



TP 2 : LES TACHES - 1

Fonctions tâches, état de haut niveau, création et Tick interrupt.

Matériel nécessaire : **Arduino UNO R4 WiFi**, câble USB, environnement Arduino v2,

FreeRTOS n'est pas à installer car inclus dans le package UNO R4

1. Les types de données et le style de codage.

Chaque port de FreeRTOS possède un fichier **portmacro.h** qui contient la définition de deux types de données : **TickType_t** et **BaseType_t**.

- **TickType_t** : **FreeRTOS** configure une interruption périodique appelée interruption *tick*. Le nombre d'interruptions *tick* qui ont eu lieu depuis le lancement de l'application porte le nom de *tick count*. Le *tick count* est utilisé pour mesurer une durée. L'intervalle de temps entre deux interruptions *tick* est appelé *tick period*. Le temps est spécifié en multiple de *tick period*. Le type **TickType_t** est utilisé pour maintenir une valeur du *tick count* et pour spécifier une durée. En fonction de la valeur de **configUSE_16_BIT_TICKS** dans **FreeRTOS.h**, la taille et le type de **TickType_t** peuvent varier :

configUSE_16_BIT_TICKS	taille	type	utilisation
1	16 bits	uint16_t	Architectures 8 bits et 16 bits.
0	32 bits	uint32_t	Architectures 32 bits.

- **BaseType_t** : il est défini comme étant le type de donnée le plus efficace pour l'architecture. Typiquement c'est un type de 32 bits pour une architecture 32 bits, 16 bits pour une architecture 16 bits et 8 bits pour une architecture 8 bits. **BaseType_t** est généralement utilisé pour des types retournés (par des fonctions ou API) qui contiennent une plage de valeurs très limitées, et des types booléens comme **pdTRUE/pdFALSE** (cf. plus loin dans ce cours).

Certains compilateurs considèrent les variables **char** comme non signées et d'autres comme étant signées. **FreeRTOS** qualifie chaque utilisation de **char** avec l'attribut *signed* ou *unsigned*, sauf si le char est utilisé pour contenir un caractère ASCII ou un pointeur vers un **char** destiné à pointer vers une chaîne de caractères.

Les noms de variables.

Les variables sont préfixées avec leurs type : 'c' pour **char**, 's' pour **int16_t** (short), 'l' pour **int32_t** (long), et 'x' pour le type **BaseType_t** et tout autre type non standard (structures, task handle, queue handle, etc...).

Si une variable est **unsigned**, elle sera préfixée avec un 'u'. Si une variable est un **pointeur**, elle sera préfixée avec un 'p'. Par exemple, une variable de type **uint_8** sera préfixée avec 'uc' et une variable de type **pointeur vers un char** sera préfixée avec 'pc'.

Les noms de fonctions et API. (Application Programming Interface)

Les fonctions sont préfixées avec à la fois le type retourné et le fichier dans lequel elles sont définies. Par exemple :

- **vTaskPrioritySet()** retourne un **void** et est définie dans **task.c**.
- **xQueueReceive()** retourne une variable de type **BaseType_t** et est définie dans **queue.c**.
- **pvTimerGetTimerID()** retourne un **pointeur vers void** et est défini dans **timer.c**.

Les fonctions private seront préfixées avec '**prv**'.

Les noms des macros.

La plupart des macros sont écrites avec des lettres majuscules et préfixées avec des lettres minuscules qui indiquent l'endroit où la macro est définie. La table suivante fournit une liste des préfixes :

Prefix	Location of macro definition
port (for example, portMAX_DELAY)	portable.h or portmacro.h
task (for example, taskENTER_CRITICAL())	task.h
pd (for example, pdTRUE)	projdefs.h
config (for example, configUSE_PREEMPTION)	FreeRTOSConfig.h
err (for example, errQUEUE_FULL)	projdefs.h

Les macros définies à la table suivante sont utilisées dans les codes sources :

Macro	Value
pdTRUE	1
pdFALSE	0
pdPASS	1
pdFAIL	0

PS : les informations de ce polycopié sont extraites du paragraphe 1.5 du document *Mastering the FreeRtos Real Time Kernel*. Disponible sur le site officiel de FreeRTOS :

https://freertos.org/Documentation/161204_Mastering_the_FreeRTOS_Real_Time_Kernel-A_Hands-On_Tutorial_Guide.pdf

FreeRTOS propose 3 types d'ordonnancements qui seront présentés au chapitre **LES TACHES (3/3)**. Pour la suite, on utilise l'algorithme d'ordonnement de type **préemptif à priorité avec découpage temporel** (*Prioritized Pre-emptive Scheduling with Time Slicing*). Pour cela, dans le fichier **FreeRTOSConfig.h**, mettre les champs suivants comme suit et enregistrer le fichier :

The FreeRTOSConfig.h settings that configure the kernel to use Prioritized Pre-emptive Scheduling with Time Slicing

Constant	Value
configUSE_PREEMPTION	1
configUSE_TIME_SLICING	1

NB : Ce fichier FreeRTOSConfig.h à modifier est placé dans le répertoire :

C:\Users\monCompte\AppData\Local\Arduino15\packages\arduino\hardware\renesas_uno\1.0.4\libraries\Arduino_FreeRTOS\src\

Vérifier les lignes 21 pour configUSE_PREEMPTION et 96 pour configUSE_TIME_SLICING

2. Les fonctions tâches.

Elles sont implémentées comme des fonctions en langage C. Leur prototype doit respecter une syntaxe stricte. Elle retourne un *void* et accepte un paramètre de type *pointeur vers void*. Leur prototype est le suivant :

```
void ATaskFunction( void *pvParameters );
```

The task function prototype

Chaque tâche est un petit morceau de code. Elle ne possède pas de point d'entrée, s'exécutera indéfiniment dans une boucle et n'en sortira pas. La structure typique d'une tâche est donnée à la suite (code générique, non spécifique à Arduino Uno):

```
void ATaskFunction( void *pvParameters )
{
    /* Variables can be declared just as per a normal function. Each instance of a task
    created using this example function will have its own copy of the lVariableExample
    variable. This would not be true if the variable was declared static - in which case
    only one copy of the variable would exist, and this copy would be shared by each
    created instance of the task. (The prefixes added to variable names are described in
    section 1.5, Data Types and Coding Style Guide.) */
    int32_t lVariableExample = 0;

    /* A task will normally be implemented as an infinite loop. */
    for( ;; )
    {
        /* The code to implement the task functionality will go here. */

        /* Should the task implementation ever break out of the above loop, then the task
        must be deleted before reaching the end of its implementing function. The NULL
        parameter passed to the vTaskDelete() API function indicates that the task to be
        deleted is the calling (this) task. The convention used to name API functions is
        described in section 0, Projects that use a FreeRTOS version older than V9.0.0
        must build one of the heap_n.c files. From FreeRTOS V9.0.0 a heap_n.c file is only
        required if configSUPPORT_DYNAMIC_ALLOCATION is set to 1 in FreeRTOSConfig.h or if
        configSUPPORT_DYNAMIC_ALLOCATION is left undefined. Refer to Chapter 2, Heap Memory
        Management, for more information.
        Data Types and Coding Style Guide. */
        vTaskDelete( NULL );
    }
}
```

The structure of a typical task function

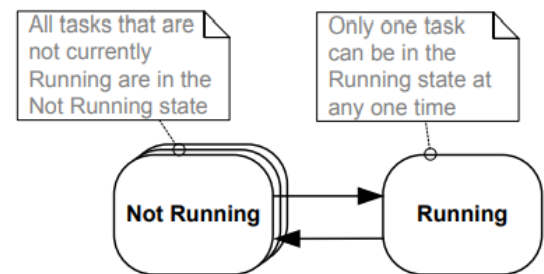
Les tâches **FreeRTOS** ne permettent pas de retourner quelque chose, elles ne doivent pas posséder d'instruction **return**. Elles ne doivent pas exécuter quelque chose dans la boucle sans fin. Si une tâche n'est plus utile, elle doit être explicitement supprimée.

Une unique définition de fonction de tâche peut être utilisée pour créer plusieurs tâches – chaque tâche créée étant une instance d'exécution séparée, avec sa propre pile (**stack**) et sa propre copie de variables automatiques (**stack**) définie dans la tâche elle-même.

3. Les états hauts niveaux des tâches.

Une application consiste en plusieurs tâches. Si le processeur contient un seul cœur (**core**), alors une seule tâche peut être exécutée à un instant donné. Cela implique qu'une tâche peut exister seulement dans l'un des deux états : **Running** et **Not Running**. Ce modèle simpliste est utilisé en premier lieu par souci de simplification. Nous verrons par la suite que l'état **Not Running** contient des sous états.

Lorsqu'une tâche est dans l'état **Running**, le processeur exécute le code de la tâche. Lorsqu'une tâche est dans l'état **Not Running**, la tâche est dormante, son état a été sauvé prêt à reprendre son exécution la prochaine fois que l'ordonnanceur décidera de la faire entrer dans l'état **Running**. Lorsqu'une tâche reprend son exécution, elle le fait à partir de l'instruction qui était sur le point d'être exécutée avant de quitter l'état **Running**.



Top level task states and transitions

Une tâche qui passe de l'état **Not Running** vers l'état **Running** est dite '**switch in**' ou '**swapped in**'. Dans le sens inverse, on utilisera les expressions '**switch out**' ou '**swapped out**'. L'ordonnanceur de **FreeRTOS** est la seule entité capable de réaliser ces commutations ('**switch in**' et '**switch out**').

NB : Le modèle complet des états et transitions des tâches FreeRTOS est en ANNEXE p17

4. La création de tâches.

L'API utilisé pour créer une tâche est **xTaskCreate()**. C'est probablement l'API le plus complexe, malheureusement le premier que vous rencontrez mais il est fondamental et il est impossible de l'éviter. Tous les programmes l'utilisent. Son prototype est le suivant :

```

BaseType_t xTaskCreate( TaskFunction_t pvTaskCode,
                        const char * const pcName,
                        uint16_t usStackDepth,
                        void *pvParameters,
                        UBaseType_t uxPriority,
                        TaskHandle_t *pxCreatedTask );
  
```

The xTaskCreate() API function prototype

La description des champs est indiquée dans le tableau suivant (pour plus de précisions, se référer aux pages 50 à 53 du document *Mastering the FreeRTOS Real Time Kernel*):

Nom du Paramètre/ valeur retournée	Description
<i>pvTaskCode</i>	Pointeur vers une fonction qui implémente la tâche (juste le nom de la fonction).
<i>pcName</i>	Nom descriptif pour la tâche. Inutilisé dans <i>FreeRTOS</i> . Utile pour le débogage. Permet d'identifier une tâche avec un nom « humain » plutôt qu'un handle.
<i>usStackDepth</i>	Taille de la pile (stack) allouée par le noyau à la tâche lors de sa création. Il s'agit du nombre de mots que la pile pourra contenir, pas le nombre d'octets.
<i>pvParameters</i>	La fonction tâche accepte un paramètre de type <i>void*</i> , c'est-à-dire <i>pointeur vers void</i> . La valeur affectée à <i>pvParameters</i> est la valeur transmise à la tâche.
<i>uxPriority</i>	Définit la priorité avec laquelle la tâche va s'exécuter. Les priorités vont de 0, qui est la priorité la plus faible, jusqu'à (<i>configMAX_PRIORITIES-1</i>), et <i>configMAX_PRIORITIES</i> est une constante utilisateur définie dans le fichier <i>FreeRTOSConfig.h</i> .
<i>pxCreatedTask</i>	Utilisé pour passer un handle à la tâche créée. Ce <i>handle</i> pourra être utilisé pour référencer la tâche dans les appels des API. Si l'application n'utilise pas <i>ce handle</i> , ce champ sera mis à <i>NULL</i> .
Returned value	Deux valeurs possibles : <ul style="list-style-type: none"> • <i>pdPass</i> : indique que la tâche a été créée avec succès. • <i>pdFail</i> : indique que la tâche n'a pas été créée avec succès en raison de RAM insuffisante disponible pour le TCB et la pile de la tâche.

L'**exemple 1** suivant montre les étapes pour créer deux tâches simples, puis le démarrage de l'exécution des tâches. Les tâches impriment périodiquement une chaîne de caractères dans la console, en utilisant une simple boucle pour créer un retard périodique.

Les deux tâches possèdent la même priorité, sont identiques excepté pour la chaîne de caractères affichée et le retard périodique.

Ex2.1 : Tester ce programme et l'expliquer. (fichier « exemple 1- Création de deux tâches.ino ») :

```

#include <Arduino_FreeRTOS.h>

// define two Tasks
void Task1( void *pvParameters );
void Task2( void *pvParameters );

// the setup function runs once when you press reset or power the board
void setup() {
    // initialize serial communication at 9600 bits per second:
    Serial.begin(9600);

    while (!Serial) {
        ; // wait for serial port to connect. Needed for native USB, on LEONARDO, MICRO,
    }

    // Now set up two Tasks to run independently with the same priority.
    xTaskCreate(
        Task1
        , "Task 1" // A name just for humans
        , 128 // This stack size can be checked & adjusted by reading the Stack Highwater
        , NULL //Parameters for the task
        , 1 // Priority 1, with 3 (configMAX_PRIORITIES - 1) being the highest, and 0 being the lowest.
        , NULL ); //Task Handle

    xTaskCreate(
        Task2
        , "Task 2" // A name just for humans
        , 128 // Stack size
        , NULL //Parameters for the task
        , 1 // Priority
        , NULL ); //Task Handle

    vTaskStartScheduler(); //Start Scheduler
}

void loop()
{
    // Empty. Things are done in Tasks.
}


/*-----*/
/*----- Tasks -----*/
/*-----*/

void Task1( void *pvParameters ) // This is a Task 1.
{
    // prints Tache 1 to the serial monitor
    volatile uint32_t cnt ;
    for (;;) // A Task shall never return or exit.
    {
        //Serial.print(millis());
        Serial.println(" Tache 1");
        //delay(1000);
        for(cnt=0;cnt<=2400000;cnt++);
    }
}

void Task2( void *pvParameters ) // This is a Task 2.
{
    // prints Tache 2 to the serial monitor
    volatile uint32_t cnt ;
    for (;;) // A Task shall never return or exit.
    {
        //Serial.print(millis());
        Serial.println(" Tache 2");
        //delay(500);
        for(cnt=0;cnt<=1200000;cnt++);
    }
}

```

Ex2.1.1 : Interprétez les informations de la console. Activer

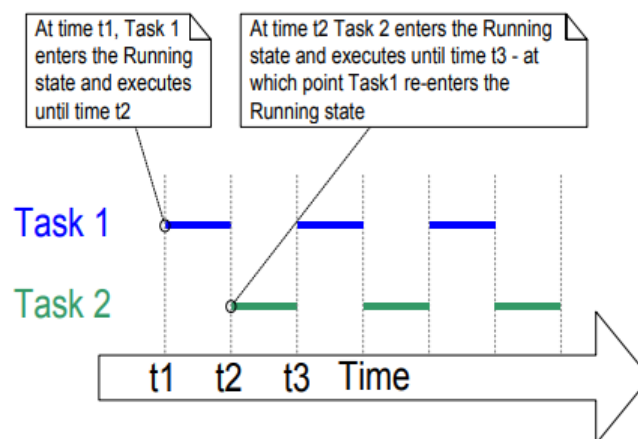
l'horodatage de la console texte, icône  afin d'avoir les informations temporelles d'affichage des messages.

```
03:47:23.893 -> Tache 1
03:47:24.191 -> Tache 2
```

Ex2.1.2 : Mesurer aproximativement le temps de boucle de la tâche et celui de la tâche 2, commenter.

Expliciter vos justifications dans le rapport de TP

On observe que les deux tâches semblent s'exécuter simultanément ; en réalité, comme les deux tâches s'exécutent sur le même processeur, ce n'est pas le cas. En réalité, les deux tâches sont rapidement placées et sorties de l'état **Running**. Les deux tâches s'exécutent avec la même priorité et donc partagent le temps d'exécution sur le processeur. Le véritable modèle d'exécution est présenté à la figure suivante :



The actual execution pattern of the two Example 1 tasks

Ex2.2 : Créer une troisième tâche de même priorité qui affiche votre prénom, environ toutes les 2 secondes. La tester.

Dans l'exemple précédent, les deux tâches sont créées à partir du `setup()`, avant de lancer l'ordonnanceur. Un API `vTaskStartScheduler()` est utilisé après la création des tâches pour lancer l'ordonnanceur.

ATTENTION pour la suite !

Avant l'arrivée du UNO R4, le lancement de l'ordonnanceur était automatique, les programmes qui suivent sont pour certains écrits pour l'IDE avec le UNO R3 qui lance donc automatiquement l'ordonnanceur.

Ici avec le UNO R4, il faut donc ajouter l'API `vTaskStartScheduler()` comme dans l'Exemple 1

Il faut également adapter, au nouveau processeur beaucoup plus rapide, la valeur du retard périodique, c'est à dire la valeur d'arrêt du compteur utilisé comme « `delay()` » non bloquant.

```
for(cnt=0;cnt<=2400000;cnt++); // ~ 1 seconde mais dépend fortement du nombre de tâche !
```

```
for(cnt=0;cnt<=1200000;cnt++); // ~ 0.5 seconde, dépend fortement du nombre de tâche !
```

Il est aussi possible de créer une tâche à partir d'autres tâches comme le montre l'exemple suivant (*fichier exemple 2- Création d'une tâche dans une autre*):

Ex2.3 : Tester ce programme et l'expliquer.

exemple2-Cr_ation_d_une_tache_dans_une_autre.ino

```

1  //--Exemple 2 - Création d'une tâche dans une autre -----
2  #include <Arduino_FreeRTOS.h>
3
4  // define two Tasks
5  void Task1( void *pvParameters );
6  void Task2( void *pvParameters );
7
8  // the setup function runs once when you press reset or power the board
9  void setup() {
10     // initialize serial communication at 9600 bits per second:
11     Serial.begin(9600);
12     while (!Serial) {
13         ; // wait for serial port to connect. Needed for native USB, on LEONARDO, MICRO,
14     }
15
16     // Now set up two Tasks to run independently with the same priority.
17     xTaskCreate(
18         Task1
19         , "Task 1" // A name just for humans
20         , 128 // This stack size can be checked & adjusted by reading the Stack Highwater
21         , NULL //Parameters for the task
22         , 1 // Priority 1, with 3 (configMAX_PRIORITIES - 1) being the highest, and 0 being the lowest.
23         , NULL ); //Task Handle
24
25     vTaskStartScheduler(); //Start Scheduler
26 }
27
28 void loop()
29 {
30     // Empty. Things are done in Tasks.
31 }
32
33 /*-----*/
34 /*----- Tasks -----*/
35 /*-----*/
36
37 void Task1( void *pvParameters ) // This is a Task 1.
38 {
39     // prints Tache 1 to the serial monitor
40     volatile uint32_t cnt ;
41     xTaskCreate(
42         Task2
43         , "Task 2" // A name just for humans
44         , 128 // Stack size
45         , NULL //Parameters for the task
46         , 1 // Priority
47         , NULL ); //Task Handle
48
49     for (;;) // A Task shall never return or exit.
50     {
51         Serial.println(" Tache 1");
52         //delay(1000);
53         for(cnt=0;cnt<=1200000;cnt++);
54     }
55 }
56
57 void Task2 (void *pvParameters ) // This is a Task 2.
58 {
59     // prints Tache 2 to the serial monitor
60     volatile uint32_t cnt ;
61
62     for (;;) // A Task shall never return or exit.
63     {
64         Serial.println(" Tache 2");
65         //delay(500);
66         for(cnt=0;cnt<=1200000;cnt++);
67     }
68 }
69

```


Ex2.4 : Créer une quatrième tâche (de même priorité) qui affiche votre NOM, environ toutes les 4 secondes, à partir de la troisième tâche de l'exercice 1. La tester.

**Ex2.5 : Ajouter une tâche permettant d'implémenter la fonction réalisée par l'Ex7 du TP1 :
Pilotage de la LED par l'ordinateur**

5. Utilisation du paramètre de la tâche.

Si on place le même retard périodique dans la tâche 2 que dans de la tâche 1, c'est-à-dire « `for(cnt=0; cnt<=2400000; cnt++);` » alors les tâches 1 et 2 sont identiques à l'exception de la chaîne de caractères émise vers la console.

Cette duplication peut être évitée en créant deux instances d'une implémentation d'une seule tâche. Le paramètre de la tâche peut être utilisé pour passer à chaque tâche la chaîne à imprimer.

Le listing suivant (*Exemple 3- passage de paramètre entre tâches*) contient le code de deux tâches. Notez que le paramètre des tâches subit une conversion forcée (*cast*) en *char** pour obtenir la chaîne de caractères à imprimer.

Ex2.6 : Tester ce programme et l'expliquer.

```
Exemple3-Passage_de_arametres_entre_taches$
#include <Arduino_FreeRTOS.h>
const char *pcTextForTask1 = "Tache 1 en execution";
const char *pcTextForTask2 = "Tache 2 en execution";

// define two Tasks
void Task1( void *pvParameters );
void Task2( void *pvParameters );

// the setup function runs once when you press reset or power the board
void setup() {

    // initialize serial communication at 9600 bits per second:
    Serial.begin(9600);

    while (!Serial) {
        ; // wait for serial port to connect. Needed for native USB, on LEONARDO, MICRO,
    }

    // Now set up two Tasks to run independently with the same priority.
    xTaskCreate(
        Task1
        , "Task 1" // A name just for humans
        , 128 // This stack size can be checked & adjusted by reading the Stack Highwater
        , (void*) pcTextForTask1//Parameters for the task
        , 1 // Priority 1, with 3 (configMAX_PRIORITIES - 1) being the highest, and 0 being the lowest.
        , NULL ); //Task Handle

    xTaskCreate(
        Task2
        , "Task 2" // A name just for humans
        , 128 // Stack size
        , (void*) pcTextForTask2 //Parameters for the task
        , 1 // Priority
        , NULL ); //Task Handle

    // Now the Task scheduler, which takes over control of scheduling individual Tasks, is automatically started.
}

void loop()
{
    // Empty. Things are done in Tasks.
}

```

```

void Task1( void *pvParameters ) // This is a Task 1.
{
    // prints Tache 1 to the serial monitor
    char *pcTaskName;
    pcTaskName=(char *)pvParameters;
    volatile uint32_t cnt ;

    for (;;) // A Task shall never return or exit.
    {
        Serial.println(pcTaskName);
        //delay(1000);
        for(cnt=0;cnt<=320000;cnt++);
    }
}

void Task2 (void *pvParameters ) // This is a Task 2.
{
    // prints Tache 2 to the serial monitor
    char *pcTaskName;
    pcTaskName=(char *)pvParameters;
    volatile uint32_t cnt ;

    for (;;) // A Task shall never return or exit.
    {
        Serial.println(pcTaskName);
        //delay(500);
        for(cnt=0;cnt<=320000;cnt++);
    }
}

```

On reprend le programme précédent mais en créant le code d'une seule tâche.

Ex2.7 : Tester ce programme et l'expliquer.

(fichier Exemple 4- Autre méthode de passage de paramètre entre tâches)

```

Exemple_4_-Autre_m_thode_de_passage_de_param_tres$
#include <Arduino_FreeRTOS.h>
const char *pcTextForTask1 = "Tache 1 en execution";
const char *pcTextForTask2 = "Tache 2 en execution";

// define two Tasks
void TaskFunction( void *pvParameters );

// the setup function runs once when you press reset or power the board
void setup() {

    // initialize serial communication at 9600 bits per second:
    Serial.begin(9600);
    while (!Serial) {
        ; // wait for serial port to connect. Needed for native USB, on LEONARDO, MICRO,
    }

    // Now set up two Tasks to run independently with the same priority.
    xTaskCreate(
        TaskFunction
        , "Task1" // A name just for humans
        , 128 // This stack size can be checked & adjusted by reading the Stack Highwater
        , (void*) pcTextForTask1//Parameters for the task
        , 1 // Priority 1, with 3 (configMAX_PRIORITIES - 1) being the highest, and 0 being the lowest.
        , NULL ); //Task Handle

```

```

xTaskCreate(
    TaskFunction
    , "Task 2" // A name just for humans
    , 128 // Stack size
    , (void*) pcTextForTask2 //Parameters for the task
    , 1 // Priority
    , NULL ); //Task Handle

    // Now the Task scheduler, which takes over control of scheduling individual Tasks, is automatically started.
}

void loop()
{
    // Empty. Things are done in Tasks.
}

/*-----*/
/*----- Tasks -----*/
/*-----*/

void TaskFunction( void *pvParameters ) // This is a Task 1.
{
    // prints Tache 1 to the serial monitor
    char *pcTaskName;
    pcTaskName=(char *)pvParameters;
    volatile uint32_t cnt ;

    for (;;) // A Task shall never return or exit.
    {
        Serial.println(pcTaskName);
        //delay(1000);
        for(cnt=0;cnt<=320000;cnt++);
    }
}

```

Même si une seule implémentation de tâche est créée, plusieurs instances de cette tâche sont possibles. Chaque nouvelle instance créée s'exécutera indépendamment sous le contrôle de l'ordonnanceur de **FreeRTOS**.

6. Les priorités des tâches.

Le paramètre **uxPriority** de l'API **xTaskCreate()** assigne une priorité initiale à la tâche qui vient d'être créée. Cette priorité peut être changée après le démarrage de l'ordonnanceur en utilisant l'API **vTaskPrioritySet()**.

Les priorités vont de **0**, qui est la priorité la plus faible, jusqu'à (**configMAX_PRIORITIES-1**), et **configMAX_PRIORITIES** est une constante utilisateur définie dans le fichier **FreeRTOSConfig.h**.

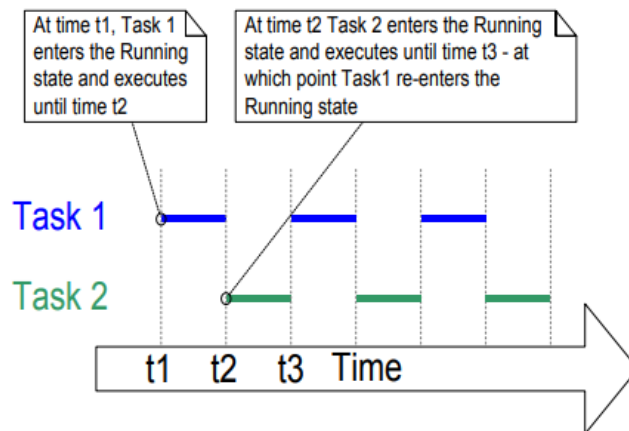
N'importe quel nombre de tâches peuvent partager la même priorité, ce qui fournit un maximum de flexibilité.

7. La mesure du temps et l'interruption Tick.

Dans le TP intitulé **LES TACHES (3 sur 3)**, les algorithmes d'ordonnancement décrivent une caractéristique optionnelle nommée '**time slicing**'. Le '**time slicing**' a été utilisé dans les exemples présentés jusqu'à présent et explique le comportement observé en sorties.

Dans les exemples, deux tâches étaient créées avec la même priorité, et les deux tâches étaient en mesure d'être exécutées. Chaque tâche était exécutée durant une tranche temporelle (**time**

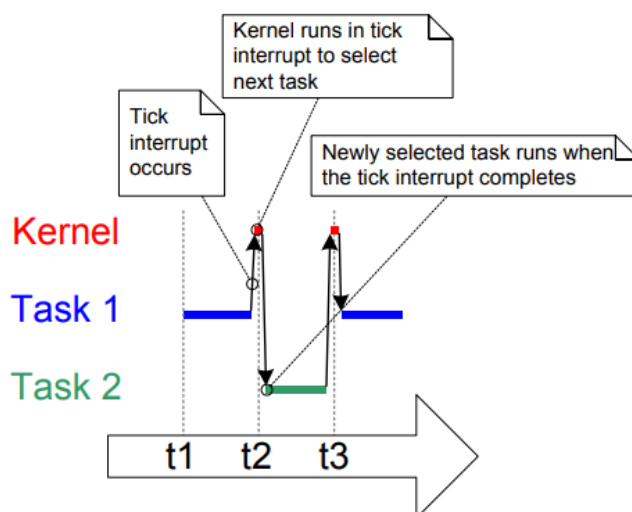
slice), entrant dans l'état **Running** au début du *time slice* puis sortant de l'état **Running** à la fin du *time slice*. Dans la figure suivante, l'intervalle de temps entre t1 et t2 est un *time slice*



Pour être capable de sélectionner la prochaine tâche à exécuter, l'ordonnanceur doit lui-même s'exécuter à la fin de chaque *time slice*. Une interruption périodique, appelée '*tick interrupt*', est utilisée à cette fin. La durée du *time slice* est imposée par la fréquence du '*tick interrupt*', qui est configurée par la variable `configTICK_RATE_HZ` dans le fichier `FreeRTOSConfig.h`. Par exemple, si la variable `configTICK_RATE_HZ` est fixée à 100 (Hz), alors le *time slice* sera de 10 ms. L'intervalle de temps entre deux '*tick interrupt*' est appelé '*tick period*'. Un '*time slice*' est égal à une '*tick period*'.

Notez que la valeur optimale de la variable `configTICK_RATE_HZ` dépend de l'application proposée et du processeur, et une valeur de 100 est fréquemment rencontrée.

La figure précédente peut être développée pour montrer l'exécution de l'ordonnanceur dans la séquence d'exécution. Cela est indiqué dans la figure suivante, dans laquelle la ligne du haut indique les instants où l'ordonnanceur s'exécute, et la flèche en trait fin montre la séquence d'exécution depuis une tâche vers le '*tick interrupt*', puis du '*tick interrupt*' vers une tâche différente.



The execution sequence expanded to show the tick interrupt executing

Les appels des API FreeRTOS spécifient toujours le temps en multiples de '*tick periods*', ce que l'on appelle généralement '*ticks*'. La macro `pdMS_TO_TICKS()` convertit le temps

spécifié en ms en un temps spécifié en '*ticks*'. La résolution disponible dépend de la valeur de la fréquence du *tick*, et la macro *pdMS_TO_TICKS()* ne peut pas être utilisée si la fréquence du *tick* est supérieure à 1 kHz (ou de manière équivalente si *configTICK_RATE_HZ* est supérieure à 1000).

La figure suivante montre la manière d'utiliser la macro *pdMS_TO_TICKS()* pour convertir un temps de 200 ms en son équivalent en *ticks*.

```
/* pdMS_TO_TICKS() takes a time in milliseconds as its only parameter, and evaluates
to the equivalent time in tick periods. This example shows xTimeInTicks being set to
the number of tick periods that are equivalent to 200 milliseconds. */
TickType_t xTimeInTicks = pdMS_TO_TICKS( 200 );
```

**Using the *pdMS_TO_TICKS()* macro to convert 200 milliseconds into an
equivalent time in tick periods**

Note : il est recommandé dans les applications d'utiliser cette API, ce qui permet de spécifier le temps en ms et de s'assurer que le temps spécifié restera le même si la fréquence du *tick* est changée.

La valeur du '*tick count*' est le nombre total de *tick interrupt* depuis le démarrage de l'ordonnanceur, en supposant qu'il n'y a pas eu d'*overflow* du '*tick count*'. Les applications utilisateurs n'ont pas à gérer ces *overflow* lorsqu'elles spécifient les durées (*delay period*), car la consistance du temps est gérée en interne sous *FreeRTOS*. Dans le TP intitulé **LES TACHES (3 sur 3)**, on précisera les configurations qui affectent le moment où l'ordonnanceur sélectionne une nouvelle tâche à exécuter et le moment où une *tick interrupt* s'exécutera.

Exemple : Expérimentation des priorités (fichier exemple 5- changement de priorité d'une tâche) .

L'ordonnanceur va toujours assurer que la tâche de plus haute priorité qui est capable de s'exécuter sera la tâche sélectionnée pour entrer dans l'état *Running*. Dans nos exemples, jusqu'à présent, deux tâches ont été créées avec la même priorité, elles entraient et sortaient de l'état *Running* à tour de rôle. L'exemple suivant montre ce qui arrive lorsque la priorité d'une des deux tâches est changée.

Ex2.8 : Tester ce programme et l'expliquer.

Exemple 5- changement de priorité d'une tâche

```

#include <Arduino_FreeRTOS.h>

// define two Tasks
void Task1( void *pvParameters );
void Task2( void *pvParameters );

// the setup function runs once when you press reset or power the board
void setup() {
    // initialize serial communication at 9600 bits per second:
    Serial.begin(9600);

    // Configurez maintenant deux tâches avec des priorités différentes
    xTaskCreate(
        Task1
        , "Task 1" // A name just for humans
        , 128 // This stack size can be checked & adjusted by reading the Stack Highwater
        , NULL //Parameters for the task
        , 3 // Priority 3, with 3 (configMAX_PRIORITIES - 1) being the highest, and 0 lowest.
        , NULL ); //Task Handle

    xTaskCreate(
        Task2
        , "Task 2" // A name just for humans
        , 128 // Stack size
        , NULL //Parameters for the task
        , 2 // Priority
        , NULL ); //Task Handle

    vTaskStartScheduler(); //Start Scheduler
}

void loop() {
    // Empty. Things are done in Tasks.
}

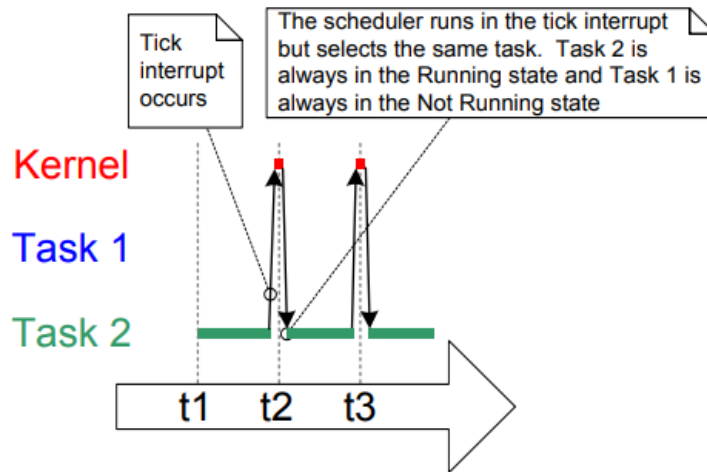
/*-----*/
/*----- Tasks -----*/
/*-----*/

void Task1( void *pvParameters ) // This is a Task 1.
{
    // prints Tache 1 to the serial monitor
    volatile uint32_t cnt ;
    for (;;) // A Task shall never return or exit.
    {
        Serial.println(" Tache 1 en execution");
        for(cnt=0;cnt<=2400000;cnt++);
    }
}

void Task2 (void *pvParameters ) // This is a Task 2.
{
    // prints Tache 2 to the serial monitor
    volatile uint32_t cnt ;
    for (;;) // A Task shall never return or exit.
    {
        Serial.println(" Tache 2 en execution");
        for(cnt=0;cnt<=1200000;cnt++);
    }
}

```

La tâche de plus haute priorité est la seule à entrer dans l'état **Running** et n'en sort pas car elle n'attend rien pour s'exécuter. L'autre tâche ne s'exécute jamais, on parle de '**Famine**' (*Starving*). La figure suivante montre la séquence d'exécution.



The execution pattern when one task has a higher priority than the other

ANNEXE : États possibles d'une tâche sous FreeRTOS

Le cycle de vie d'une tâche

A tout moment de l'exécution d'une application sous FreeRTOS, chacune des tâches créées dans le système possède un *état*. La liste des états possibles est la suivante :

- **En cours d'exécution** (*Running*) : La tâche est exécutée en ce moment et elle utilise le processeur ;
- **Prête** (*Ready*) : la tâche est prête à être exécutée (elle n'est ni suspendue, ni bloquée), mais est en attente car une tâche d'une priorité égale ou supérieure utilise actuellement le processeur ;
- **Bloquée** (*Blocked*) : Une tâche peut être bloquée pour plusieurs raisons : Attente d'un événement temporel ou externe, d'un événement sur une file de message ou un sémaphore. Les tâches bloquées ont toutes un délai au delà duquel elle sont débloquées. Les tâches bloquées ne sont pas examinées par l'ordonnanceur ;
- **Suspendue** (*Suspended*) : Tout comme les tâches bloquées, les tâches suspendues ne sont pas examinées par l'ordonnanceur. la différence principale entre une tâche suspendue et une tâche bloquée est que la tâche suspendue peut l'être indéfiniment, contrairement à la tâche bloquée qui l'est jusqu'à l'expiration du délai.

On peut modéliser le cycle de vie d'une tâche sous FreeRTOS par l'automate sur la figure 4.1 page 34.

La pile d'une tâche

Chaque tâche créée dans le système possède une pile : c'est un espace continu en mémoire RAM, utilisé pour stocker les variables locales à la tâche, ainsi que pour sauvegarder le contexte de la tâche lors de sa suspension.

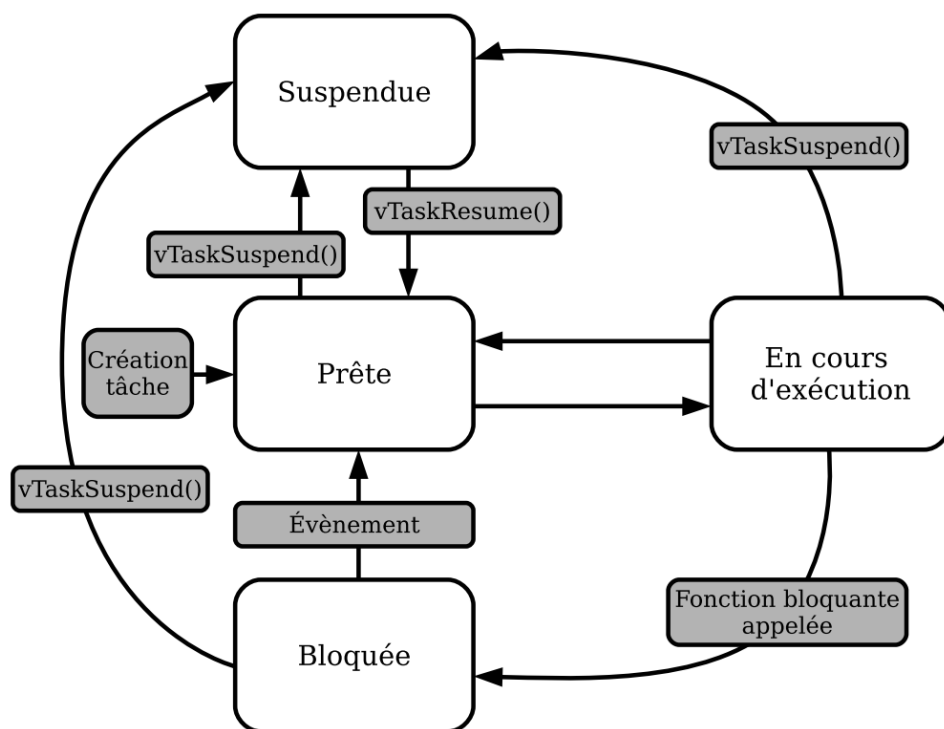


FIGURE 4.1 – Le cycle de vie d'une tâche sous FreeRTOS.