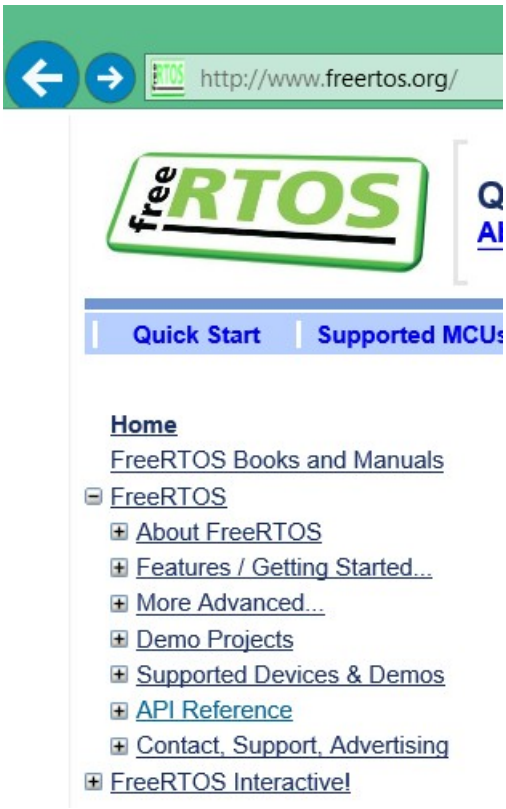
# FREERTOS COURS ET EXERCICE SUR STM32 AVEC SW4STM32 ET STM32CUBE

**Objectifs : Être capable de comprendre l'utilité d'un OS temps réel**
**Être capable de mettre en œuvre FreeRTOS sur STM32**

---

Référence de l'API : www.freertos.org menu de gauche




## 1. PROGRAMMATION TEMPS RÉEL AVEC FREERTOS

### 1.1. Qu'est ce que le temps réel

Un système temps réel est un système qui :
Lit toutes les données entrantes suffisamment vite pour que les valeurs lues soient encore significative
Réagit à ces données suffisamment vite.

Dans un système temps réel les actions se réalisent à un temps déterminée quoiqu'il se produise dans l'environnement (inteRruption IT, attente de donnée…. : n'influence pas le bon déroulement temporel du système) : on parle de système déterministe

Il existe 2 types de système temps réel : mou ou dur
RT soft (mou) : le système accepte que parfois les contraintes temporelles ne soient pas respectées.

RT hard (dur) : le système exige un strict management du temps et des ressources (CPU, PILE, (stack), TAS (heap)

Attention : en fonction du système les contraintes temporelles sont diverses et variées : régulation de température (minute), régulation de process chimique (minute heure voir jours) : un RT n'est donc pas un système obligatoirement rapide !

## 1.2. Définitions

***Programme :***
description d'un algorithme résolvant un problème. Plusieurs langages possibles (C, C++, java...)
Le programme est statique.

***Processus (process) :***
entité chargée en mémoire
Contient du code , des données et des ressources systèmes (registre CPU, ram, rom…)
Executé dans son propre espaces d'adressage privé,
Composé d'une ou plusieurs tâches.

***Tâche (task or thread) :***
entité dynamique résultant d'un partitionnement (découpage) du processus,
séquence de code dédiée à une tâche donnée,
possède sa propre priorité et pile (stack).

***Processeur (processor uP) :***
entité matérielle qui exécute une séquence.

***Préemption :***

c'est un des outils permettant de rendre un système déterministe

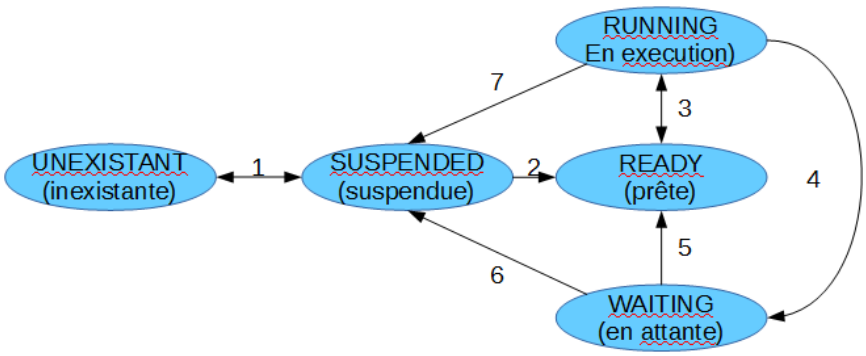
> *Ex : Deux taches tournent sur le système si la tâche 2 veut prendre la main car elle est prioritaire on dit qu'elle préempte la tâche 1.*

***Ressources :***

Ce sont les objets utilisés par une tâche ou un processus.

Ressources matériels (hardware) : processeur, mémoire, périphériques (device)
Ressources logicielles : programme, données, fichiers, message

## 1.3. Les différents états d'une tâche



## 1.4. Composition d'une tâche

Une tâche utilisée généralement 3 zones :

La zone de code : instruction, constante
La zone de donnée : on peut y écrire ou lire , elle est partagée avec les différentes tâches du même processus.
La zone PILE : contient les information temporaire : variables locales, contexte de sous programme (compteur porgramme(contient l'adresse de retour), registres …)

Les informations sur la tâche sont contenues dans le TCB : task control block
Il contient : l'identifiant, la priorité, les droits, l'état et la taille de la pile allouée (>128octets)
Le temps allouée à une tâche est le SYSTIC timer.

## 1.5. Passage d'une tâche à une autre : (Task context switch) fig 2.18

Ce context switch fournit :

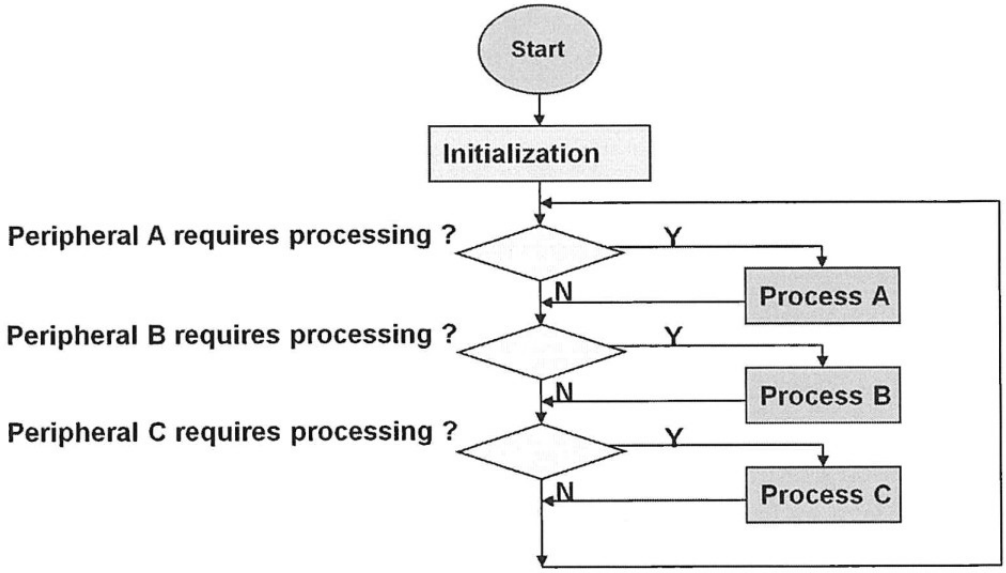
- Une sauvegarde des registres internes
- Le choix de la tâche suivante à exécuter
- Le contexte de la nouvelle tâche

Les sauvegardes des contextes se font par des PUSH (pousser dans la PILE) ou des PULL (lire de la PILE)

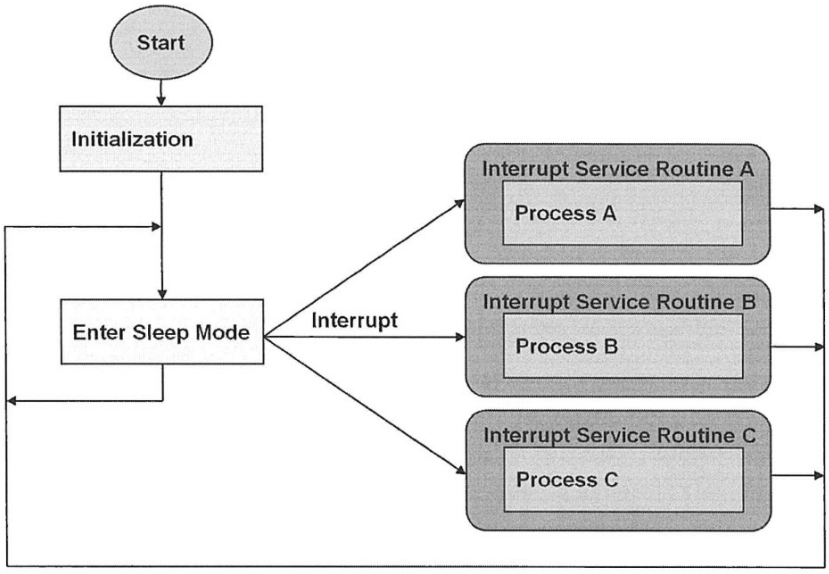
# 2. GESTION DU TEMPS RÉEL D'UN PROGRAMME.

La gestion des tâches se fait grâce à un ordonnanceur (SCHEDULER) qui va définir l'ordre des tâches avec leur priorité…
Plusieurs méthodes de gestion des tâches existent et de nombreux cas peuvent se produire.

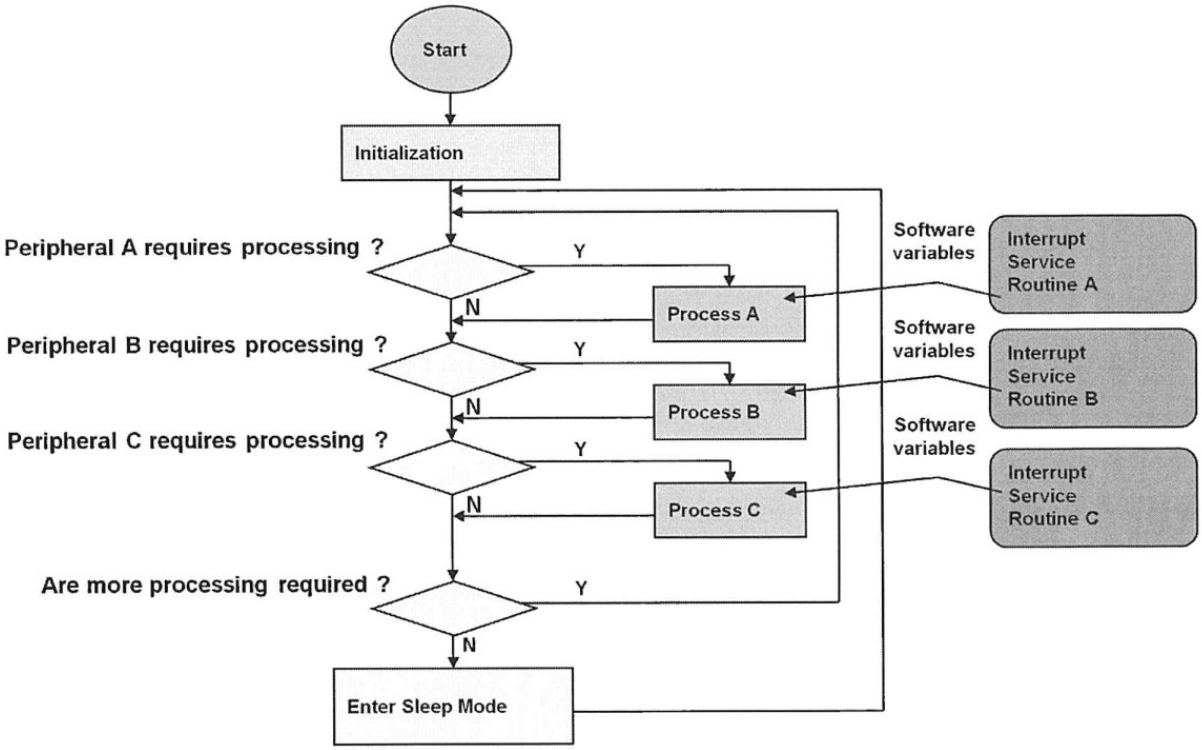
## 2.1. La scrutation (polling) :




## 2.2. Les interruptions (interrupt) :



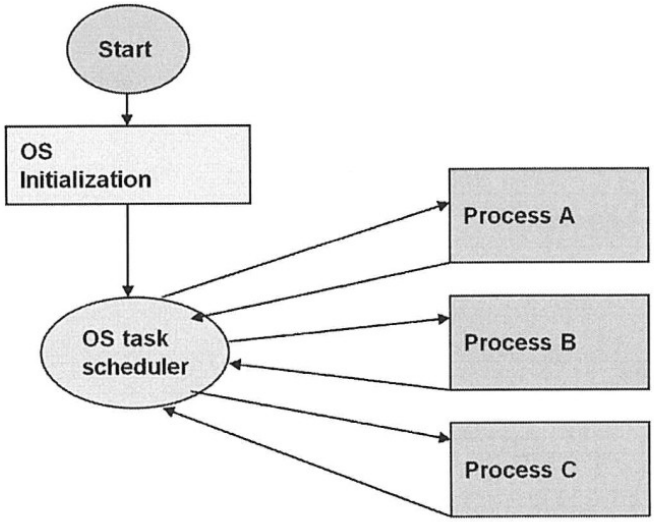
## 2.3. Scrutation + interuption fig 2.4




L'osRT (operating system real time) permet de gérer les tâches avec les priorités par différentes méthodes.

## 2.4. L'ordonnancement

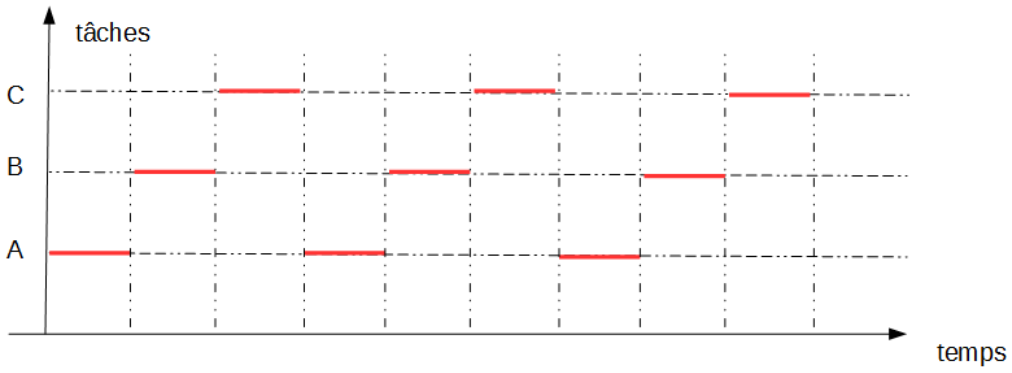
Dans ce type de gestion on utlise un ordonnanceur (SCHEDULER)
Il existe plusieurs stratégie pour ordonnancer les tâches




### *Le TIME SHARING (Round robin) : partage du temps*

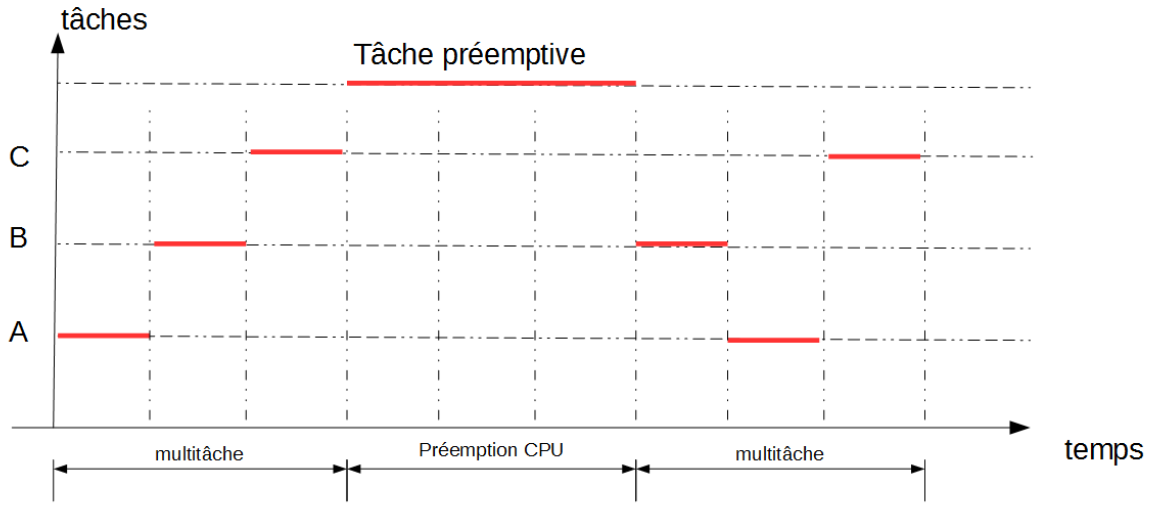
L'ordonnanceur alloue le temps CPU à une tâche durant un temps lmité : TIME SLICE
A chaque CLOCK TICK (période?) l'ordonnanceur alloue le CPU à la tâche suivante.
On a alors un effet 'round robin' (tourniquet).

Certains osRT allouent un nombre spécifique de TICK pour chaque tache avant de laisser le processeur executer une autre tâche (ayant la même priorité : ce n'est pas le cas de FreeRTOS.

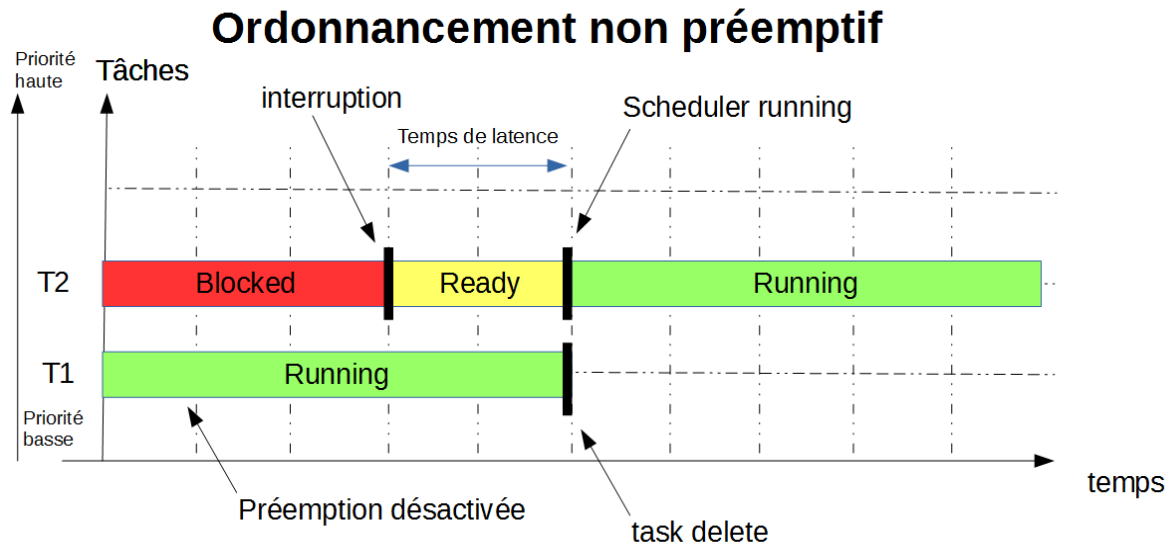


### *Ordonnancement sur priorités*

Le temps CPU est alloué en fonction de la priorité de la tâche.
L'ordonnanceur vérifie en permanence que la tâche en cours d'execution est la plus prioritaire.
Si plusieurs tâches ont la même priorité on applique un ROUND ROBIN.

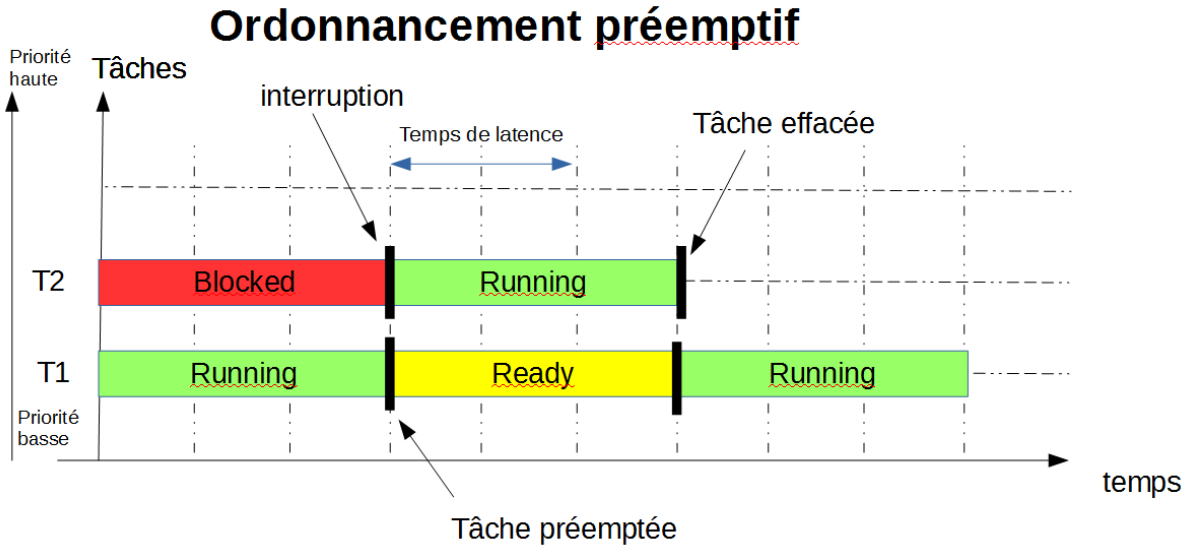



### *Exemple d'ordonnancement non préemptif*




Un temps de latence apparaît avant que T2 puisse fonctioné : mauvais car priorité T2 > priorité T1.

***Exemple d'ordonnancement préemptif***




La tâche T2 est prioritaire sur T1 elle passe en mode 'Running' , la tâche T1 passe de 'running' à 'ready' en attendant la fin de T2 pour reprendre son fonctionnement.

# 3. FREERTOS

## 3.1. Avantage d'un osRT

Autorise le multi tâche
Ordonnancement avec priorités (la priorité la + haute gagne)
Communication intertâche (queue)
Synchronisation des accès aux ressources (sémaphores et mutex)
Prédictibilité
Gestion des interruptions

## 3.2. Avantage FreeRTOS

Gratuit (dépend des versions)
Faible impact : consomme peu de ROM, RAM et ressources processeur

## 3.3. Installation

### *Utilisation de SW4STM32*

Lors du choix du processeur cochez la case use FREERTOS

### *Utilisation sur un autre microcontroleur*

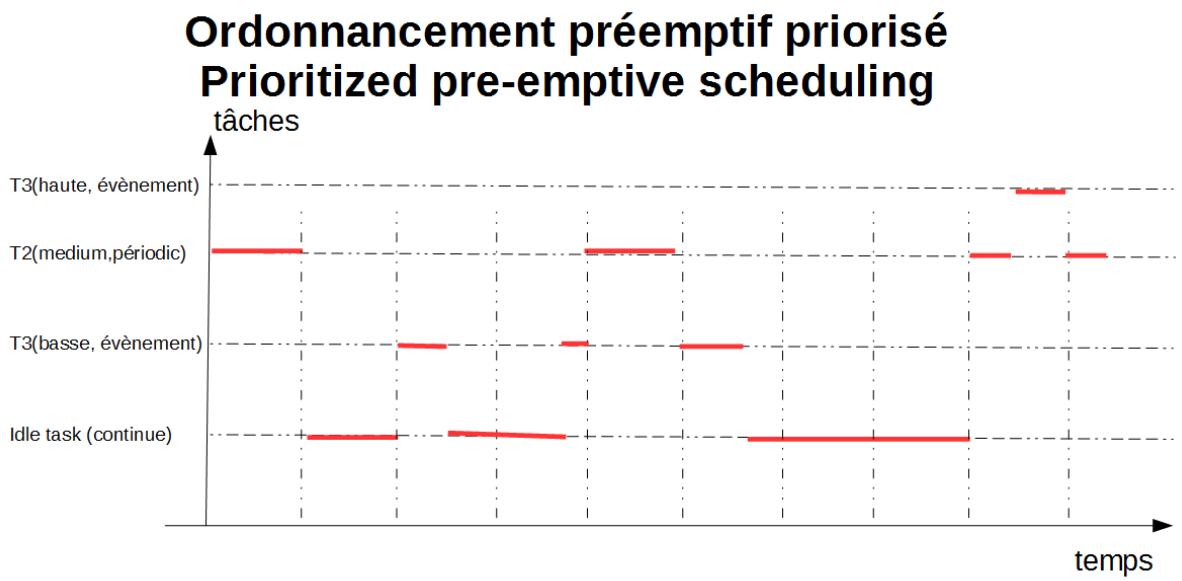
Placer les fichiers nécessaire .c et .h dans votre projet

## 3.4. Ordonnancement préemptif priorisé.

Chaque tâche possède une priorité
Chaque tâche peut exister dans plusieurs états
Une seule tâche fonctionne (running) à la fois
La tâche la plus prioritaire est en mode 'running'
Les tâches peuvent attendre en mode 'bloqué' un événement. :
Les tâches repassent automatiquement à 'ready' si :

Un événement temporel arrive régulièrement : tâche périodique ou comportement d'un time out

Un événement de synchronisation arrive : envoyé par une queue ou un sémaphore (signal asynchrone d'interruption).




IDLE : cette tâche est toujours présente elle représente la tâche d'attente de l'ordonnanceur.
T3 (priorité basse, déclenchée sur un événement ) : la tâche est interrompue par les tâches plus prioritaire.
T2 (priorité moyenne, tâche périodique) : la tâche se déroule régulièrement (ex : TIMER) : elle est interrompue par T1 plus prioritaire.

Si tâche de même priorité : Round robin avec un context switch (changement de contexte pour sauver les données nécessaire au retour dans la tâche dès que possible).

### *Options de configuration de FreeRTOS*

configUSE_PREEMPTION = 0 l'ordonnanceur n'est pas préemptif :
Si une tâche T1 de faible priorité devance une tâche T2 de priorité plus haute , la tâche T1 continue quand même à  s'executer.

configUSE_TIME_SLICING  = 1 : le round robin est activé en cas de priorité indentique.
L'ordonnanceur alterne entre les tâches à chaque intervalle de temps TICK interrupt.

# 4. LA FONCTION TASK (TÂCHE)

## 4.1. Définition et propriétés

Une tâche est un petit programme qui possède c'est propre droit.
Une tâche est un fonction C avec un point d'entrée qui tourne dans une boucle infinie.
On ne peut pas en sortir : on ne met pas de RETURN
On peut l'effacer si elle n'est plus utile.

Une tâche est une instance indépendante composée :

De sa propre pile
De sa propre copie de toutes les variables locales à la tâche

Une seule définition de fonction tâche peut être utilisée pour créer toutes les tâches.
Un prototype de tâche doit :
Avoir un paramètre void pointeur
Retourner un void

## 4.2. Exemple

```
Void DummyTaskFunction (void *pvParameters)
{
/*
Les variables sont délcarées comme dans une fonction classique
Chaque instance de tâche créée utilisant la fonction aura sa propre copie de la variable i (sauf si déclarée en static)
*/

int i=0 ;

// une tâche est normalement implantée dans une boucle infinie
        For(;;) //ou while(1)
        {
                /*code de la tâche ici
                i=i+3 ; //exemple
        }
}
/*au cas ou on sort de la boucle infinie il faut delete la tâche*/
vTaskDelete(NULL) ;
/*NULL est un paramètre que fait référence à la tâche elle même.
}//fin fonction DummyTaskFunction
```

Les fonctions suivantes sont définies dans l'os : FreeRTOS :

# 5. NOTATIONS FREERTOS

## 1.1. Types de base

**portCHAR** : char (entier sur 8bits)

**portSHORT** : short (entier sur 16bits)

**portLONG** : long (entier sur 64bits)

**portBASE_TYPE :** int (entier sur 32bits)

## 1.2. Notations des fonctions et des variables

Par convention, dans un programme FreeRTOS, on choisit des noms de variables avec un préfixe comportant l'indication du type de la variable.
On choisit des noms de fonctions avec 2 préfixes : le premier indique le type de la variable retournée et le second indique le fichier dans lequel la fonction est définie.

**Prefixe de type :**

**c** : char

**s** : short

**l** : long

**x** : portBASE_TYPE

**p** : pointeur

**v** : void (fonction ne retournant aucun paramètre)

**u** : unsigned

**uc** : unsigned char

***pc*** *:* pointeur sur un char ...

Exemples :

**x**Semaphore : variable avec le type de base "portBASE_TYPE"

***vTask***PrioritySet() retourne un ***void*** et est définie dans ***task.c***

***xQueue***Receive() retourne un ***portBASE_TYPE*** et est définie dans ***Queue.c***

***vSemaphore***CreateBinary() retourne un ***void*** et est définie dans ***semphr.h***

### 5.1. Nom des fonctions :

Préfixe précédent pour la variable de retour
Exemple : vTaskPrioritySet () retourne un void.

***Fonctions utiles***

xTaskCreate(…..): voir la doc pour les paramètres

L'appel de l'ordonnanceur se fait par :
vTaskStartScheduler() ;

Création d'un délai à l'aide de la fonction :
vTaskDelay (…) ;

# 6. CRÉATION DE TÂCHE

Lors de la création d'une tâche, le noyau alloue un espace pour la TCB et la pile dans le Tas.

La fonction de création de tâche est xTaskCreate(), elle utilise 6 paramètres et retourne une valeur de type ***"portBASE_TYPE"*** :

```
void Task1( void *pvParameters );
const char *pvParametersTask1 = "Passage d'un pointeur sur une chaîne de caractères !";

/**
* @fn main
* @brief main entry point
*/
int main(void){

  // Création d'une  tâche
  xTaskCreate( Task1,                  // Pointeur sur la fonction implémentant la tâche
      "Tache 1",                       // Chaîne de caractères associée à la tâche (debug).
      configMINIMAL_STACK_SIZE,        // Taille de la pile associée à la tâche
      pvParametersTask1,               // Paramètre passé à la tâche
      tskIDLE_PRIORITY + 1,            // Priorité associée à la tâche
      NULL);                           // "handle" pour la gestion de la tâche
  ...
}
```

## 6.1. Paramètres de la fonction

**Task1 :** pointeur sur la fonction implémentant la tâche : la fonction en langage C

**"Tache 1" :** chaîne de caractères associée à la tâche (sauvée dans la TCB). Utilisé par des outils de trace ou pour de l'instrumentation de code (par exemple, stack overflow).

**configMINIMAL_STACK_SIZE :** taille de la pile associée à la tâche. Cette macro est définie dans le fichier **FreeRTOSConfig.h**. Il s'agit par défaut de la taille de la pile pour la tâche Idle.

Attention, cette taille est donnée en "Words" et non en octets. La taille d'un Word dépend de l'architecture matérielle utilisée.

Dans notre cas un Word = 32bits (les PIC32MX sont des MCU's 32bits).

**pvParametersTask1 :** pointeur permettant de passer un paramètre à la tâche NULL s'il n'y en a pas.

**tskIDLE_PRIORITY + 1** : priorité de la tâche.

tskIDLE_PRIORITY ou "0" est la priorité minimale qui correspond à la priorité de la tâche Idle.

La priorité maximale vaut **configMAX_PRIORITIES** définie dans **FreeRTOSConfig.h.**

**NULL :** pointeur de préemption sur les tâches

Lorsque ce pointeur vaut "NULL" il pointe sur la tâche elle-même en cours de création. A manipuler avec précaution.

### 6.2. Valeur retournée

L'instruction xTaskCreate() retourne une valeur de type **portBASE_TYPE** qui peut prendre 2 valeurs possibles :

- **pdTRUE** : tout c'est bien passé

**errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY** : la tâche n'est pas créée car il manque de la place en mémoire

### 6.3. Création de plusieurs instances d'une tache

Si on crée 2 instances de la même tâche on crée 2 zones TCB + pile
Chaque instance est indépendante : elles sont gérées indépendamment et ont leurs propres variables : ce sont des variables locales, sauf si on les déclare « static » auquel cas la variable est partagée entre les différentes instances de la tâche
Exemple de création de 2 tâches totalement indépendantes utilisant le même code source Task1 :

```
xTaskCreate(Task1,"instance1DeTask1",1024,NULL,2,NULL);
xTaskCreate(Task1,"instance2DeTask1",1024,NULL,2,NULL);
```

## 7. QUEUE DE MESSAGES

### 7.1. Création d'une queue de messages

Une queue de messages (messages queue ou queue) ou boîte aux lettres (mailbox) est un outil logiciel asynchrone permettant des communications voir des synchronisations entre tâches.

Une queue comporte deux files d'attentes : une stockant l'ordre d'arrivée des messages et une seconde stockant l'ordre d'arrivée des tâches.
Une queue de messages contient un nombre fini d'éléments dont la taille est configurable.
Elle est régie par le principe de fonctionnement d'une FIFO (First In First Out). Le premier entré dans la file sera le premier à en sortir.
FreeRTOS propose si nécessaire des alternatives à ce fonctionnement.

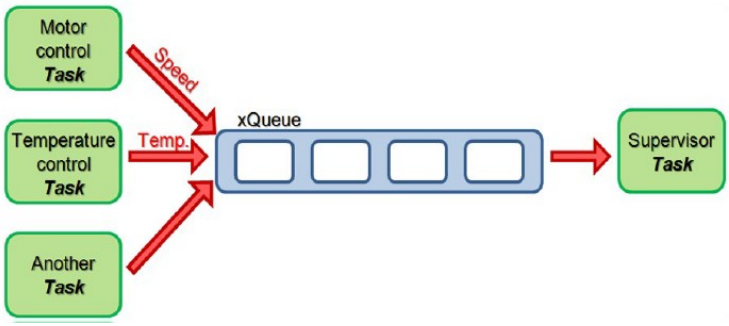
FreeRTOS crée les queues de messages et alloue les ressources mémoire nécessaires dans le Tas.

Une queue de messages peut avoir plusieurs écrivains et plusieurs lecteurs.

Cependant, de façon générale, elles sont principalement utilisées avec des écrivains multiples et un seul lecteur.
Les écrivains sont les tâches pouvant écrire dans une queue de messages.
Les lecteurs sont donc celles susceptibles de la lire.

Un élément lu est retiré de la file d'attente.

Si un lecteur cherche à lire une queue de messages vide, il se fait bloquer jusqu'à l'arrivée d'un nouveau message.
Pour des tâches de même priorité, ce sera la première bloquée qui sera la première réveillée (FIFO).
FreeRTOS étant un exécutif temps réel, si une tâche de plus haute priorité se fait bloquer, elle passe en tête de file.
Exemple de scénario :




## 7.2. API de FreeRTOS

Les deux principales fonctions proposées par FreeRTOS sont :

- **xQueueSend() ou xQueueSendToBack() :** envoi d'une donnée vers une queue de messages à la suite des éléments déjà présents.

  Cette fonction n'est bloquante que si la queue de messages est pleine.

- **XQueueReceive() :** Récupère la donnée en tête d'une queue de messages.

  L'élément lu est retiré de la file d'attente.

  Le second élément passe alors en tête de file.

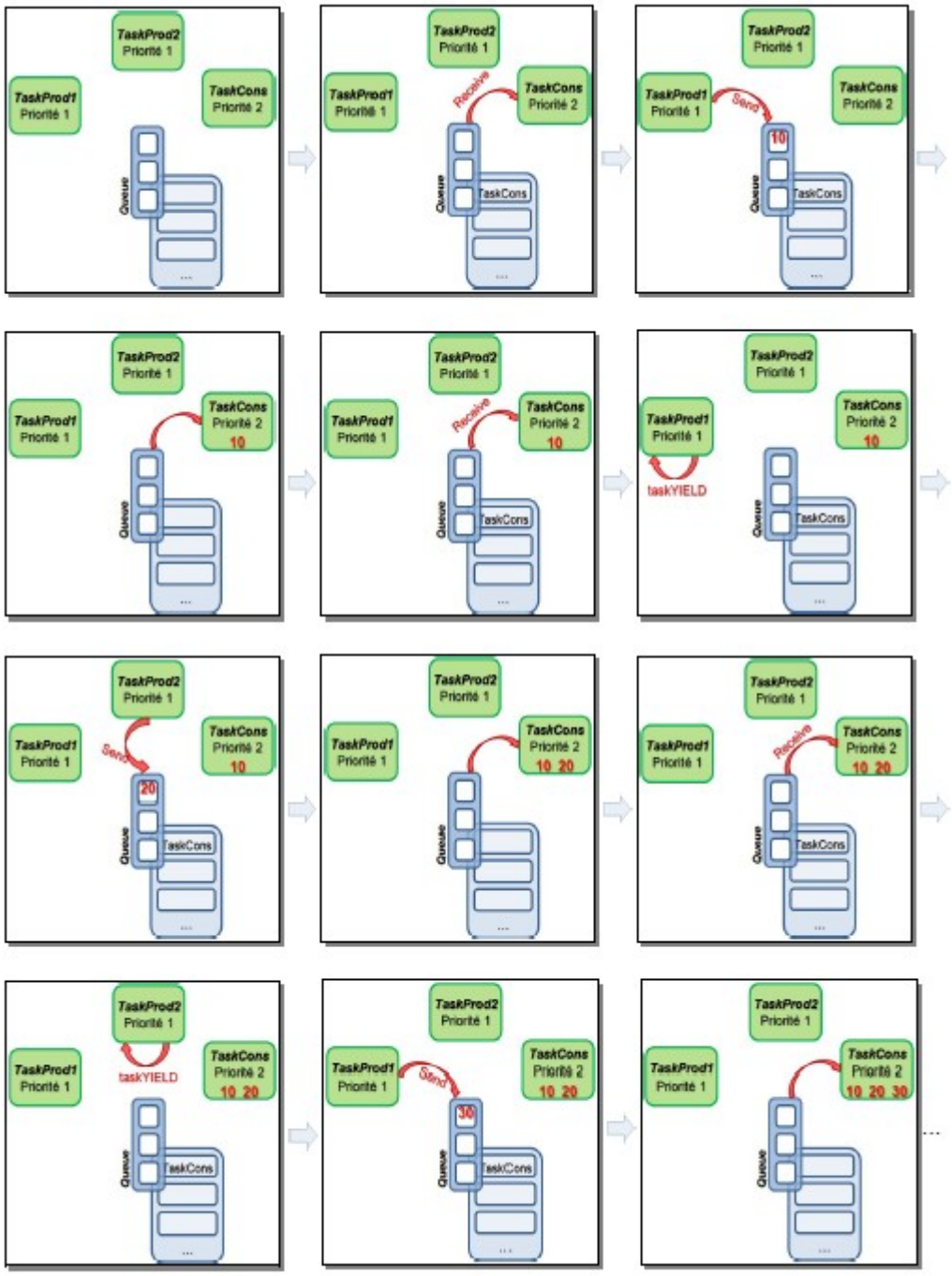
  Cette fonction n'est bloquante que si la queue de messages est vide.

Fonctions complémentaires :

- **xQueueSendToFront()** écrit une donnée en tête de file d'attente.

  Les données sont alors prioritaires et placées en tête de la queue de message

Dans l'exemple ci-dessous le noyau travaille en mode coopératif.
Ce scénario présente une application avec deux écrivains et un lecteur :




## 7.3. Timeout

Certaines fonctions pour la gestion de queue de messages et de sémaphores utilisent un Timeout ou temps mort.

Lorsqu'une tâche est bloquée après l'appel de l'une de ces fonctions, celle-ci se réveillera (passage à l'état prêt) automatiquement après un laps de temps nommé Timeout, même si l'événement attendu n'est pas arrivé.

L'utilisation du Timeout permet de garantir la robustesse d'un code en forçant par exemple le réveil d'une tâche en attente d'un événement devant être envoyé par une tâche boguée.

Prenons l'exemple de l'API suivante :

```
portBASE_TYPE xQueueReceive(
        xQueueHandle xQueue,
        void *pvBuffer,
        portTickType xTicksToWait
);
```

Le paramètre xTicksToWait permet de fixer le Timeout. Sous FreeRTOS, le Timeout est donné en ticks.

Si nous ne souhaitons pas utiliser cette fonctionnalité (Timeout infini), il suffit de passer en paramètre la macro **portMAX_DELAY** (définie dans **portmacro.h**).

# 8. SEMAPHORES

Un Sémaphore est un outil utilisé pour la protection de ressources partagées (variables, périphériques, espaces mémoires ...)

Le principe de fonctionnement des sémaphores est très proche de celui des queues de messages

Sous FreeRTOS il n'y a aucun fichier source propre aux sémaphores.

Les API pour la gestion des sémaphores ne sont que des macros appelant des fonctions propres aux queues de messages (queue.c).

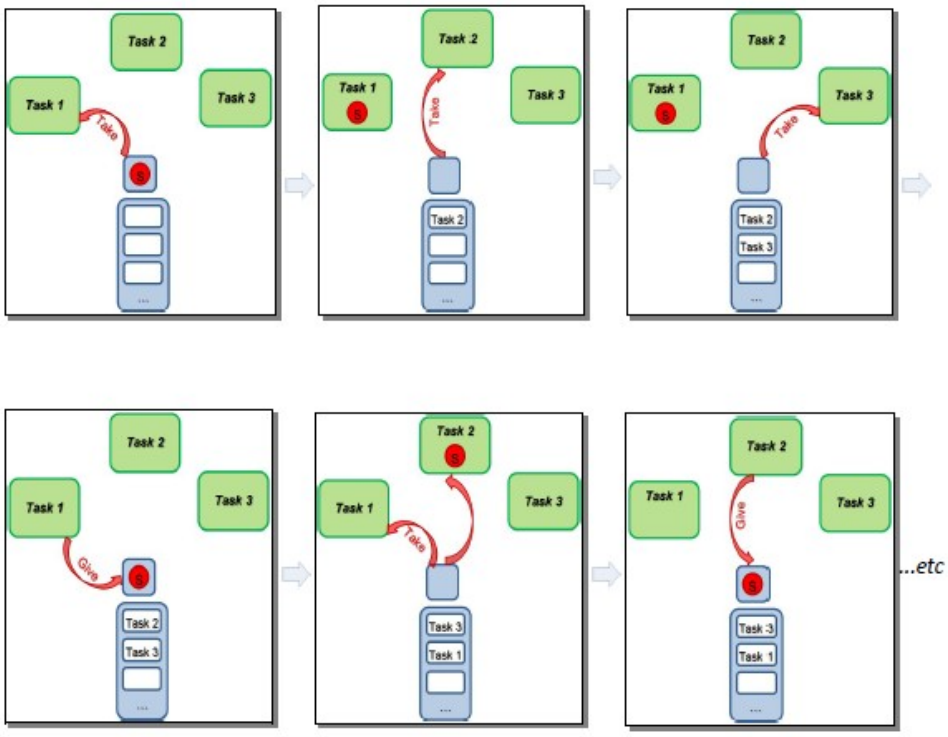
## 8.1. Sémaphore binaire

Un sémaphore binaire peut être vu comme une variable booléenne associée à une file d'attente préservant l'ordre d'arrivée des tâches (cf. queues de messages).

Un sémaphore binaire Pris (Take) par une tâche ne peut plus l'être par une autre, ni même par elle-même tant qu'il n'est pas explicitement Rendu (Give).

Une tâche cherchant à prendre un sémaphore binaire déjà pris se verra bloquée (blocked) jusqu'à ce que la ressource soit relâchée.

Prenons un exemple de scénario :




Le scénario précédent reflète par exemple une application où 3 tâches de même priorité cherchent simultanément à envoyer des données via un UART.

La ressource partagée est alors l'UART qui est protégé par un sémaphore binaire.

Durant ce laps de temps nous nous trouvons dans une section critique qui peut cependant être préemptée par le noyau.

Les différentes tâches utiliseront donc l'UART à tour de rôle (exclusion mutuelle)

## 8.2. MUTEX

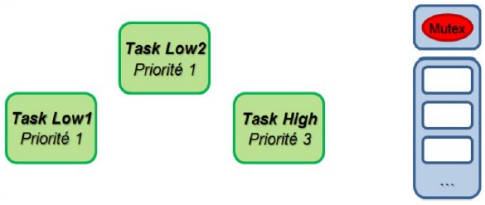
Mutex signifie MUTual EXclusion (ou exclusion mutuelle), il s'agit du concept très rapidement présenté durant le scénario présenté ci-dessus.

Une exclusion mutuelle traduit la notion de protection de ressources partagées.

Sous FreeRTOS, elle peut notamment être réalisée à partir d'un sémaphore binaire (inversion de priorité) ou d'un mutex (héritage de priorité).

Pour FreeRTOS, le principe de fonctionnement de ces deux outils est exactement le même à ceci près que le Mutex effectue un héritage de priorité.

Illustrons le concept d'héritage de priorité :



Héritage de priorité : La tâche Task Low2 vient de prendre une ressource (Mutex) et elle ne la rendra qu'une fois avoir fini ce pourquoi elle l'a pris.

Cependant la tâche Task Low1 est également prête (état ready), l'ordonnanceur applique donc le round-robin et partage le temps CPU entre les deux tâches de même priorité.

Imaginons maintenant que la tâche Task High (de priorité supérieure) cherche également à prendre la ressource (Mutex).

Ceci est impossible, le MUTEX ayant déjà été pris et elle se fait donc bloquer, c'est ce que l'on appelle l'inversion de priorité.

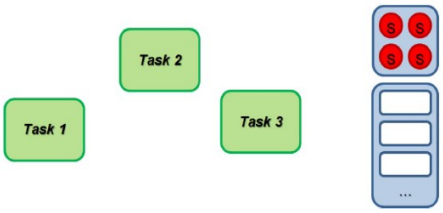
Une tâche de haute priorité se fait bloquer par une tâche de plus basse priorité, le système de priorité choisi par le développeur est inversé.

Cependant avec l'héritage de priorité, la tâche Task Low2 hérite temporairement de la priorité de Task High et pourra donc potentiellement finir de s'exécuter plus rapidement que sans héritage (plus de round-robin avec la tâche de même priorité)

De façon général, dans un système temps réel une tâche ne doit jamais (ou le moins possible !) rester bloquée par une tâche de moindre priorité.

### 8.3. Sémaphore à compteur

Le principe de fonctionnement d'un sémaphore à compteur est identique à celui d'un sémaphore binaire sauf que la ressource protégée peut maintenant être prise (Take) à plusieurs reprises.
Elle peut être prise soit par la même tâche, soit par une nouvelle.
Exemple de sémaphore à 4 jetons :



# 9. CRÉATION D'UN PROJET AVEC CUBE4STM32

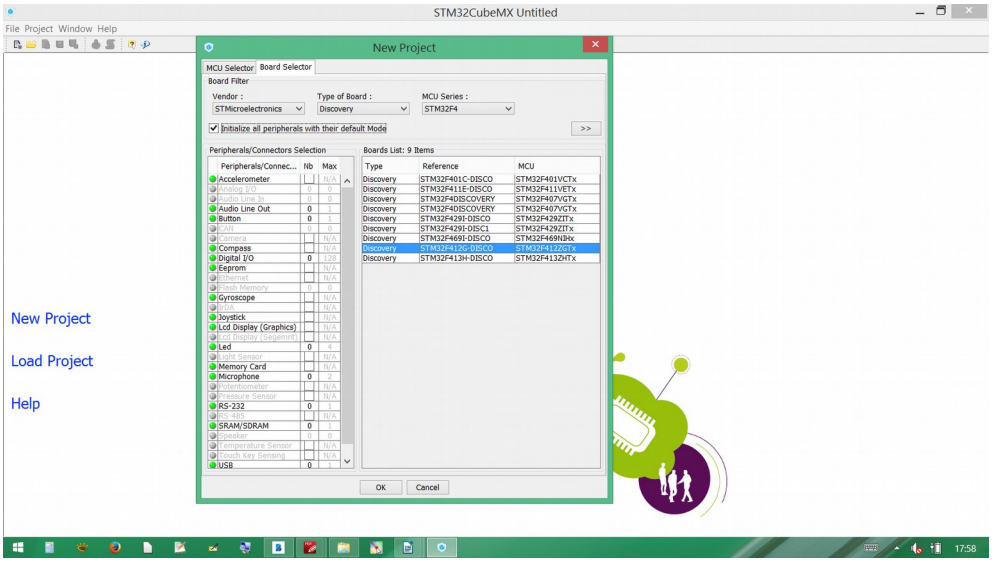
A utilisant Cube il est possible de configurer l'utlisation de FreeRTOS ainsique le nombre de tâches à créer, les sémaphores et mutex éventuels.

Le corps du programme sera crée.

Les fonctions de lancement de l'ordonnanceur, de création de tache et d'appel des tâches seront  générées.

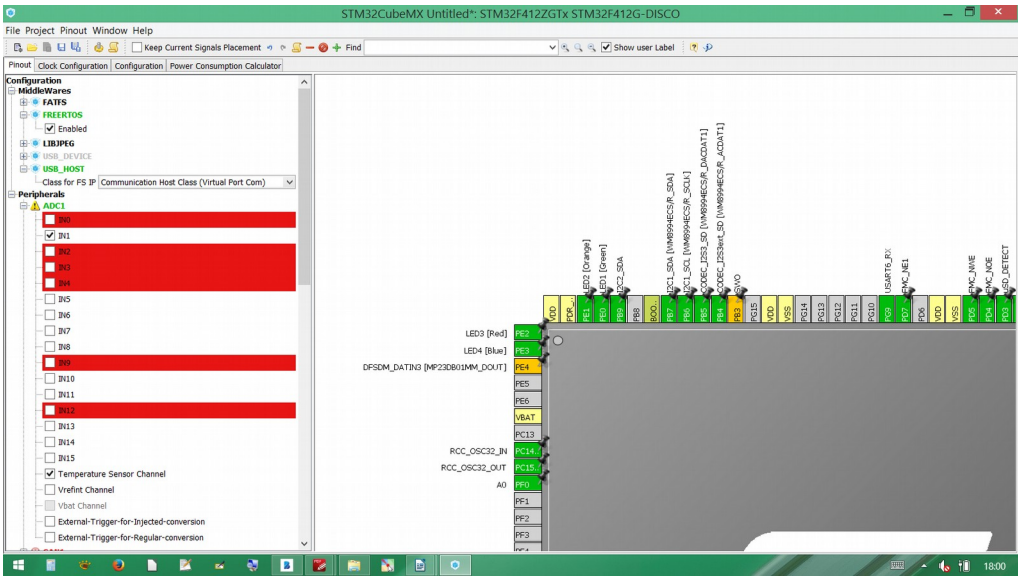
Créer un nouveau projet :

Sélectionner la carte de développement :



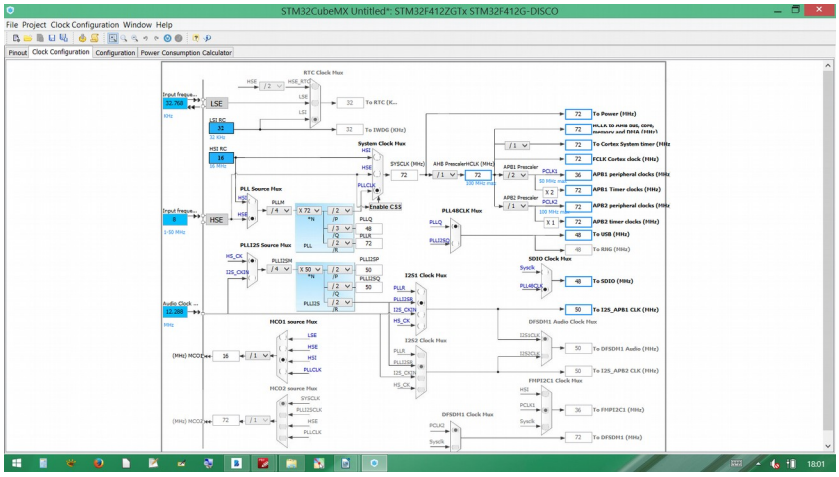

Attention : Activer tous les périphériques

Sélectionner les périphériques nécessaire à votre projet :

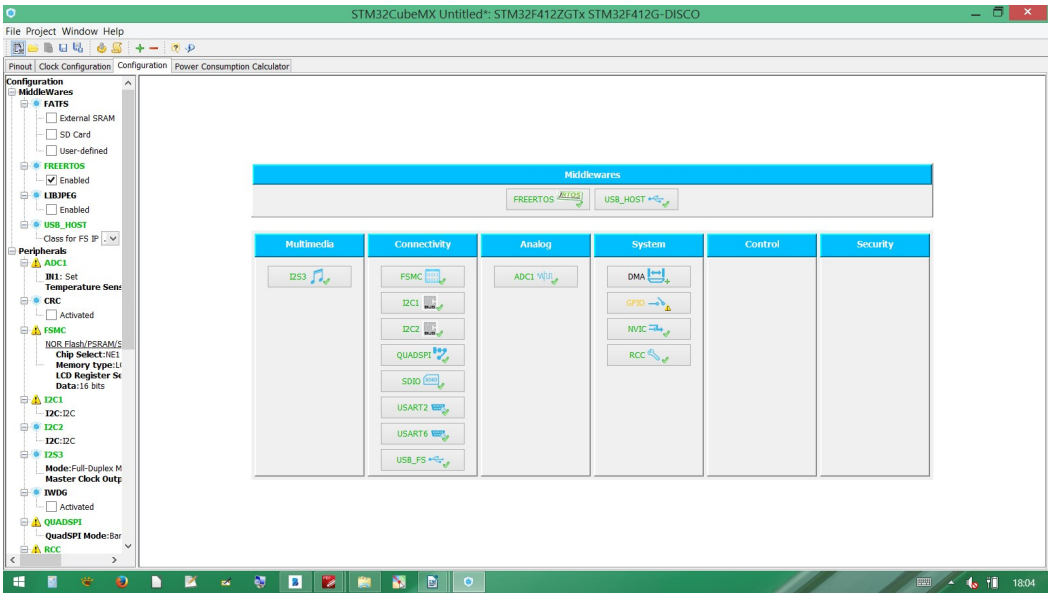



Activer l'utilisation de FreeRTOS.

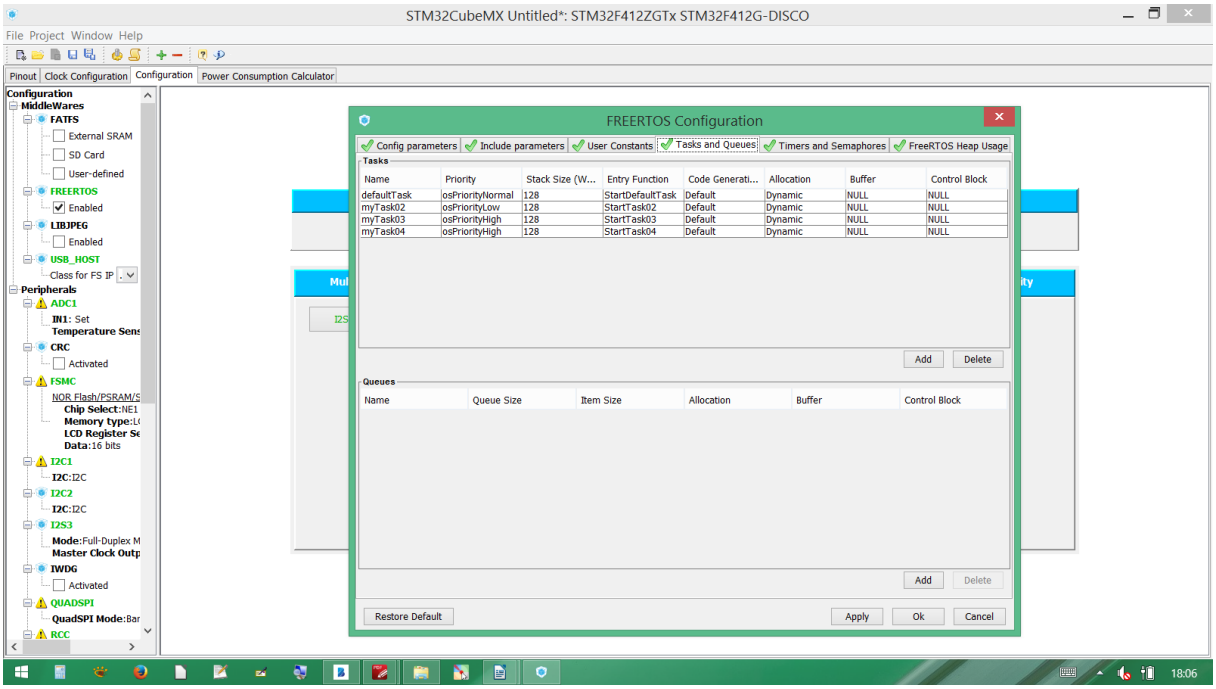
Configurer la clock (72MHz) :



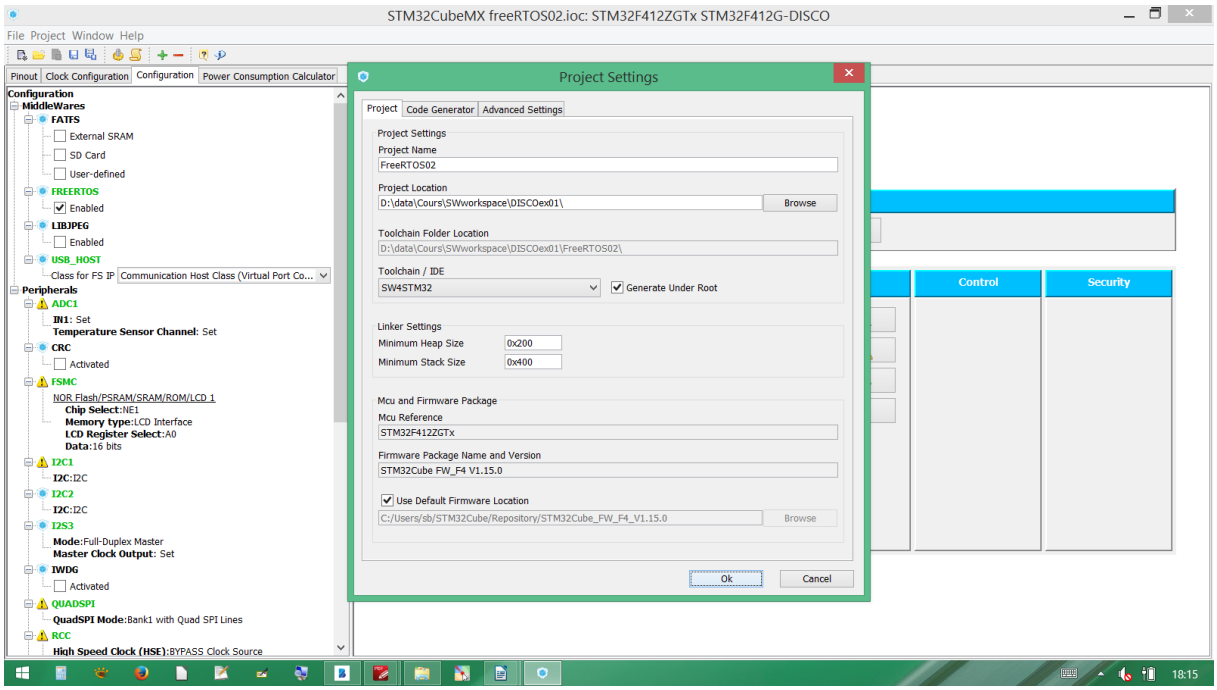

Configurer les périphériques par l'onglet "configuration" :




Choisir FreeRTOS et créer 3 tâches de priorité différentes (ou identiques) :




Générer le code : "project" + "generate code "…

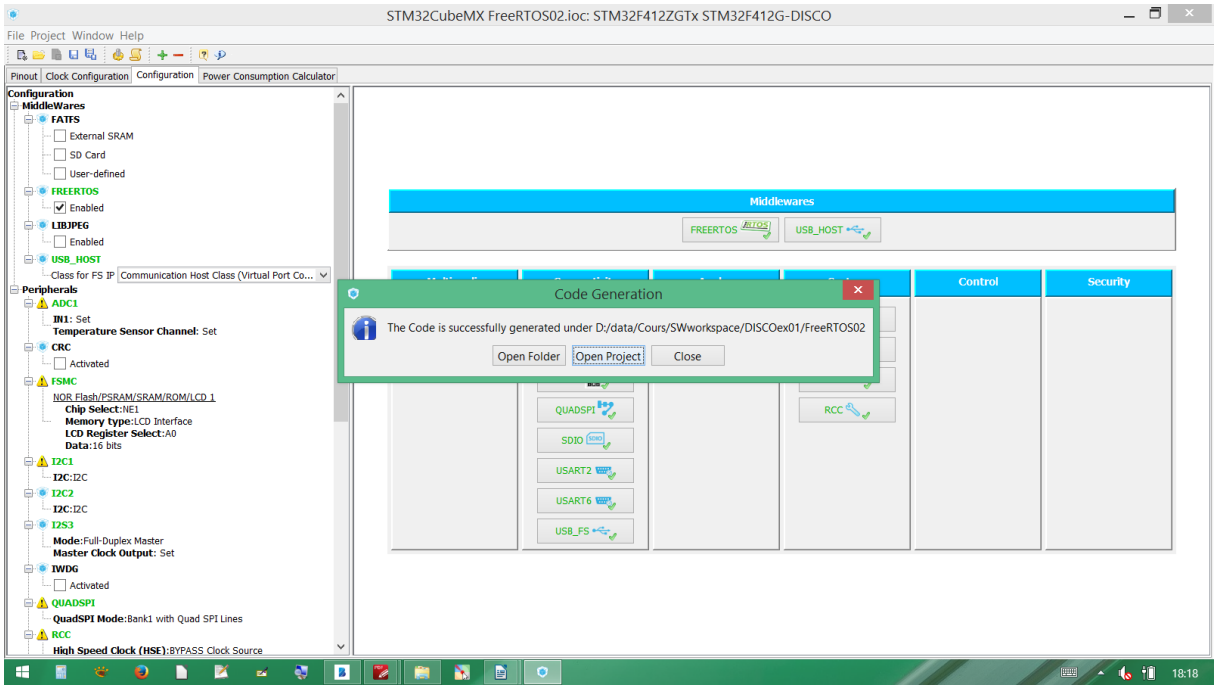



Un dossier est crée contenant  la configuration CubeSTM32 ( .ioc)…

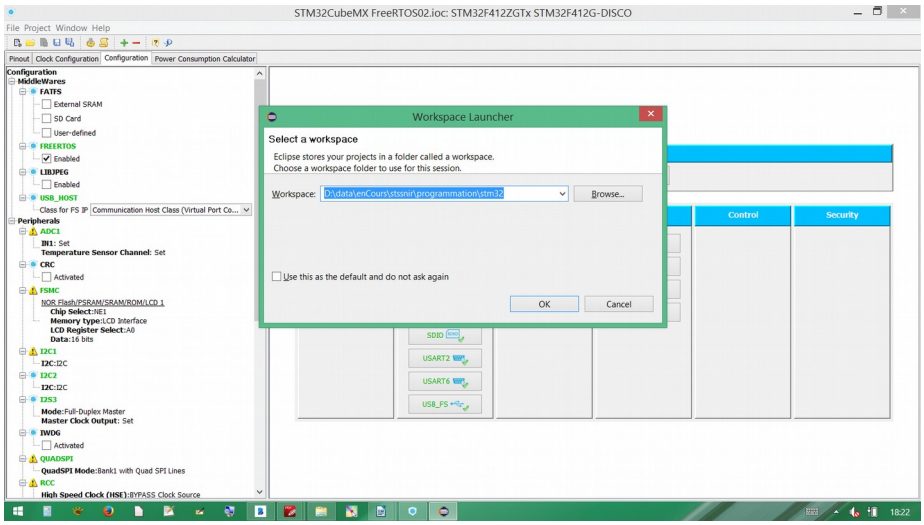
ATTENTION : bien choisir "Toolchain" :SW4STM32

Le projet est généré :

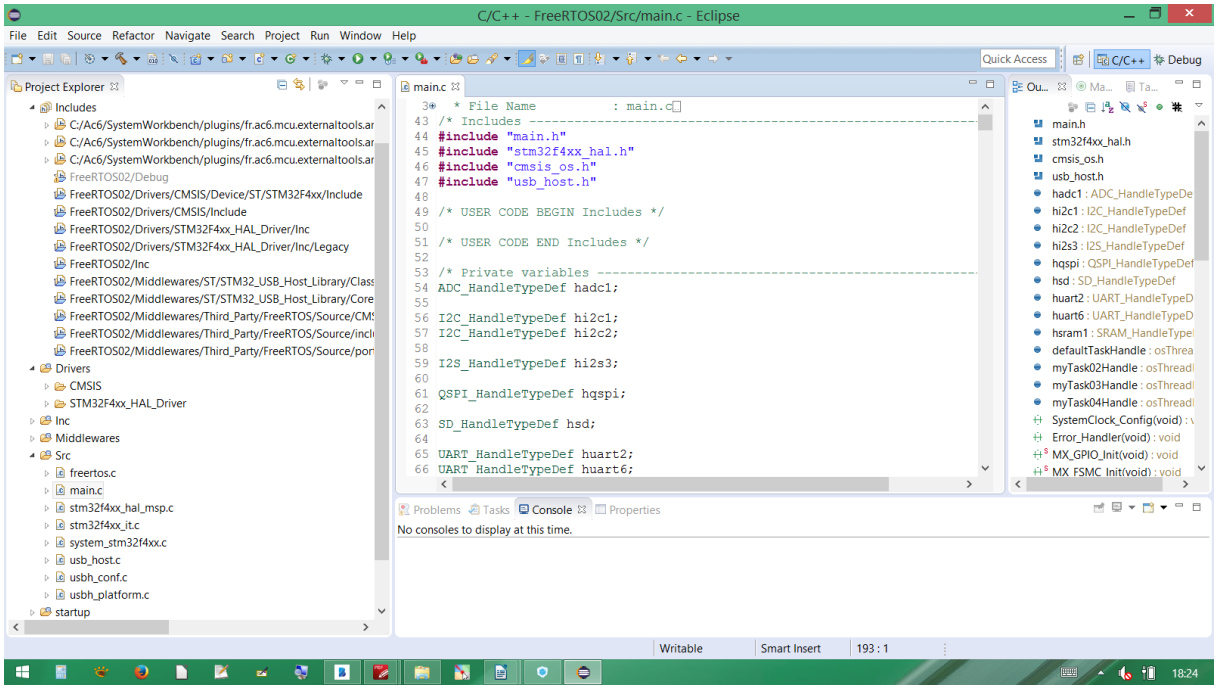
Choisir : Ouvrir projet afin que SW4STM32 se lance…




Choisir un Workspace pour SW4STM32 ….




Le projet est crée ainsi que les fichiers associés.




# 10. EXEMPLE DE PROGRAMME AVEC SW4STM32

On retrouve un code prédéfinit qui permet de développer plus rapidement.
Visiter le code afin de le comprendre.
Retrouver les fonctions associées à FreeRTOS.
Identifier les zones de codage réservées au développeur.

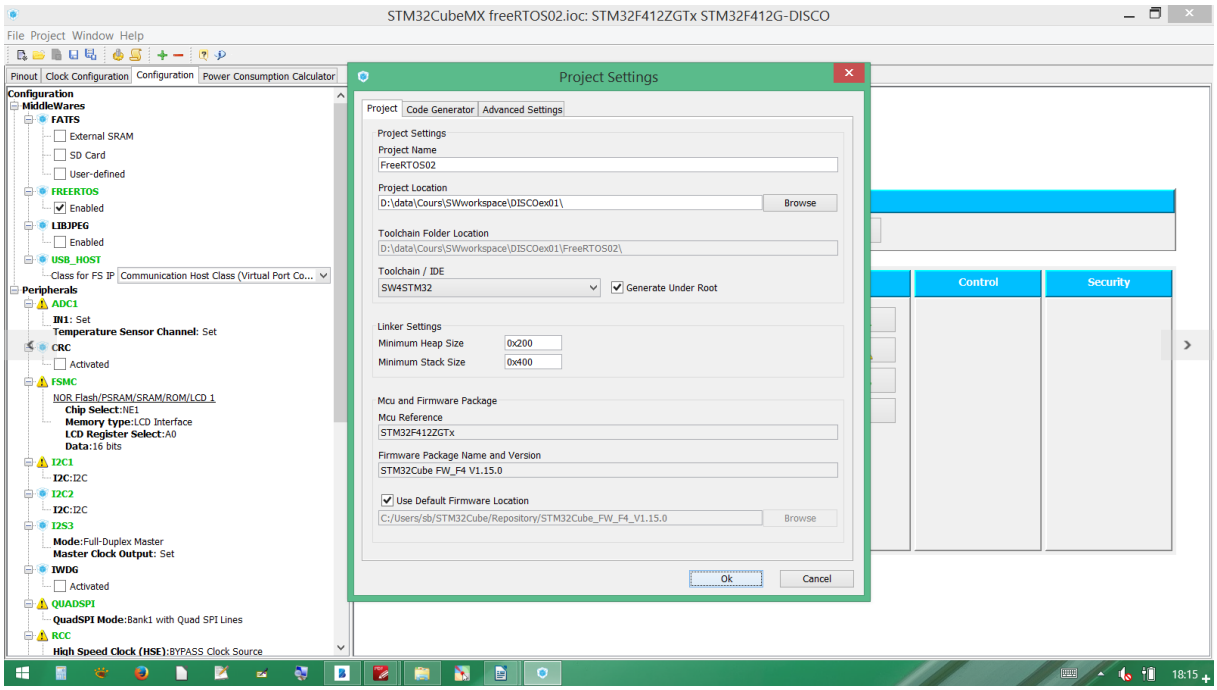
A gauche le menu permet de naviguer dans les fichiers : drivers, includes, src…
En ouvrant le man.c on retrouve la création des tâches…

```
/* Create the thread(s) */

  /* definition and creation of defaultTask */

    osThreadDef(defaultTask,  StartDefaultTask,  osPriorityNormal,  0,
128);

 defaultTaskHandle = osThreadCreate(osThread(defaultTask), NULL);


  /* definition and creation of myTask02 */

  osThreadDef(myTask02, StartTask02, osPriorityLow, 0, 128);

 myTask02Handle = osThreadCreate(osThread(myTask02), NULL);


  /* definition and creation of myTask03 */

  osThreadDef(myTask03, StartTask03, osPriorityHigh, 0, 128);

 myTask03Handle = osThreadCreate(osThread(myTask03), NULL);


  /* definition and creation of myTask04 */

  osThreadDef(myTask04, StartTask04, osPriorityHigh, 0, 128);

 myTask04Handle = osThreadCreate(osThread(myTask04), NULL);


  /* USER CODE BEGIN RTOS_THREADS */

  /* add threads, ... */
```

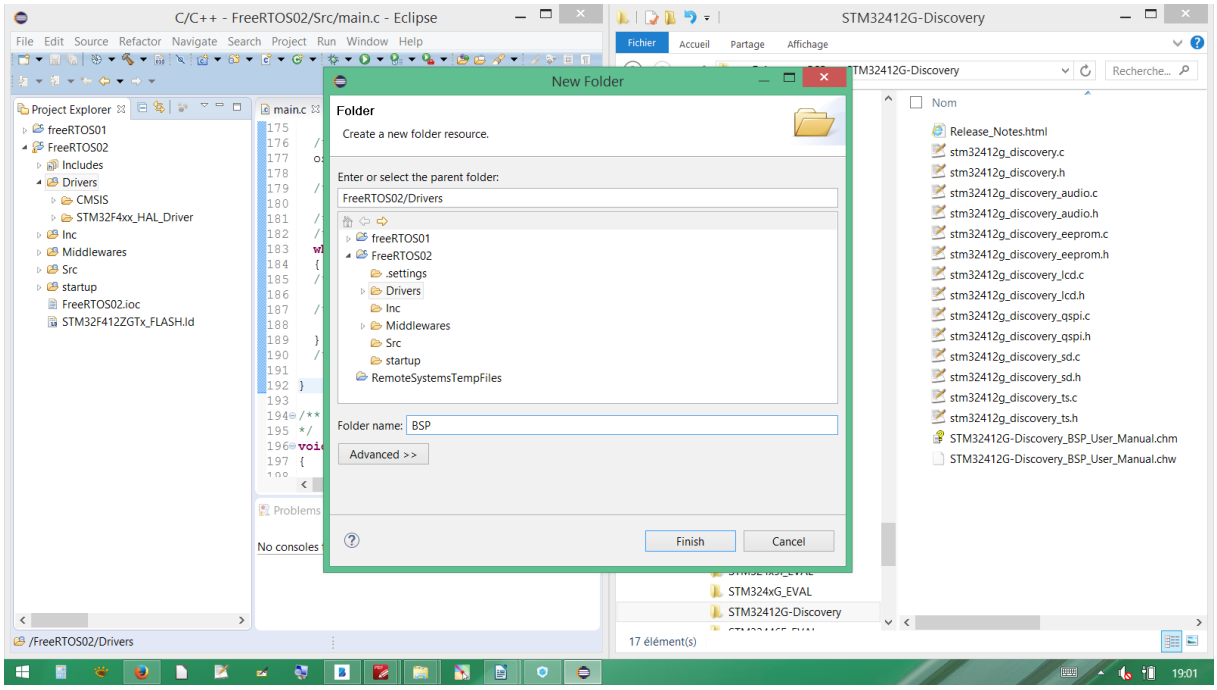
```
  /* USER CODE END RTOS_THREADS */


  /* USER CODE BEGIN RTOS_QUEUES */

  /* add queues, ... */

  /* USER CODE END RTOS_QUEUES */




  /* Start scheduler */

 osKernelStart();


   /* We should never get here as control is now taken by the
scheduler */


  /* Infinite loop */

  /* USER CODE BEGIN WHILE */

  while (1)

 {

  /* USER CODE END WHILE */


  /* USER CODE BEGIN 3 */


 }

  /* USER CODE END 3 */


}
```

Afin d'utiliser la carte DISCOSTM32F412 de manière optimale il faut ajouter les drivers BSP.

Repérer le dossier de STM32CubeMx afin de trouver l'emplacement du driver/BSP.

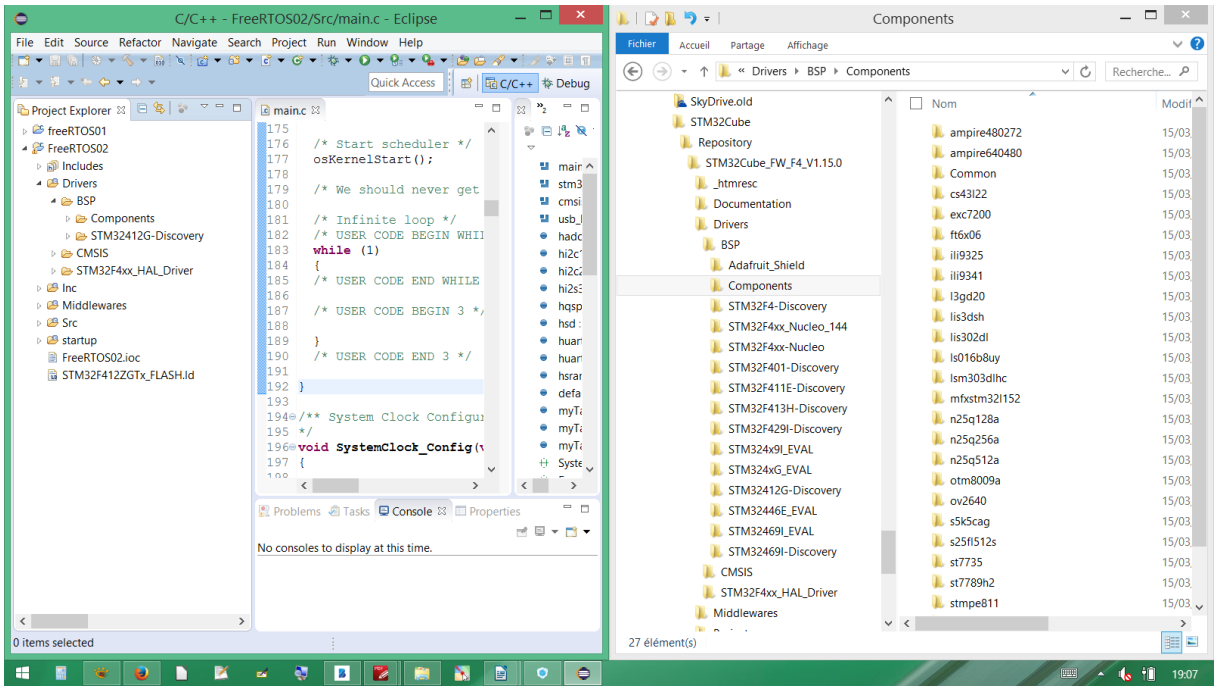



Créer un dossier BSP :

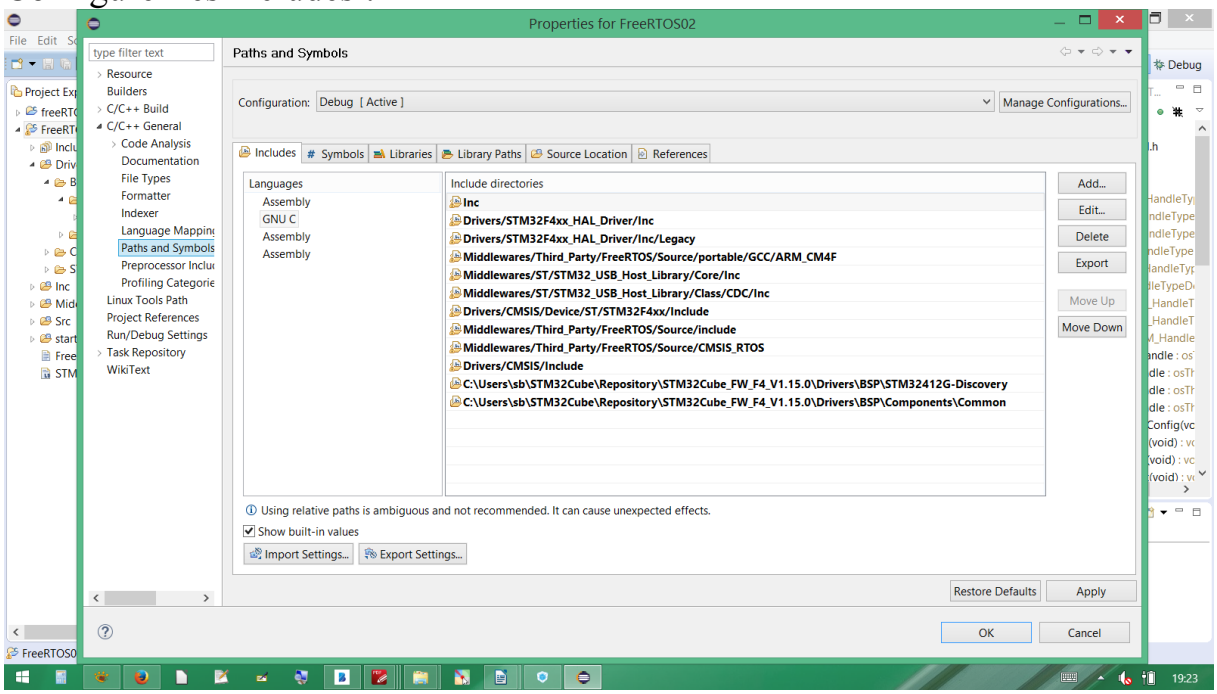



Y copier le dossier STM32412G-Discovery trouvés dans le firmware.

Faire de même avec le dossier "components/common" :




Configurer les includes :



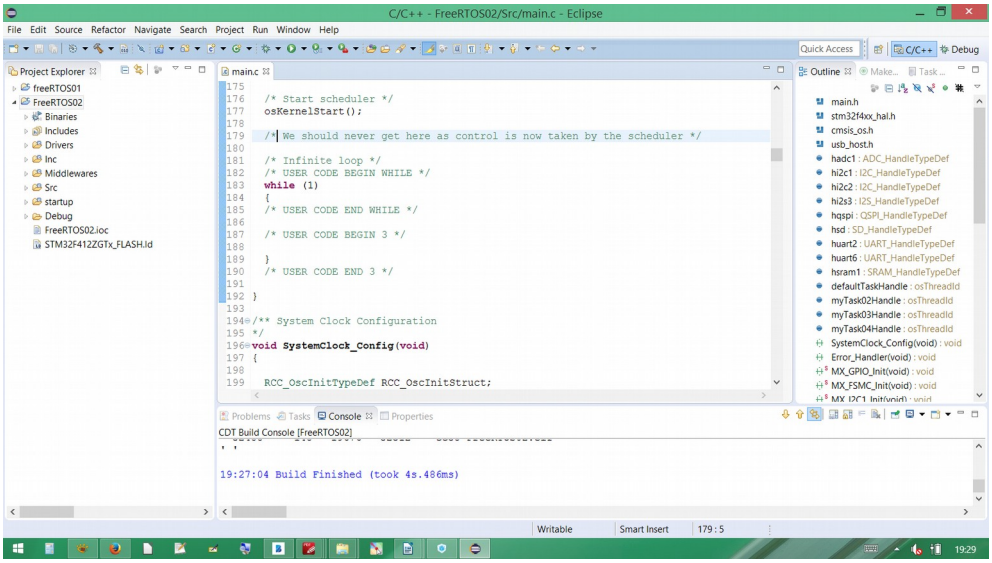

Add...+ FileSystem + pointer sur le chemin du firmware /drivers/BSP/STM32412G
puis sur BSP/Components/common

# 11. COMPILATION

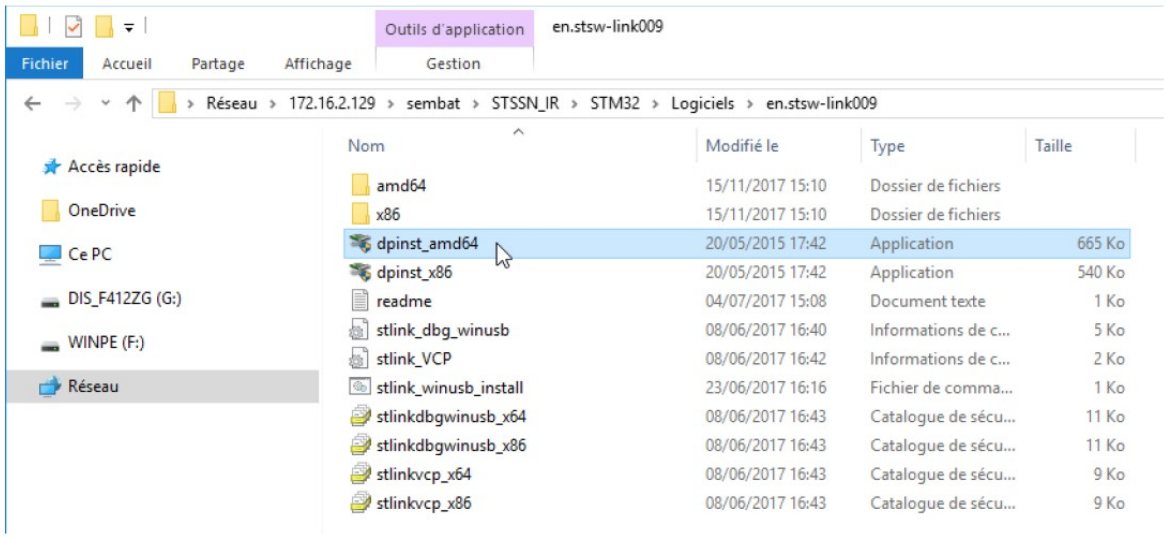
En lançant le built du projet les éventuelles erreurs de drivers apparaissent.
Les régler :

le driver audio peut être "delete" car pas utilisé dans notre projet.
Recompiler

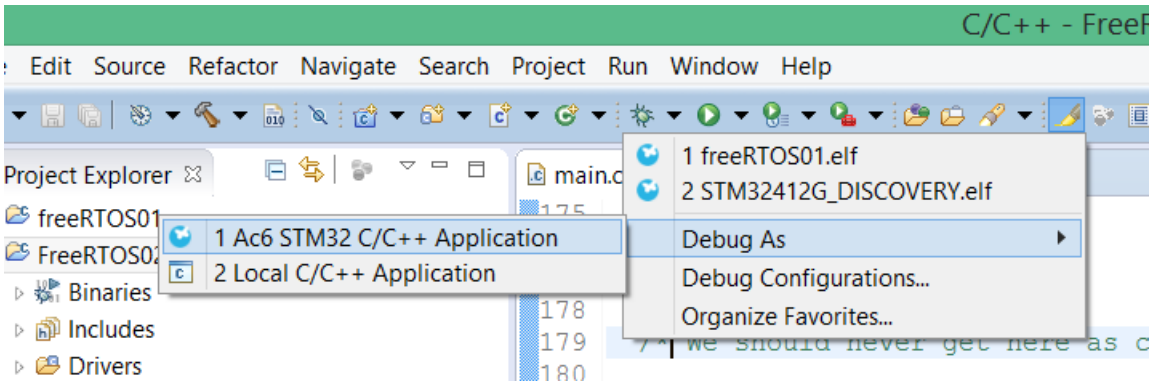
Il n'y a plus d'erreur nous pouvons créer notre programme.



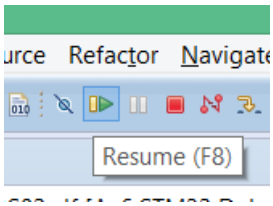

Brancher la maquette sur le port USB : ST-LINK
Vérifier que le driver est ok sinon l'installer.




Le transfert vers la carte d'un programme se fait par :




Puis faire F8 ou cliquer sur :




Vous passez en perspective "debug".
Pour arrêter cliquer sur le carré rouge.
Vous pouvez repasser en perspective C/C++ en cliquant en haut à droite :

# 12. EXERCICES

## 12.1. Clignotement de LED

Faire clignoter la LED1 dans la tâche par défaut (période 1s)

solution :

```
void StartDefaultTask(void const * argument)
{
  /* init code for USB_HOST */
 MX_USB_HOST_Init();


  /* USER CODE BEGIN 5 */
  /* Infinite loop */
  for(;;)
 {


  HAL_GPIO_TogglePin(LED1_GPIO_Port,LED1_Pin);
  HAL_Delay(1000);
  osDelay(1);
 }
  /* USER CODE END 5 */
}
```

## 12.2. Mise en avant des priorités

Faire clignoter les autres LED avec des temps identiques afin de vérifier les priorités des différentes tâches.
Changer les priorités des tâches pour voir l'effet.

Solution :

```
void StartDefaultTask(void const * argument)
{
  /* init code for USB_HOST */
 MX_USB_HOST_Init();

  /* USER CODE BEGIN 5 */
  /* Infinite loop */
  for(;;)
 {
 HAL_GPIO_TogglePin(LED1_GPIO_Port,LED1_Pin);
 HAL_Delay(5000);
  osDelay(1);
 }
  /* USER CODE END 5 */
}

/* StartTask02 function */
void StartTask02(void const * argument)
{
  /* USER CODE BEGIN StartTask02 */
  /* Infinite loop */
  for(;;)
 {
```

```
  HAL_GPIO_TogglePin(LED2_GPIO_Port,LED2_Pin);
  HAL_Delay(500);
    osDelay(1);
  }
    /* USER CODE END StartTask02 */
}


/* StartTask03 function */
void StartTask03(void const * argument)
{
    /* USER CODE BEGIN StartTask03 */
    /* Infinite loop */
    for(;;)
  {
  HAL_GPIO_TogglePin(LED3_GPIO_Port,LED3_Pin);
  HAL_Delay(500);
    osDelay(1);
  }
    /* USER CODE END StartTask03 */
}


/* StartTask04 function */
void StartTask04(void const * argument)
{
    /* USER CODE BEGIN StartTask04 */
    /* Infinite loop */
    for(;;)
  {
  HAL_GPIO_TogglePin(LED4_GPIO_Port,LED4_Pin);
  HAL_Delay(500);
    osDelay(1);
```

```
  }

    /* USER CODE END StartTask04 */

}
```

*Remarque : La tâche IDLE ne se fait plus car les 3 autres sont de "heute priorité".*

# 13. RÉFÉRENCES :

Cours de Maurice Lardellier
FreeRTOS.org
www.logicvoltage.com
http://www.openstm32.org/HomePage
stssnsb.free.fr rubrique : TP IR programation/C sur STM32 platine DISCO STM32F412G