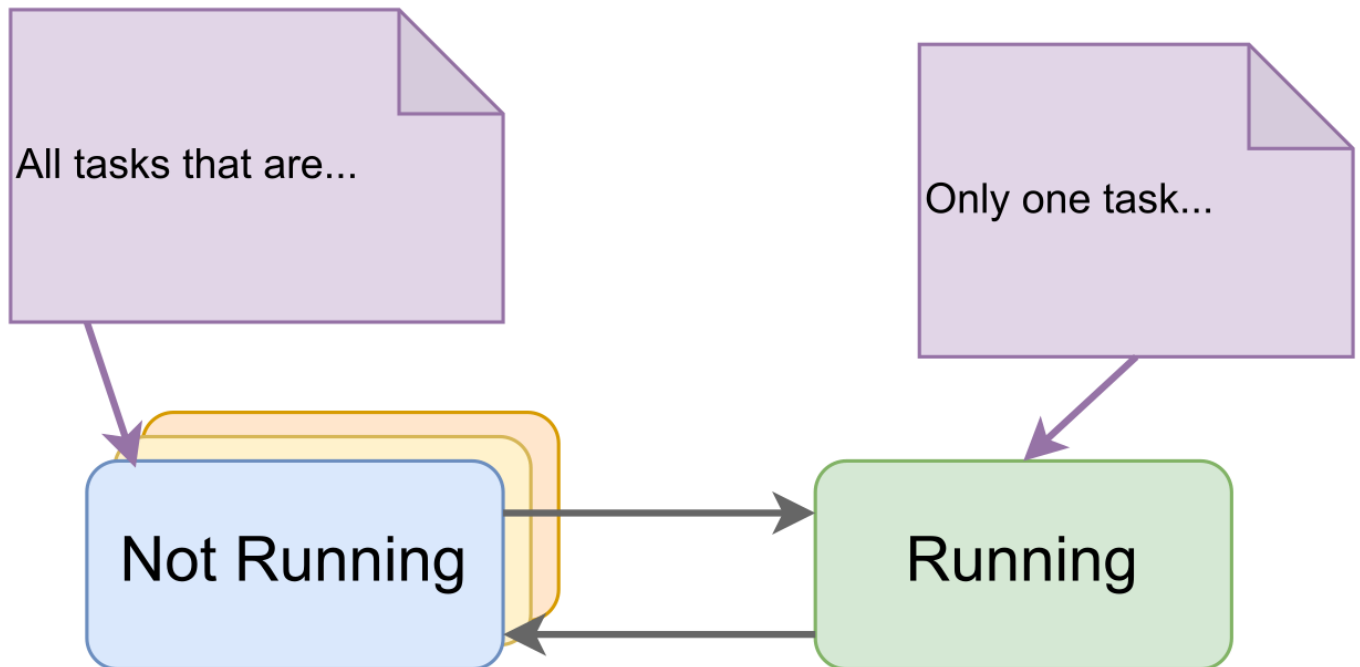
# Introduction aux systèmes embarqués II

Le livre [Mastering the FreeRTOS™ Real Time Kernel](#), a été utilisé pour cet article. Je reprendrais certains anglicisme par cohérence avec les sources et la documentation officielle.

## FreeRTOS

Une application peut être constituée de plusieurs tâches. Si le microcontrôleur est de type monoprocesseur, seule une tâche peut être exécutée pour un temps donné. En première approche, nous pouvons classer les tâches en deux états :

- **Running**
- **Not Running**



Ce modèle simplifié va être utilisé dans un premier temps.

- Dans l'état **Running** le processeur exécute le code associé à la tâche.
- Dans l'état **Not Running**, la tâche est dormante (en veille) son status a été sauvegardé comme 'prête' (ready) pour que l'exécution puisse être reprise quand le scheduler

décidera de la replacer dans l'état *Running*.

Une tâche passant de l'état *Not Running* à *Running* est considérée comme étant `switched in` ou `swapped in`.

Au contraire, une tâche passant de l'état *Running* à *Not Running* est dite comme étant `switched out` ou `swapped out`.

L'ordonnanceur de FreeRTOS est la seule entité permettant de réaliser un `switch in` ou `out` d'une tâche.

Quand une tâche est en reprise d'exécution, elle reprend depuis la dernière instruction effectuée après avoir quitté le l'état de *Running*. C'est la commutation de contexte.

# Problématique

Que se passe-t-il si une tâche A (task A) n'a rien à faire car elle est en attente d'une donnée / signal ... ? et imaginons que la tâche A est de même priorité que les tâches B et C.

L'ordonnanceur Round Robin va placer Task A en *Running*

- attendre la fin du quantum pour Task A
- placer Task B en *Running* durant un quantum
- placer Task C en *Running* durant un quantum
- placer Task A en *Running* qui aura peut-être l'information nécessaire pour faire autre chose qu'attendre, ou pas.

On remarque que l'on est pas très efficace à placer en *Running* une tâche qui ne fait rien mais le système reste opérationnel.

Que se passe-t-il si une tâche A (task A) n'a rien à faire car elle est en attente d'une donnée / signal et que cette tâche A est de priorité supérieure aux tâches B et C?

L'ordonnanceur Round Robin va placer Task A en *Running*

- attendre la fin du quantum pour Task A
- comme Task A est toujours ready et de priorité la plus élevée,
  - l'ordonnanceur replace Task A en *Running* pour un quantum

Task A empêche les tâches de priorité inférieures de s'exécuter.

Pour rendre les tâches utiles, elles doivent être réécrites pour être pilotées par des événements. Une tâche événementielle a un traitement à effectuer uniquement après l'occurrence de l'événement qui la déclenche. La tâche ne peut pas passer à l'état *Running* avant que cet événement ne se produise.

Utiliser des tâches événementielles signifie que les tâches peuvent être créées à différentes priorités sans que les tâches les plus prioritaires privent toutes les tâches de priorité inférieure du temps de traitement.

## L'état bloqué

Une tâche qui attend un événement est dite à l'état "Bloqué"

Les tâches peuvent passer à l'état Bloqué pour attendre deux types d'événements différents :

1. Événements temporels (Délai)
2. Événements de synchronisation — lorsque les événements proviennent d'une autre tâche ou interruption

Les événements de synchronisation FreeRTOS : sémaphores, mutex, files de messagerie, groupes d'événements

La possibilité de blocage des tâches est fondamentale : *C'est le seul moyen pour des tâches de priorités inférieures d'être élues*

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 paramétrée avec un timeout. 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.

## L'état suspendu

Une tâche peut être volontairement placée dans l'état "suspendu" par appel à la fonction API `vTaskSuspend()`. Elle sera alors totalement ignorée par l'ordonnanceur et ne consommera plus aucune ressource jusqu'à ce qu'elle soit retirée de l'état avec `vTaskResume()` et remise dans un état "prêt".

La plupart des applications n'utilisent pas l'état Suspendu.

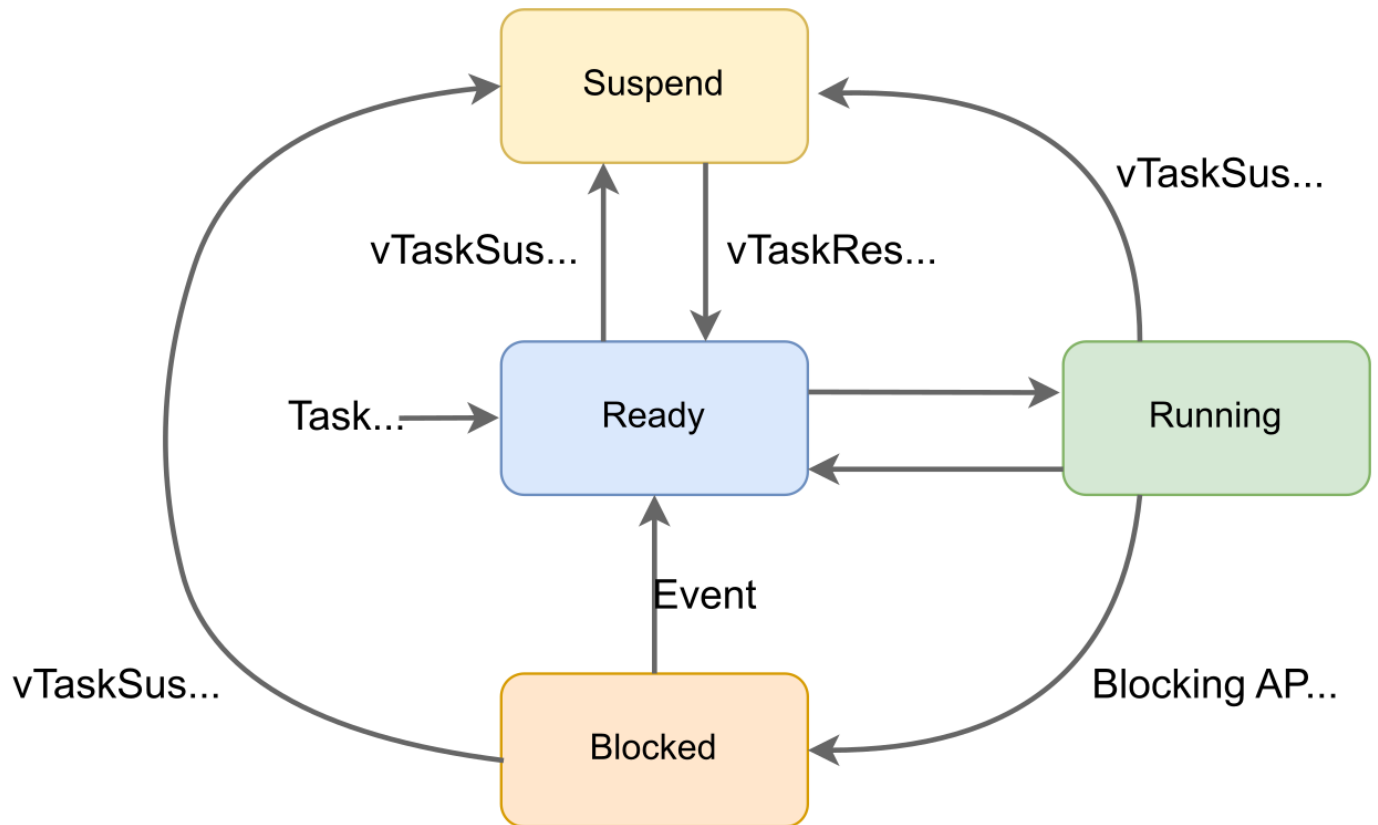
## L'état prêt

Les tâches qui sont à l'état *Not Running* mais qui ne sont :

- ni bloquées
- ni suspendues

sont dites à l'état Prêt -> READY.

- Elles ont obtenu toutes les ressources nécessaires pour s'exécuter, sauf le processeur, et donc "prêtes" à fonctionner
- Une tâche est initialement placée dans l'état Ready après sa création
- Une tâche Ready passe à l'état Running suivant la politique d'ordonnancement (priorité, round-robin)



## L'état Running

Une tâche obtient le processeur lorsqu'elle passe dans l'état Running

- Le code d'une tâche est exécuté uniquement lorsqu'elle est dans cet état
- Une tâche sort de l'état élu dans les situations suivantes :
  - Passage dans l'état Prêt :
    - Préemption par une tâche de priorité supérieure
    - La durée du quantum est atteinte (ordonnancement « round-robin »)
  - Passage dans l'état bloqué :
    - Accès à une ressource non disponible
    - Attente d'un événement
    - Suspension de l'exécution par introduction d'un délai

Une commutation de contexte est effectuée quand une tâche sort ou entre dans l'état élu

# L'état supprimé

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

La tâche 'IDLE' est créée lors du démarrage de l'ordonnanceur et se voit assigner la plus petite priorité possible.

La tâche Idle peut avoir plusieurs fonctions à remplir dont :

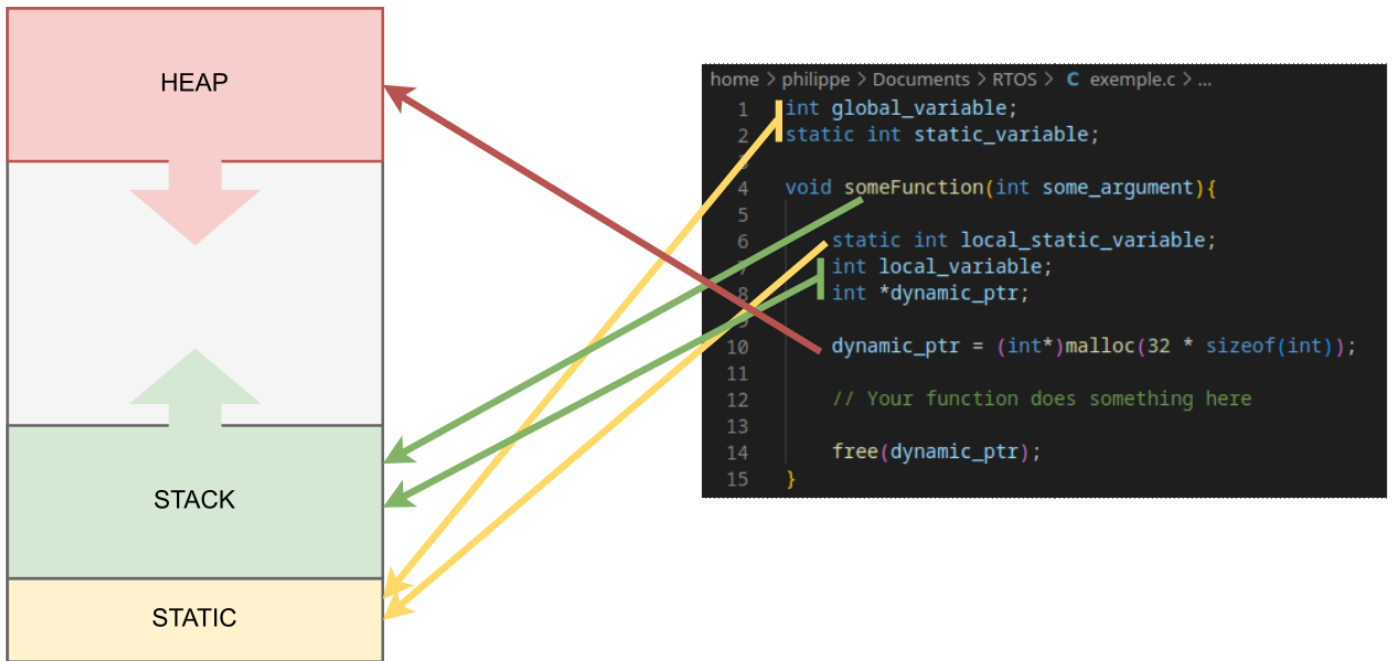
- Libérer l'espace occupé par une tâche supprimée.
- Placer le micro-contrôleur en veille afin d'économiser l'énergie du système lorsqu'aucune tâche applicative n'est en exécution.
- Mesurer le taux d'utilisation du processeur.

# Allocation mémoire

## Rappel allocation mémoire pour une fonction

- Les constantes, les variables globales et variables statiques sont stockées en STATIC
- les variables locales sont stockées dans la STACK (pile)
- les variable dynamiques sont stockées dans la HEAP (tas)

# Memory allocation



## Allocation mémoire dans FreeRTOS

Les tâches et objets Kernel sont placées dans la HEAP

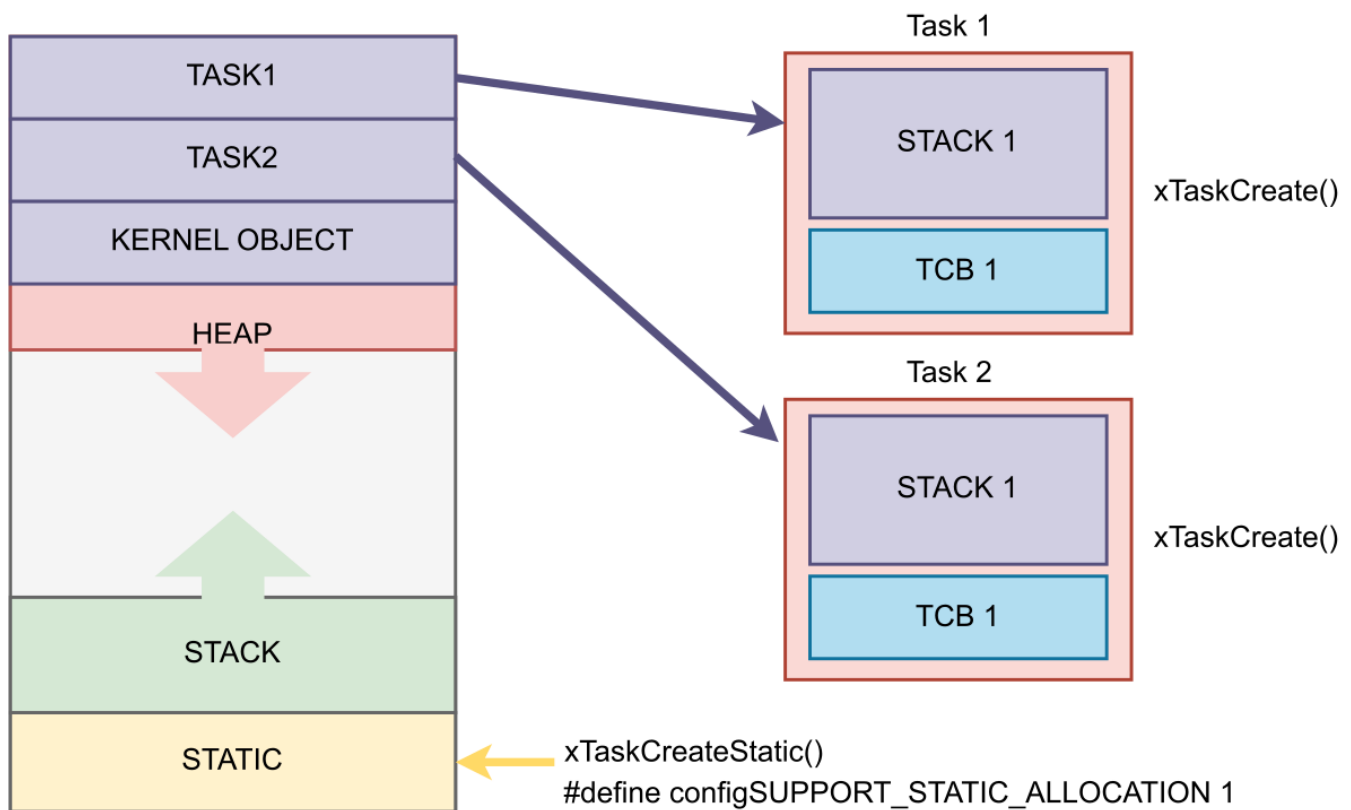
À la création d'une tâche, FreeRTOS crée et remplit la TCB correspondant à la tâche (Task control Block) qui contient toutes les informations nécessaires afin de spécifier et de représenter une tâche. Les Éléments associés à une tâche sont :

- La routine
  - Instructions (code) exécutées par la tâche
- Le niveau de priorité
  - Paramètre pris en compte par l'ordonnanceur
- L'identifiant
  - Attribué par le système
  - Utilisé comme paramètre dans les fonctions de gestions des tâches
- La STACK (pile)
  - Zone mémoire privée (stockage des données locales de la routine)

Chaque tâche possède sa propre STACK

FreeRTOS propose différentes stratégies de gestion de la HEAP, -> se référer à la documentation

# Memory allocation



## Gestion des tâches

Le système d'exploitation, via l'API, fournit des fonctions permettant de gérer les tâches

- Durée de vie
  - Création
  - Destruction
- Contrôle de l'ordonnancement
  - Introduction d'un délai d'attente
  - Modification/relecture de la priorité
  - Suspension de l'exécution

## Création d'une tâche

xTaskCreate( )

Paramètres	
pvTaskCode	Pointeur vers la routine de la tâche
pcName	Chaîne de caractère spécifiant le nom de la tâche (optionnelle, utilisée pour le débogage)

Paramètres	
usStackDepth	Taille de la pile associée à la tâche
pvParameters	Pointeur vers une structure de paramètres passés à la routine de la tâche
uxPriority	Priorité initiale de la tâche
pvCreatedTask	Pointeur vers une variable contenant l'identifiant de la tâche après création, au retour de la fonction

## Exemple de création de tâche en FreeRTOS Vanilla

L'exemple ci-dessous est codé suivant les fonctions FreeRtos sans modifications tierces, qu'on désigne sous le jargon "Vanilla". FreeRTOS pour l'ESP32 est en partie modifié spécifiquement pour cette cible, de même pour le STM32 avec la surcouche CMSIS

```
int main( void )
{
    xTaskHandle hTask1;

    xTaskCreate( Task1, "Sample Task",
        1024, NULL, 2, &hTask1);      // Création de la tâche
    ...                               // taille de Stack 1024 , priorité 2
                                     // Pas de passge de structures de paramètres par pointeur
                                     // pointeur vers variable contenant l'identifiant de la
tâche

    vTaskStartScheduler();            // Démarrage de l'ordonnanceur

    return 0;
}

static void Task1(void * pvParameter) // routine de la tâche
{
    int i;        // variable déclarée sur la pile de la tâche
    for(;;)       // boucle infinie
    {
        ...
    }
}
```

## Destruction d'une tâche



vTaskDelete()

Paramètres	
pxTask	Identifiant de la tâche à détruire

Rappel : la libération des ressources n'est pas immédiate, la tâche passera dans l'état supprimée et quand la tâche de plus faible priorité IDLE sera exécutée, la mémoire sera libérée.

## Exemple de destruction d'une tâche FreeRTOS Vanilla

```
int main( void )
{
    ...
    xTaskCreate( Task1, "Task 1",
        1024, NULL, 2, NULL);          // Création de la tâche 1
    ...
}

static void Task1(void * pvParameter)
{
    xTaskHandle hTask2;
    xTaskCreate( Task2, "Task 2",
        1024, NULL, 3, &hTask2);      // création de la tâche 2
    // Do some work
    ...
    vTaskDelete( hTask2);             // Destruction de la tâche 2
    // Do some work
    ...
    vTaskDelete( NULL);               // Destruction de la tâche 1
}

static void Task2(void * pvParameter)
{
    for(;;)
    {
        ...
    }
}
```

# Gestion du temps

Une tâche peut se placer dans l'état bloqué pendant une durée spécifiée en appelant `vTaskDelay`

Paramètres	
<code>xTicksToDelay</code>	Délai

Tous les délais spécifiés dans les fonctions FreeRTOS sont exprimés en « tick »

- Un « tick » correspond à la période de l'horloge système
  - Cette horloge, servant de base de temps à l'ordonnanceur, est générée à partir d'un périphérique de type timer.
  - L'horloge système émet des « system tick ».
- La valeur du « tick » est dépendante du matériel et du contexte de mise en œuvre du système temps-réel

---

Revision #2

Created 5 July 2023 13:47:41 by Philippe Celka

Updated 5 July 2023 13:51:07 by Philippe Celka