FreeRTOS

FreeRTOS est un système d'exploitation embarqué multitâches temps réel préemptif supporte actuellement 35 architectures.
Nous allons utiliser un RTOS conforme à la spécification RTOS de la norme CMSIS (Cortex Microcontroller Interface Standard). Cette spécification définit une API RTOS standard à utiliser avec les microcontrôleurs basés sur Cortex-M. CMSIS-RTOS est une API commune aux systèmes d'exploitation temps réel, elle fournit toutes les fonctionnalités dont nous aurons besoin pour développer un application temps réel.

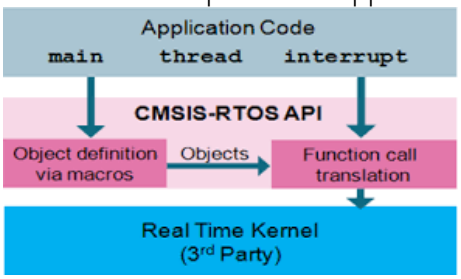


Figure 1:CMSIS RTOS Architecture

Tâche FreeRTOS

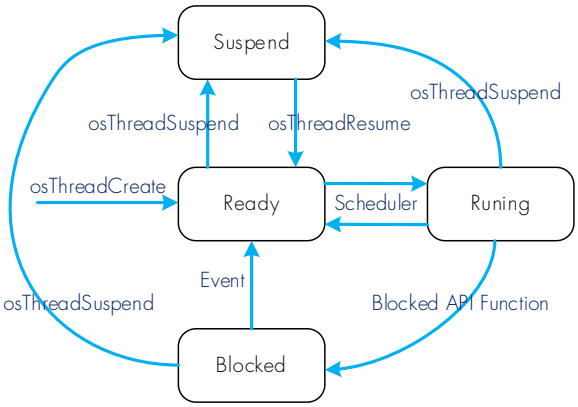
Pour développer une application basée sur un OS, on décompose l'application en un ensemble de tâches. Nous déclarons chaque tâche en tant que fonction C contenant une boucle infinie et ne renvoie pas un résultat.

```
void StartTask1( void const  *argument )
{
            ⋮
            for( ; ; )
            {

            }
}
```

Etat d'une tâche

Une tâche FreeRTOS peut se trouver dans l'un des états suivants :
Prête (Ready) : une tâche qui possède toutes les ressources nécessaires à son exécution. Elle lui manque seulement le processeur.
Active (Running) : Tâche en cours d'exécution, elle est actuellement en possession du processeur.
Attente (Blocked) : Tâche en attente d'un événement (queue de messages, sémaphores, timeout ...). Une A l'arrivé de l'événement, la tâche concernée repasse alors à l'état prêt.
Suspendu (Suspend) : tâche à l'état dormant ; elle ne fait pas partie de l'ensemble des tâches ordonnançables.



Task Control Block (TCB)

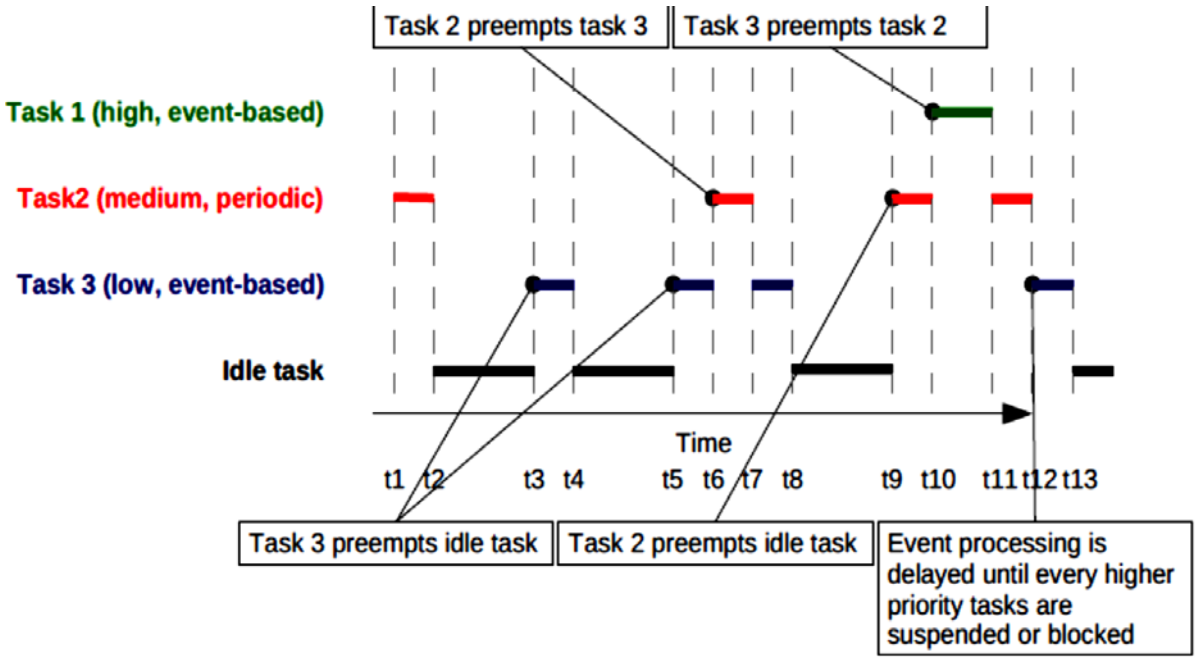
Le TCB est une structure de données contenant les informations nécessaires à la gestion des tâches. Une fois la tâche est créée, FreeRTOS lui assigne un TCB.

Ordonnancement

L'ordonnanceur de FreeRTOS est basé sur la priorité des tâches. En cas où plusieurs tâches partagent le même niveau de priorité, c'est le round-robin qui est utilisé.

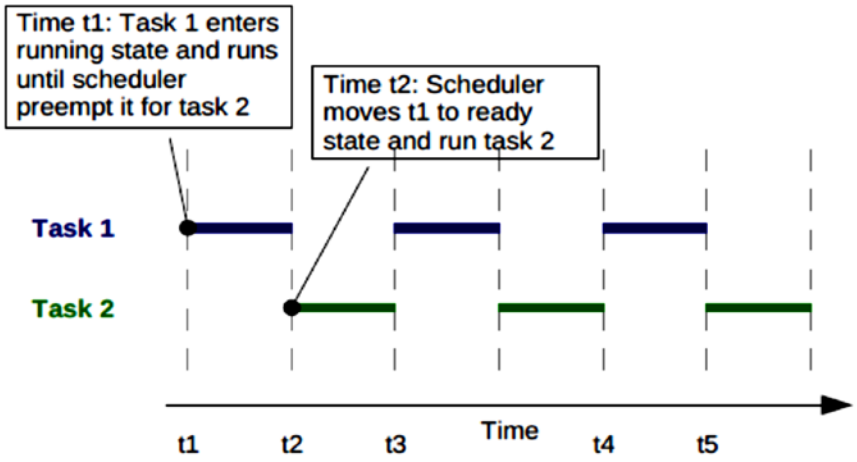
Priorité

La priorité est un nombre donné à une tâche lorsqu'elle est créée ou modifiée manuellement à l'aide de primitive *osThreadSetPriority*. Le nombre maximum de priorités est défini par CMSIS RTOS à osPriorityRealtime.



Round robin

Les tâches d'égale priorité sont traitées de manière égale par l'ordonnanceur ; il choisit à chaque quantum de temps une tâche selon l'algorithme round robin.



Gestion des tâches

Déclaration d'une tâche

Pour développer une application basée sur un OS, on décompose l'application en un ensemble de tâches. Dans FreeRTOS une tâche est fonction C contenant une boucle infinie et ne renvoie pas un résultat.

On définit pour chaque tâche un identificateur (Handle)

```
osThreadId  Task1Handle;
```

On définit aussi les paramètres de structure da la tâche.

name	Nom de la tâche.
pthread	Adresse de départ de la Tâche.
tpriority	Définit la priorité à laquelle la tâche sera exécutée. CMSIS RTOS définie les priorités suivantes : - osPriorityNormal = 0 - osPriorityAboveNormal = +1 - osPriorityHigh = +2 - osPriorityRealtime = +3
instances	Nombre maximum d'instances
stacksize	Le noyau attribue à chaque tâche lors de sa création sa propre pile. La valeur *stacksize* indique au noyau la taille de la pile.

```
osThreadDef(Task1, StartTask1, osPriorityNormal, 0, 128);
```

Une tâche est une fonction dont le traitement se trouve dans une boucle infinie.

```
void StartTask1( void const *argument )
{/* déclaration des variables locales */
    int iVariableExample = 0;
      for( ;; )
      {
            /* code de la fonction */
      }
}
```

Création d'une tâche

Une tâche est créée par l'intermédiaire de la fonction osThreadCreate().

```
Task1Handle = osThreadCreate(osThread(Task1), NULL);
```

Le pointeur NULL indique qu'il n'y a pas d'argument à passer à la tâche.

Exemple de création de tâche :

```
osThreadId myTask02Handle ;

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

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

void StartTask02(void const * argument)
{
  for(;;)
  { //…..
    osDelay(100);
  }
}
```

Suppression d'une tâche

Une tâche créée peut être supprimé par la primitive :

```
osStatus osThreadTerminate (osThreadId thread_id) ;
```

Exemple :

```
osThreadTerminate (myTask02Handle) ;
```

Obtenir l'identificateur de la tâche en cours d'exécution

On peut obtenir l'identificateur de la tâche en cours d'exécution par la primitive :

```
osThreadId osThreadGetId (void) ;
```

Exemple :

```
osThreadId CurrentTaskId ;
CurrentTaskId =  osThreadGetId();
```

Suspendre une tâche

Syntaxe : `osStatus osThreadSuspend (osThreadId thread_id) ;`

Exemple : `osThreadSuspend(myTask02Handle) ;`

Le noyau peut être suspendu en invoquant la fonction « osStatus osThreadSuspendAll (void)». Cette fonction suspend toute activité du noyau sauf les interruptions. Le noyau peut reprendre son activité par la fonction « osStatus osThreadResumeAll(void) ».

Redémarrer une tâche suspendue

La primitive osThreadResume permet de redémarrer une tâche déjà suspendue.

Syntaxe : `osStatus osThreadResume (osThreadId thread_id) ;`

Contrôle des tâches

osDelay

Spécifie le temps (en millisecondes) pendant lequel la tâche reste bloquer, ce nombre est passé comme paramètre.

Syntaxe : *osStatus osDelay (uint32_t millisec)*

Exemple :

```
void toggleTask( void const *argument )
{
/* bloquer pendant 500ms. */
      for( ;; )
      {
            /* changer l'état de la LED chaque 500ms, revient à bloquer
               la tâche chaque changement */
            ToggleLED();
            osDelay(500);
      }
}
```

osDelayUntil

La fonction osDelay n'est pas précise. Si on veut bloquer une tâche à des intervalles de temps réguliers, il vaut mieux utiliser la primitive osDelayUntil. Cette fonction spécifie le temps absolu pendant lequel la tâche va se débloquer.

Syntaxe : `void osDelayUntil(uint32_t *pxPreviousWakeTime, uint32_t millisec );`

pxPreviousWakeTime : valeur du compteur juste avant le blocage de la tâche, elle est initialisée par la valeur courent du compteur des ticks de l'ordonnaceur en invoquant la primitive **osKernelSysTick().**

millisec : temps en milliseconce pendant lequel la tâche reste bloquer.

Exemple :

```
void myTaskFunction( void const * argument )
{
```

```
        Uint32_t xLastWakeTime;
        Uint32_t xFrequency = 10;
        // Initialise  xLastWakeTime à la valeur courante du timer.
        xLastWakeTime = osKernelSysTick() ;
        for( ;; )
        {
        // Mise en attente pendant xFreqeuncy Ticks.
        osDelayUntil( &xLastWakeTime, xFrequency );

        }
}
```

osThreadSetPriority

Change la priorité d'une tâche
Syntaxe : *osStatus osThreadSetPriority (osThreadId thread_id, osPriority priority);*
Exemple :

```
osThreadSetPriority(myTask02Handle,osPriorityHigh);
```

osThreadGetPriority

Renvoie la priorité d'une tâche.
Syntaxe : *osPriority **osThreadGetPriority** (osThreadId thread_id) ;*

Contrôle du noyau

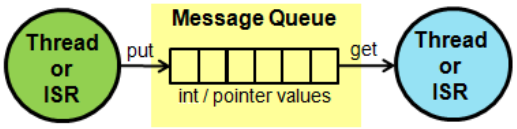
osKernelStart

`osStatus osKernelStart (void)` : Démarre le processus du noyau temps réel.
Exemple :

```
void main( void )
{
// Créer les tâches puis démarrer le noyau
myTask02Handle = osThreadCreate(osThread(myTask02), NULL);
        .
        .
        .
osKernelStart () ;   // démarrage du noyau.
}
```

Gestion de la queue

Les files d'attente sont la principale forme de communication Inter-tâches. Ils peuvent être utilisés pour envoyer des messages entre les tâches ou entre les interruptions et les tâches. Dans la plupart des cas, ils sont utilisés comme des files FIFO. Les données à transmettre peuvent être de type entier ou pointeur.



Pour créer un Message Queue, il faut suivre les étape suivantes :

1. déclarer l'identificateur du Message Queue
   `osMessageQId myQueue01Handle;`
2. allouer de la mémoire au message
   `osMessageQDef(name, queue_sz, type) ;`

paramètres :

- name : nom du message
- Queue_sz : nombre maximum de message à placer dans la Queue
- Type : type de message : entier , structure…

Exemple : `osMessageQDef(myQueue01, 10, uint16_t) ;`
Il reste maintenant de créer le Message Queue :

```
        myQueue01Handle = osMessageCreate(osMessageQ(myQueue01),NULL);
```

Cette fonction retourne l'identificateur du Message Queue ou NULL en cas d'erreur.

osMessagePut

Une fois le Message Queue est créé, il est possible déposer des informations dans la file de massage :

```
osStatus osMessagePut (osMessageQId queue_id, uint32_t info, uint32_t millisec)
```

queue_id : l'indicateur retourné par la fonction osMessageCreate.
info : information à poster ; peut-être un entier ou un pointeur.
millisec : temps en milliseconde maximal pendant lequel la tâche reste bloquer en attendant un espace disponible dans la file d'attente. *osWaitForever* pour une attente infinie.
Exemple :

```
uint16_t val;
osMessagePut(myQueue01Hande, ( uint32_t ) val, osWaitForever) ;
```

osMessageGet

La tâche réceptrice du message utilise la primitive osMassageGet.

```
osEvent osMessageGet (osMessageQId queue_id, uint32_t millisec);
```

queue_id : l'indicateur retourné par la fonction osMessageCreate.
millisec : temps en milliseconde maximal pendant lequel la tâche reste bloquer en attendant la présence d'un élément dans la file d'attente. Si *millisec = 0*, la fonction retourne immédiatement si la file est vide. La valeur osWaitForever permet une attente infinie.
osEvent : est une structure contenant les événements renvoyés par la fonction CMSIS-RTOS :

```
 typedef struct  {
      osStatus   status;    //osOK, osEventMessage or osErrorParameter;
      union  {
        uint32_t    v;      //  message as 32-bit value
        void       *p;      //   message or mail as void pointer
        int32_t  signals;   //  signal flags
     } value;                              ///< event value
     union  {
        osMailQId   mail_id;          //  mail id
        osMessageQId  message_id;     //  message id
     } def;                           //  event definition
} osEvent;
```

On s'intéresse à l'union « value », qui contient une valeur (v) ou un pointeur (p).

Gestion des ressources

Les Sémaphores

Le sémaphore est la méthode la plus utilisée pour restreindre l'accès à des ressources partagées ou synchroniser les processus dans un environnement de programmation concurrente. Un sémaphore est un conteneur de jetons ; une tâche peut acquérir un jeton à travers une primitive système ; si le sémaphore contient un ou plusieurs jetons, la tâche continu sont exécution et le nombre de jetons se décrémente. S'il n'y a aucun jeton dans le sémaphore, la tâche se met à l'état d'attente jusqu'à la libération (remise) d'un jeton par une autre tâche. A tout moment, une tâche peut remettre (ajouter) un jeton au sémaphore, ce qui entraine l'incrémentation de son compte.

osSemaphoreCreate

Pour utiliser un sémaphore, vous devez commencer par déclarer l'identificateur et la variable sémaphore.

```
osSemaphoreId mySem01Handle;    // identificateur du sémaphore
osSemaphoreDef(mySem01);           // variable sémaphore
```

Initialiser la variable sémaphore par le nombre des jetons (2 jetons par exemple).

```
 mySem01Handle = osSemaphoreCreate(osSemaphore(mySem01), 2);
```

Dans CMSIS RTOS un sémaphore binaire est un sémaphore ayant un seul jeton.

osSemaphoreWait

Primitive de demande de ressource. Cette opération est équivalente à une opération P (). Si la ressource n'est pas disponible, la tâche sera bloqué jusqu'à la disponibilité de la ressource ou l'écoulement du délai.

Syntaxe : *int32_t osSemaphoreWait (osSemaphoreId semaphore_id, uint32_t millisec);*

Semaphore_id : l'identificateur de la sémaphore (handle)

millisec : temps d'attente de la disponibilité de la ressource en milliseconde. La valeur osWaitForever permet d'avoir une attente infinie.

Elle retourne le numéro du jeton.

Exemple :

```
osSemaphoreWait( mySem01Handle, 0  ) ;
```

La tâche ne sera pas bloquée si la ressource n'est pas disponible.

```
osSemaphoreWait( mySem01Handle, osWaitForever  ) ;
```

La tâche reste bloquée jusqu'à la disponibilité de la ressource.

osSemaphoreRelease

Primitive de libération de ressource. Cette opération est équivalente à une opération V ()

Syntaxe : *osStatus osSemaphoreRelease (osSemaphoreId semaphore_id) ;*

Exemple :

```
osSemaphoreRelease(mySem01Handle) ;
```

Mutex

L'exclusion mutuelle (communément appelée Mutex) est utilisée dans divers systèmes d'exploitation pour la gestion des ressources. De nombreuses ressources d'un microcontrôleur peuvent être utilisées de manière partagée, mais uniquement par un thread à la fois (par exemple : des canaux de communication, de la mémoire et des fichiers). Les mutex sont utilisés pour protéger l'accès à une ressource partagée.

Les sémaphores binaires et les mutex sont très similaires ; les Mutex incluent un mécanisme d'héritage de prioritaire, les sémaphores binaires ne le font pas. Cela fait des sémaphores binaires le meilleur choix pour implémenter la synchronisation entre les tâches, et les mutex pour implémenter une simple exclusion mutuelle.

Pour utiliser les mutex, il faut déclarer la variable mutex et l'initialiser :

```
osMutexDef (uart_mutex);            // Déclarer le mutex
osMutexId  (uart_mutex_id);         // identificateur de Mutex
```

osMutexCreate

Crée un mutex

Syntaxe *osMutexId osMutexCreate (const osMutexDef_t *mutex_def))*

Exemple :

```
osMutexDef (uart_mutex);
osMutexId  (uart_mutex_id);
void vATask( void const * argument )
{
      uart_mutex_id = osMutexCreate (osMutex(uart_mutex));
      if( uart_mutex_id != NULL )
      {
            // Le sémaphore est créé.
      }
}
```

osMutexWait

Demande d'accès à la ressource partagée

Syntaxe : *osStatus osMutexWait (osMutexId mutex_id, uint32_t millisec)*

mutex_id : l'identificateur du mutex (handle)

millisec : temps d'attente de la disponibilité de la ressource en milliseconde. La valeur osWaitForever

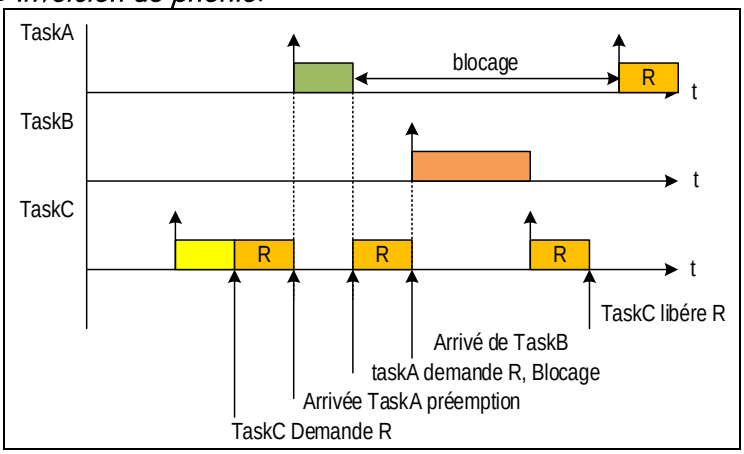
permet d'avoir une attente infinie.
Exemple :
```
osMutexWait( uart_mutex_id, osWaitForever  ) ;
```
La tâche reste bloquée jusqu'à la disponibilité de la ressource.

osMutexRelease

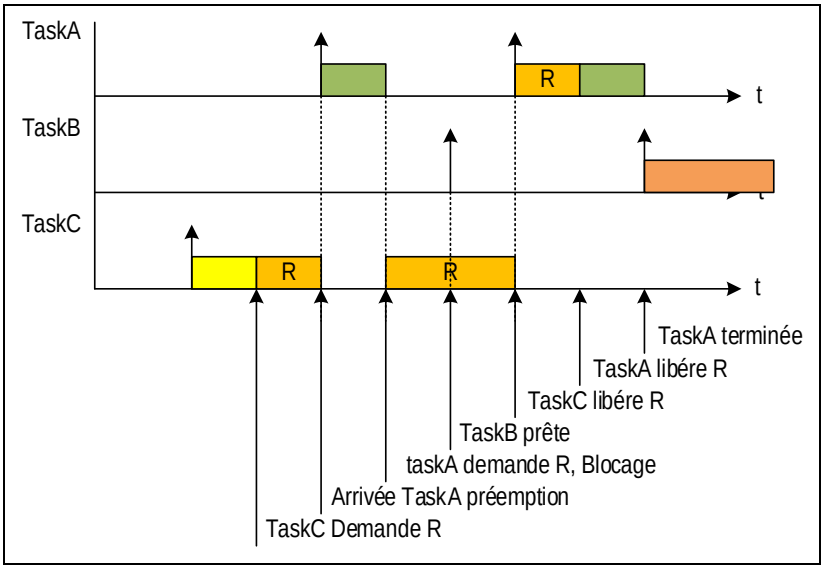
Primitive de libération de ressource.
Syntaxe : *osStatus osMutexRelease (osSemaphoreId semaphore_id) ;*
Exemple :
```
osMutexRelease(uart_mutex_id) ;
```

Problème d'inversion de priorité

Soient trois tâches TaskA (haute priorité), TaskB et taskC (Basse priorité) et R une ressource partagée par les trois tâches. Supposons que TaskC est en possession de la ressource R ; TaskC est préemptée par TaskA plus prioritaire, lorsque TaskA souhaite prendre la ressource R elle sera bloquée à cause de la non disponibilité de R (déjà pris par TaskC), dans ce cas TaskB peut être exécutée si elle est prête. Ce phénomène est appelé *inversion de priorité*.



Pour pallier ce défaut, on attribue à la tâche en possession de ressource (TaskC), *la priorité de la tâche la plus prioritaire des tâches demandant la ressource* (héritage de priorité). Dans ce cas, lorsque TaskA est bloquée, c'est la TaskC qui reprend son exécution et non pas TaskB (TaskC à la même priorité que TaskA). TaskC revient à sa priorité ordinaire quand elle sort de la section critique (libère la ressource).



Exemples :

EXP1 : communication inter-tâches (utilisation de Massage Queue)

myTask01 : envoie unpointeur sur une structure nommée **dataa** dans la file des messages toutes les secondes,

Interruption TIM10 : envoie une valeur entière Val dans la file des messages toutes les secondes,

myTask02 : consomme le message envoyé par myTask01 et l'affiche sur le terminal via la fonction printf,

myTask03 : consomme le message envoyé par l'interruption TIM10 et l'affiche sur le terminal via la fonction printf,

```
osThreadId myTask01Handle;
osThreadId myTask02Handle;
osThreadId myTask03Handle;
osMessageQId myQueue01Handle;
osMessageQId myQueue02Handle;


/* ******************************************
  structure envoyé par la tache01 et reçu par tâche02   */
typedef struct {
      char*msg;
      uint16_t sVal;
}dataa;
/* ******************************************
  valeur envoyée par l'interruption timer 10 et reçu par tâche03   */
uint16_t val = 20;

void main()
{
  /* definition and creation of myQueue01 */
      osMessageQDef(myQueue01,5,dataa);
      myQueue01Handle = osMessageCreate(osMessageQ(myQueue01),NULL);
   /* definition and creation of myQueue02 */
      osMessageQDef(myQueue02,5,uint16_t);
      myQueue02Handle = osMessageCreate(osMessageQ(myQueue02),NULL);

  /* definition and creation of myTask01 */
  osThreadDef(myTask01, StartTask01, osPriorityNormal, 0, 128);
  myTask01Handle = osThreadCreate(osThread(mytTask01), NULL);

  /* definition and creation of myTask02 */
  osThreadDef(myTask02, StartTask02, osPriorityNormal, 0, 128);
  myTask02Handle = osThreadCreate(osThread(myTask02), NULL);

  /* definition and creation of myTask03 */
  osThreadDef(myTask03, StartTask03, osPriorityNormal, 0, 128);
  myTask03Handle = osThreadCreate(osThread(myTask03), NULL);

   /* Start Scheduler */
      osKernelStart();
      while(1);
}
void StartTask01(void const * argument)
{     dataa dd ,*ptrs;
      ptrs = &dd;
      ptrs->msg = "FreeRtos";
      ptrs->sVal = 10;

  for(;;)
  {
```

```
      osMessagePut(myQueue01Handle,(uint32_t)ptrs,osWaitForever);
            ptrs->sVal++;
      osDelay(1000);
  }
}


void StartTask02(void const * argument)
{

      osEvent pevt;
      dataa *rptr;
  for(;;)
  {
            pevt= osMessageGet(myQueue01Handle, osWaitForever);
            rptr = (dataa*)pevt.value.p;
            printf("task2 sval = %u",rptr->sVal);
            printf("    nom = %s\n",rptr->msg);
  }
}
void StartTask03(void const * argument)
{

      osEvent evt;
      uint16_t rval;
  /* Infinite loop */
  for(;;)
  {
            evt= osMessageGet(myQueue02Handle, osWaitForever);
            rval = evt.value.v;
            printf("task3 rval = %u\n",rval);
  }
}

void TIM1_UP_TIM10_IRQHandler(void)
{
      HAL_TIM_IRQHandler(&htim10);
      osMessagePut(myQueue02Handle,val,0);
}
```

Debug (printf) Viewer

```
task2 sval = 11    nom = FreeRtos       /* envoyé par myTask01 */
task3 rval = 20                         /* envoyé par L'interruption TIM10 */
task2 sval = 12    nom = FreeRtos       /* envoyé par myTask01 */
task3 rval = 20                         /* envoyé par L'interruption TIM10 */
task2 sval = 13    nom = FreeRtos       /* envoyé par myTask01 */
task3 rval = 20                         /* envoyé par L'interruption TIM10 */
```

EXP2 : Synchronisation par Sémaphore

Par défaut l'ordonnanceur exécute myTask01 avant myTask02, les sémaphores permettent de synchroniser les tâches afin d'inverser l'ordre de leur exécution (myTask02 précède myTask01).

```
osThreadId myTask01Handle;
osThreadId myTask02Handle;
osSemaphoreId myBinarySem01Handle;

void main()
{
  osSemaphoreDef(myBinarySem01);
  myBinarySem01Handle = osSemaphoreCreate(osSemaphore(myBinarySem01), 1);

  /* definition and creation of mytTask01 */
  osThreadDef(myTask01, StartTask01, osPriorityNormal, 0, 128);
  myTask01Handle = osThreadCreate(osThread(mytTask01), NULL);

  /* definition and creation of myTask02 */
  osThreadDef(myTask02, StartTask02, osPriorityNormal, 0, 128);
  myTask02Handle = osThreadCreate(osThread(myTask02), NULL);

  osMutexWait(myMutex01Handle, 0);
       osKernelStart();
      while(1);
}
void StartTask01(void const * argument)
{
  for(;;)
  {
      osSemaphoreWait(myBinarySem01Handle, osWaitForever);
      printf("task1 is runing now --");
      osDelay(1000);
  }
}
void StartTask02(void const * argument)
{
  for(;;)
  {
      printf("task2 is runing now --");
      osSemaphoreRelease(myBinarySem01Handle);
      osDelay(1000);
  }
}
```

Avant la synchronisation

Après synchronisation

EXP3 : Protection des ressources partagées par Mutex.

La fonction *printf* de *debug viewer* est considérée comme ressource partagée. Nous utilisons les Mutex pour afficher les messages convenablement (sans chevauchement).

```
osThreadId myTask01Handle;
osThreadId myTask02Handle;
osMutexId myMutex01Handle;

void main()
{
  osMutexDef(myMutex01);
  myMutex01Handle = osMutexCreate(osMutex(myMutex01));

  osThreadDef(myTask01, StartTask01, osPriorityNormal, 0, 128);
  myTask01Handle = osThreadCreate(osThread(mytTask01), NULL);

  osThreadDef(myTask02, StartTask02, osPriorityNormal, 0, 128);
  myTask02Handle = osThreadCreate(osThread(myTask02), NULL);

       osKernelStart();
      while(1);
}
void StartTask01(void const * argument)
{
  for(;;)
  {
      osMutexWait(myMutex01Handle, osWaitForever);
            printf("myTask01 est en exécution --");
            HAL_Delay(100);
            printf("myTask01 est encore en exécution \n");
      osMutexRelease(myMutex01Handle);
   osDelay(1000);
  }
}
void StartTask02(void const * argument)
{
  /* USER CODE BEGIN StartTask03 */

  /* Infinite loop */
  for(;;)
  {   osMutexWait(myMutex01Handle, osWaitForever);
            printf("myTask02 est en exécution --");
            HAL_Delay(100);
            printf("myTask02 est encore en exécution \n");
      osMutexRelease(myMutex01Handle);
      osDelay(1000);
  }
}
```

Avant la protection de la ressource : les messages se chevauchent.

```
Debug (printf) Viewer
myTask01 est en exécution -- myTask02 est en exécution -- myTask01 est encore en exécution
myTask02 est encore en exécution
myTask01 est en exécution -- myTask02 est en exécution -- myTask01 est encore en exécution
myTask02 est encore en exécution
myTask01 est en exécution -- myTask02 est en exécution -- myTask01 est encore en exécution
myTask02 est encore en exécution
```

Après protection de ressource :

```
Debug (printf) Viewer
myTask01 est en exécution -- myTask01 est encore en exécution
myTask02 est en exécution -- myTask02 est encore en exécution
myTask01 est en exécution -- myTask01 est encore en exécution
myTask02 est en exécution -- myTask02 est encore en exécution
myTask01 est en exécution -- myTask01 est encore en exécution
```