II2 - RTOS

Saša Radosavljevic

Mars 2021

1 TP

1.1 Raccourcis et astuces

- **CTRL + F :** Rechercher un terme dans le code. (Permet également de faire du remplacement)
- **CTRL + FIN :** Permet d'arriver à la fin du code. (C'est cool pour 1600 lignes de code)
- **CTRL + SPACE :** Si vous avez oublier le nom complet d'une variable ou d'une fonction, cela permet de faire de l'auto-completion ou d'activer les propositions.
- **TAB — MAJ+TAB :** Permet d'indenter/désindenter les lignes sélectionnées.
- **CTRL + MAJ + C :** Permet de commenter/dé-commenter les lignes sélectionnées.

Si vous ne savez pas toutes les fonctions disponibles avec une librairie, vous pouvez à partir de la navigation de fichiers vous rendre dans la librairie désirée et observer à partir de l'aperçu Outline les prototypes :

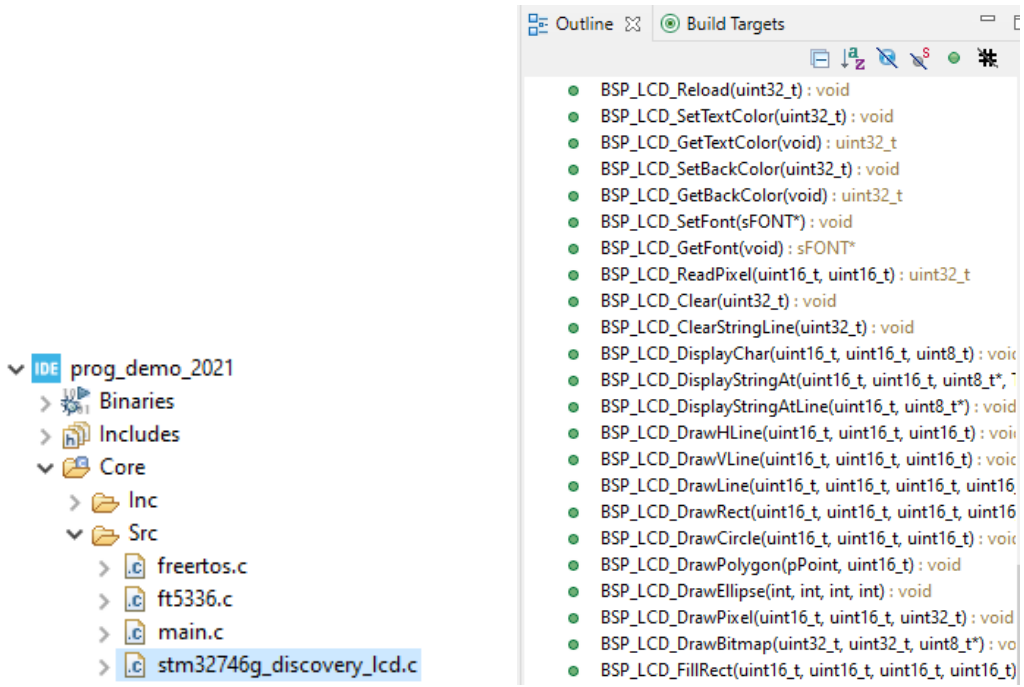

Figure 1: Fonctions

Pour ouvrir l'aperçu outline :

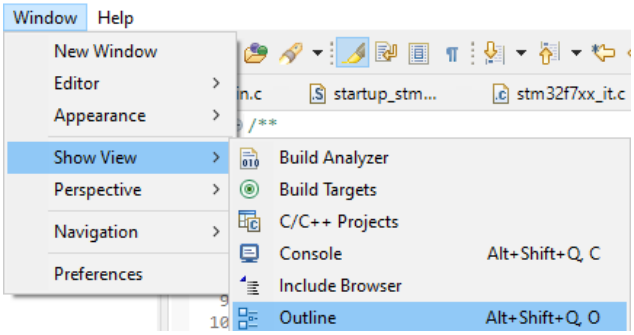

Figure 2: Aperçu outline

1.2 Creation d'un nouveau projet

Pour éviter de refaire toutes les configurations, vous pouvez repartir d'un fichier de configuration .ioc déjà créé auparavant **ou bien faire un copier/coller d'un projet déjà existant en pensant à renommer le .ioc et à supprimer le dossier debug**. Pour ce qui est de la création à partir d'un fichier de configuration, il faut aller dans File>New>STM32 Project from an existing STM32.

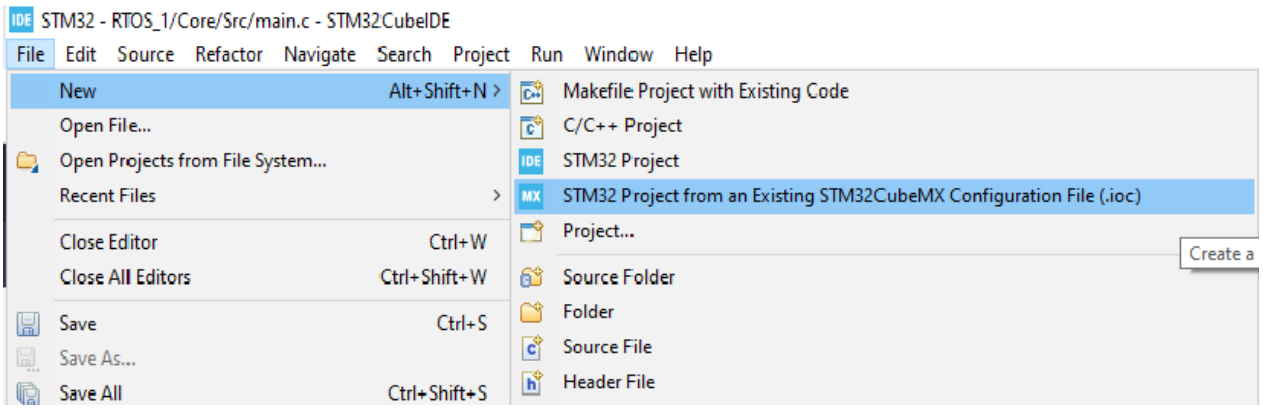

Figure 3: Création d'un projet à partir d'un fichier de config

Il faut ensuite aller chercher le fichier (par exemple dans prog_demo_2021).

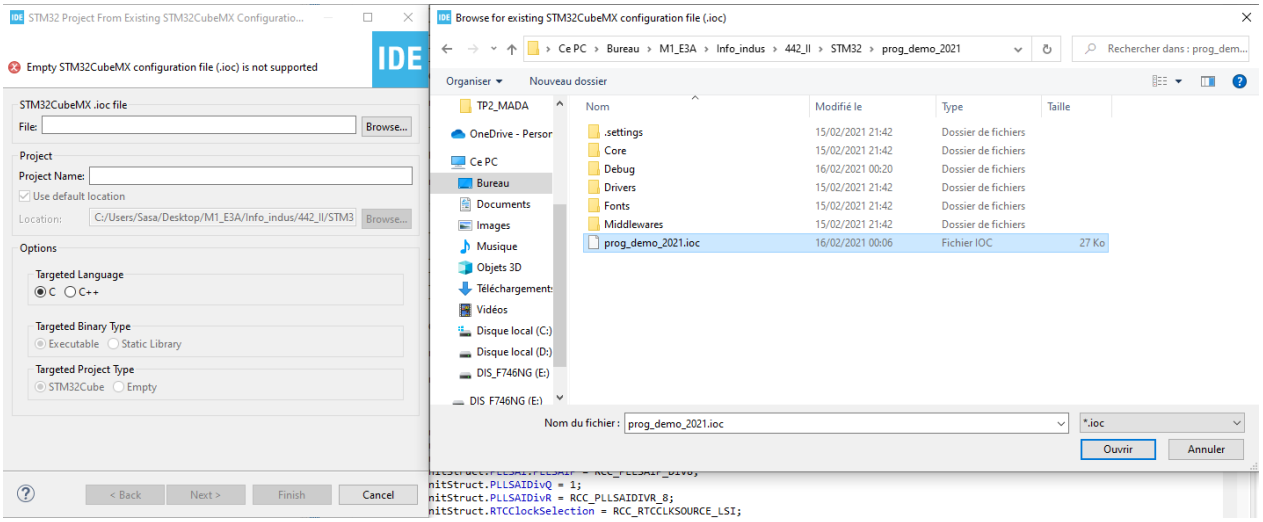

Figure 4: Création d'un projet à partir d'un fichier de config

Une fois ceci fait, un simple build du projet devra être bon. Il ne faudra pas oublier que le main a été recréé par défaut vierge, à vous de remettre les bouts de code que vous avez déjà fait et qui vous seront utiles. **(Le plus simple reste quand même de copier/coller un projet déjà fait)**

1.3 Configuration RTOS

Dans cette partie, il faudra aller tripoter le fichier de configuration .ioc. Il est généralement marqué d'une petite icône bleue MX. https://www.freertos.org/features.html

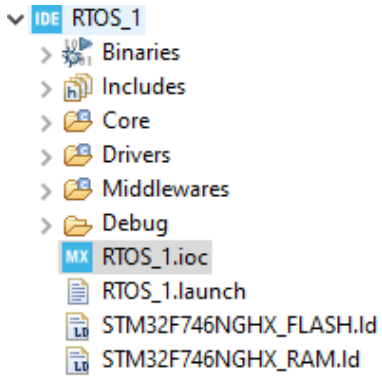

Figure 5: Configuration RTOS

Dans la fenêtre qui vient de s'ouvrir, vous pourrez naviguer dans les différentes catégories de configuration (TOR, ADC, UART, FREERTOS).

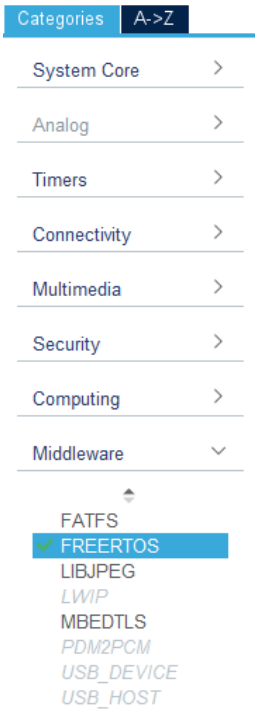

Figure 6: Configuration RTOS

Ici, vous aurez accès à BEAUCOUP de paramètres de configuration. Nous verrons dans l'ordre du cours à quoi servent ces paramètres (pour ceux que l'on verra en cours).

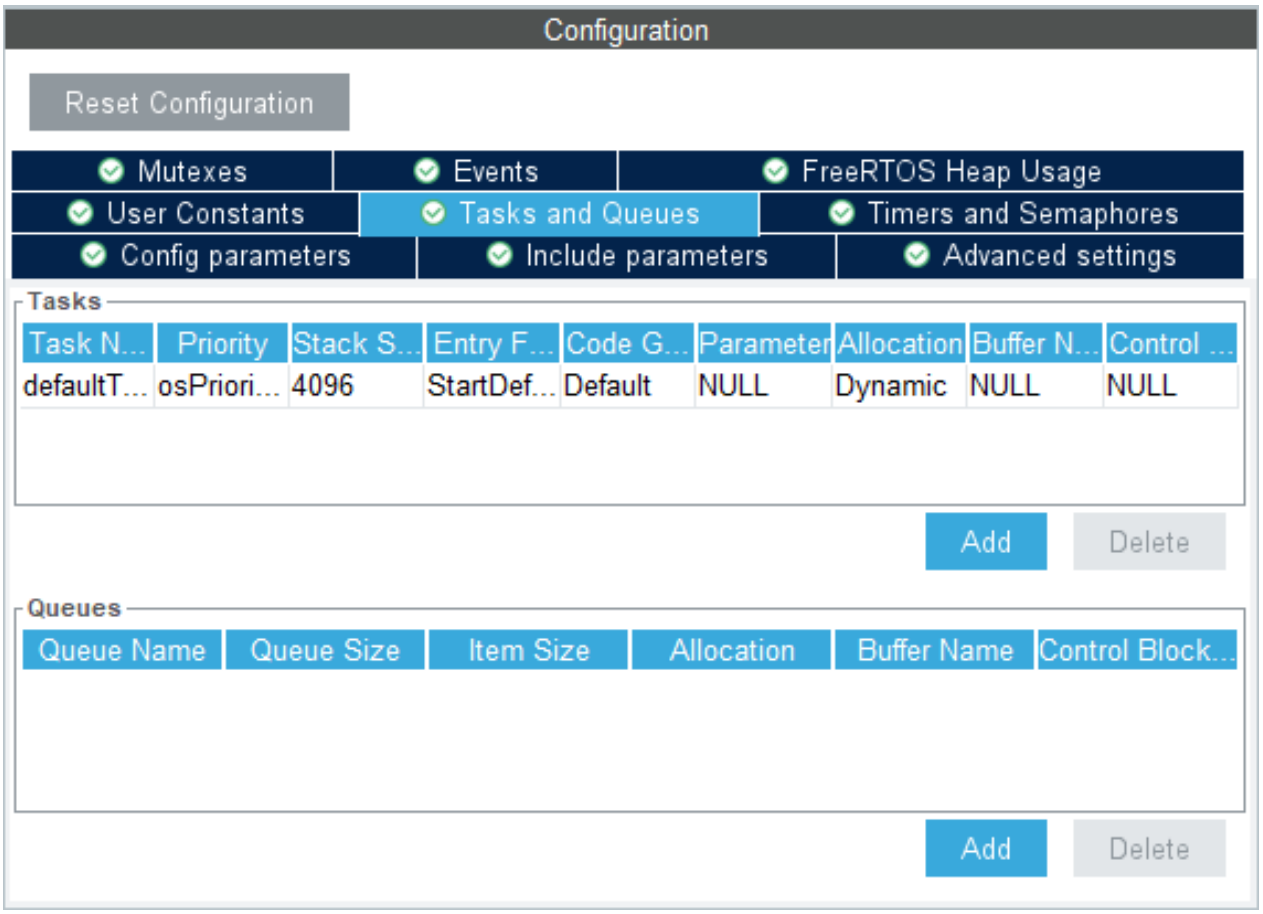

Figure 7: Configuration RTOS

Nous nous intéresserons en premier lieu aux Tâches que nous avons vus en premier ! (pour ce qui est des queues de communication inter-tâches vous pouvez vous renseigner ici https://www.freertos.org/Embedded-RTOS-Queues.html)

On peut voir qu'une tâche est déjà présente (defaultTask) elle est en quelque sorte le nouveau main que l'on avions l'habitude d'utiliser (En effet le while(1) du main devient inutilisable une fois l'os activé). Pour en rajouter, il suffit d'appuyer sur le bouton "Add".

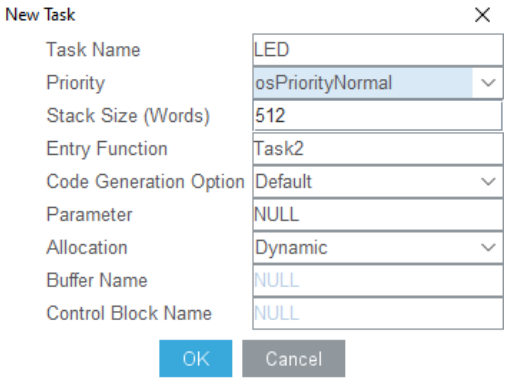

Figure 8: Configuration Tâches

Le Task Name importe peu, c'est à votre goût. Ce qui nous intéresse ici, ce sont la priorité (Au départ on pourra mettre toutes nos tâches avec la même priorité, ce qui va rapidement changer avec vos objectifs).

Pour être en mesure de gérer l'affichage, un stockage de 512 octets sera nécessaire. l'Entry Function sera le nom visible dans le programme. Pas besoin de s'occuper du reste des paramètres, du moins pour le moment.

Enfin, il ne faudra pas oublier de générer le code pour appliquer les modifications dans le programme "main.c". Encore une fois, veillez à bien respecter les balises /* User code */ pour éviter les mauvaises surprises lors de la régénération du code.

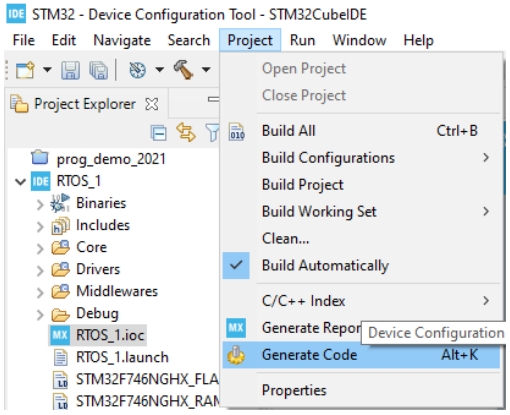

Figure 9: Génération du code à partir du fichier de configuration

On peut voir que la modification a bien été prise en compte, il se peut que le code soit un peu loin vers le bas. Une combinaison CTRL + FIN sur votre clavier permet d'arriver à la fin du code.

```
1328 void Led1(void const * argument)
1329 {
1330   /* USER CODE BEGIN Led1 */
1331   /* Infinite loop */
1332   for(;;)
1333   {
1334     osDelay(1);
1335   }
1336   /* USER CODE END Led1 */
1337 }
1338
1339 /* USER CODE BEGIN Header_Led2 */
1340 /**
1341 * @brief Function implementing the LED2 thread.
1342 * @param argument: Not used
1343 * @retval None
1344 */
1345 /* USER CODE END Header_Led2 */
1346 void Led2(void const * argument)
1347 {
1348   /* USER CODE BEGIN Led2 */
1349   /* Infinite loop */
1350   for(;;)
1351   {
1352     osDelay(1);
1353   }
1354   /* USER CODE END Led2 */
1355 }
```

Figure 10: main.c modifié

Vous êtes maintenant prêts à vous amuser avec le temps réel !

1.4 Configuration Mutex (Mutual exclusion)

Afin de créer un Mutex, pas de secret, il faut se rendre à nouveau dans le fichier de configuration .ioc. Au niveau de la catégorie FreeRTOS vous devriez donc apercevoir l'onglet Mutexes.

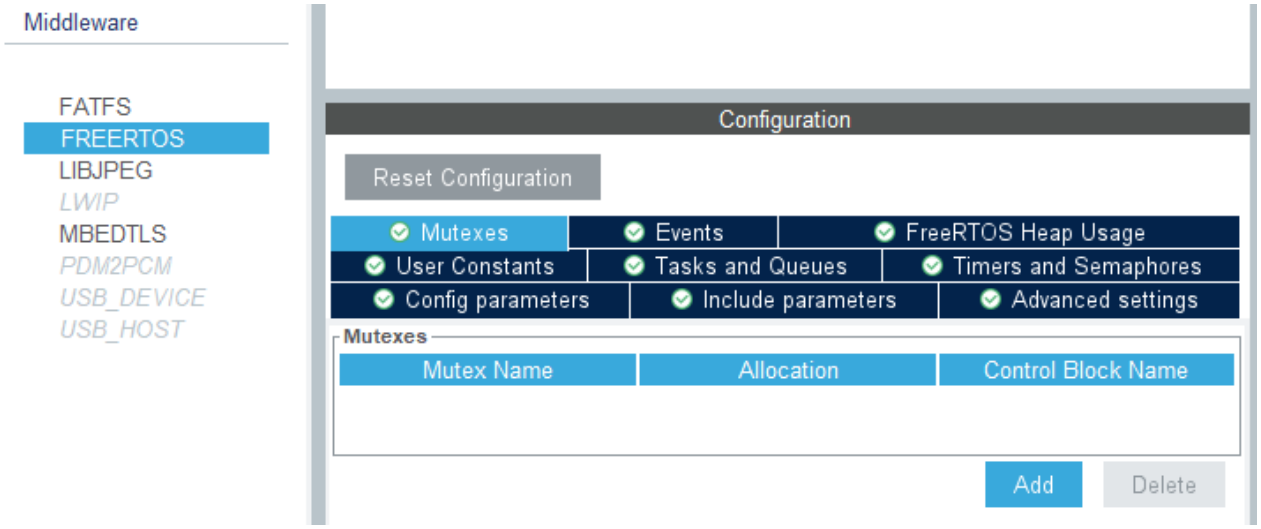

Figure 11: Mutex

C'est ici que vous pourrez créer un nouveau Mutex. Il suffit seulement de cliquer sur le bouton Add et d'entrer un nom.

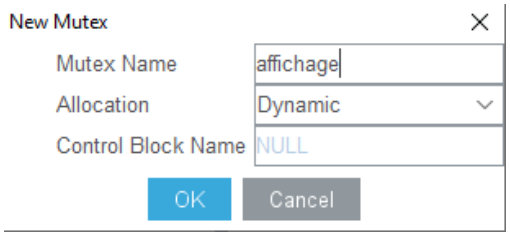

Figure 12: Config Mutex

Vous devriez alors, une fois le code généré, obtenir ceci dans votre main.c :

```
187   /* Create the mutex(es) */
188   /* definition and creation of affichage */
189   osMutexDef(affichage);
190   affichageHandle = osMutexCreate(osMutex(affichage));
```

Figure 13: Initialisation du Mutex

On l'utilise enfin, dans une tâche dont on veut qu'elle se serve du mutex le code suivant :

```
for(;;){
    xSemaphoreTake(affichageHandle,portMAX_DELAY);
    ...
    ...
    xSemaphoreGive(affichageHandle);
    vTaskDelayUntil( &xLastWakeTime, xFrequency );
}
```

en faisant attention à laisser la mise en veille après, sinon ça ne marche plus très bien.

1.5 Configuration Queue

Nous avons vus précédemment que les Queues se trouvaient au même endroit que les Tasks **(Figure 7)**, on obtient ainsi en ajoutant une Queue :

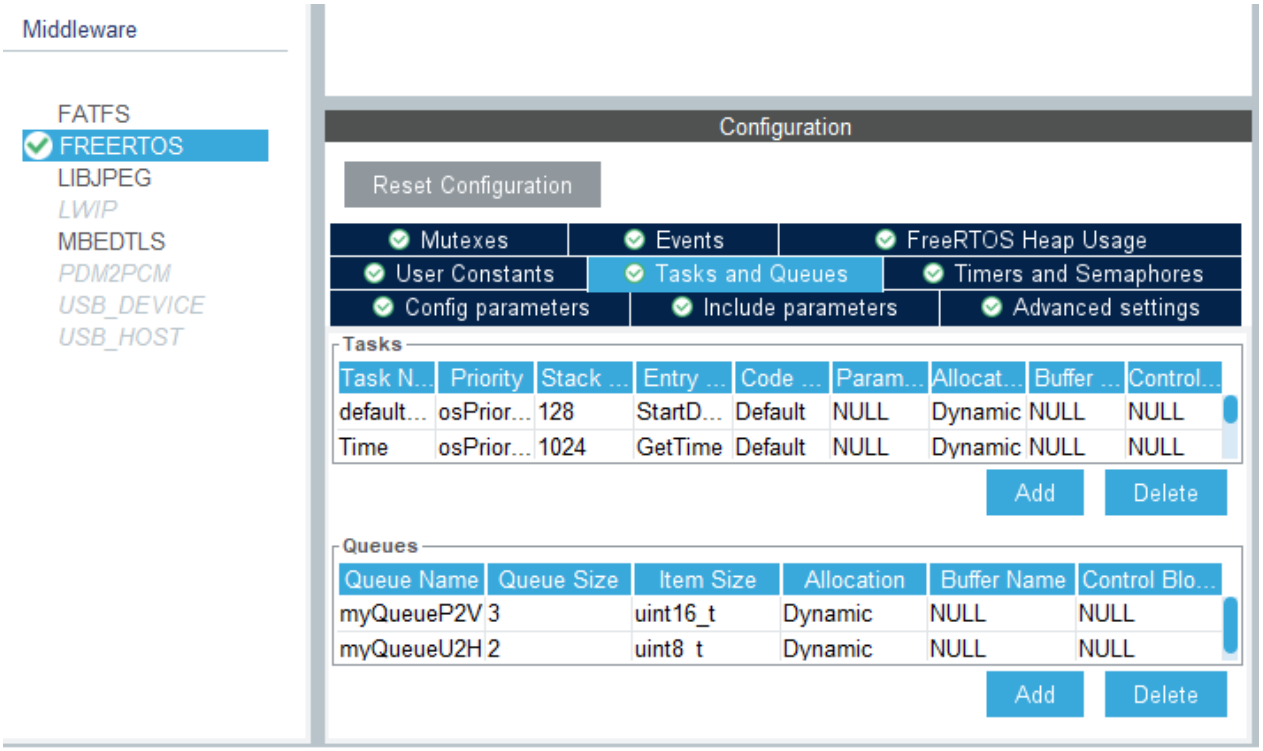

Figure 14: Création Queue

En cliquant sur Add, on obtient les paramètres suivants :

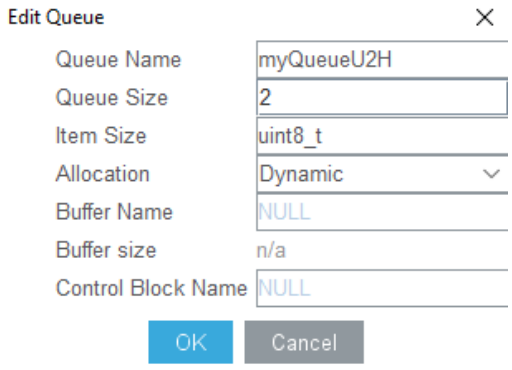

Figure 15: Config Queue

On peut indiquer la taille de la Queue et le type de données.

On obtient alors :

```
419  /* Create the queue(s) */
420  /* definition and creation of myQueueP2V */
421  osMessageQDef(myQueueP2V, 3, uint16_t);
422  myQueueP2VHandle = osMessageCreate(osMessageQ(myQueueP2V), NULL);
423
424  /* definition and creation of myQueueU2H */
425  osMessageQDef(myQueueU2H, 2, uint8_t);
426  myQueueU2HHandle = osMessageCreate(osMessageQ(myQueueU2H), NULL);
```

Figure 16: Création code Queue

A partir de cette Queue, on peut réaliser plusieurs actions :

- Envoyer des messages

```
1  uint16_t Message[3];
2  ...
3  xQueueSend(myQueueP2VHandle, &Message, 0);
```

- Envoyer des messages depuis une fonction d'interruption

```
1  uint8_t Message[2];
2  ...
3  xQueueSendFromISR(myQueueU2HHandle, &Message, 0);
```

- D'attendre la réception d'un message dans une tâche. Elle sera bloquée le temps de réception ou du Timeout

```
1  uint16_t Message[3];
2  ...
3  xQueueReceive(myQueueP2VHandle, &Message, TickType_t Timeout);
```

1.6 Interruptions

On commence par activer l'interruption dans le fichier .ioc, Connectivity>USART1>NVIC Settings :

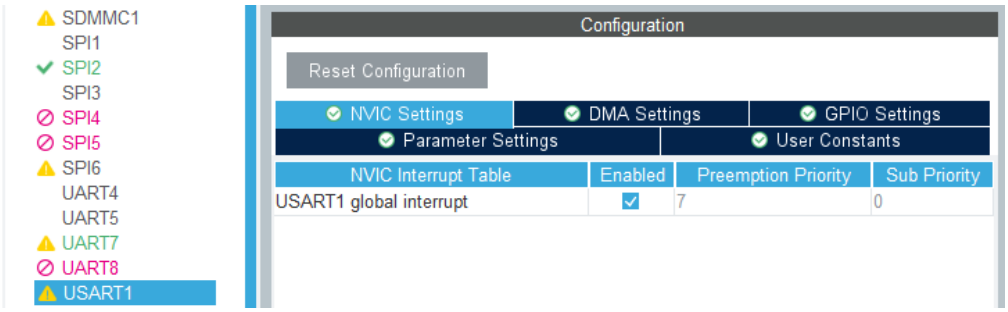

Figure 17: Activation interruption

On peut modifier la priorité de préemption (gérée par l'ordonnanceur) dans System Core>NVIC.

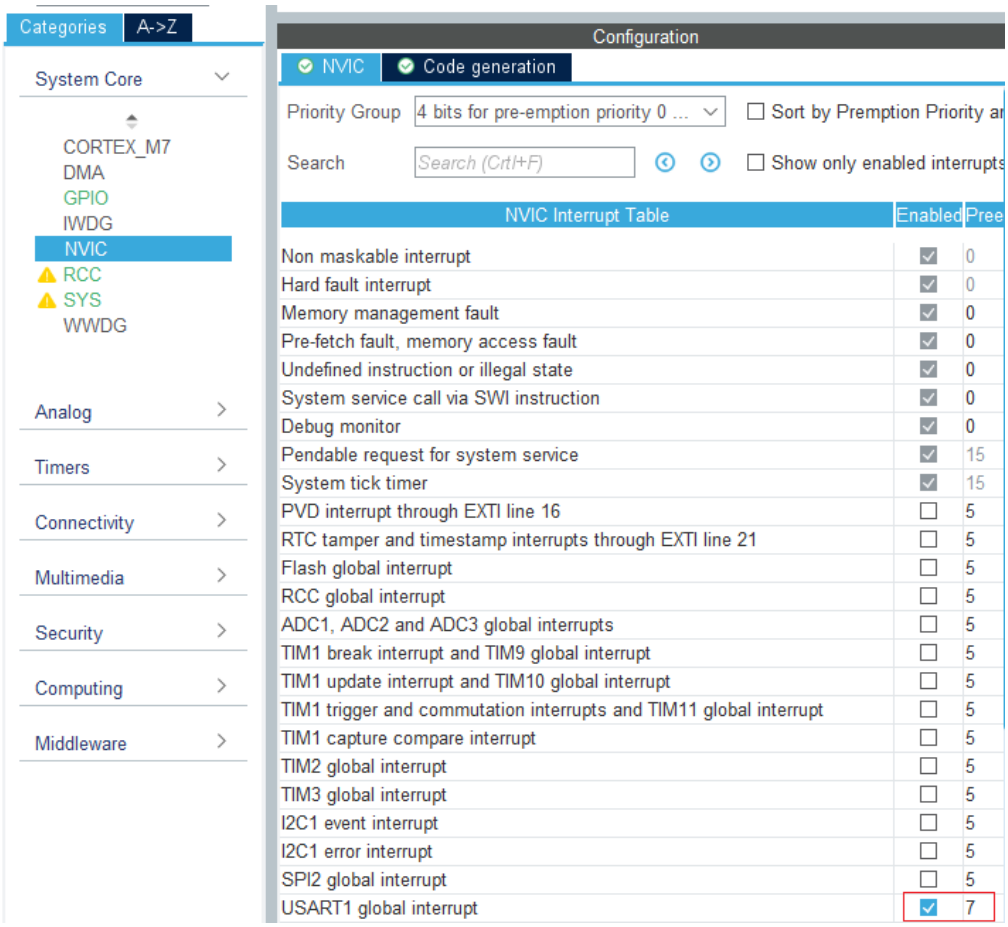

Figure 18: Gestion de la priorité

Dans le cas d'une interruption sur communication série, il va falloir créer une variable globale dans la zone de code 0 du main.c (Autour de la ligne 120). C'est ici que vous créerez vos variables globale en général.

```
1 /* USER CODE BEGIN 0 */
2 uint8_t rxbuffer[10];
3 /* USER CODE END 0 */
```

Il faudra activer l'interruption de réception d'un ou plusieurs caractères dans la zone de code 2 dans la fonction main(void). (Autour de la ligne 180)

```
/* USER CODE BEGIN 2 */
// Initialisation précédentes avec l'afficheur
HAL_UART_Receive_IT(&huart1,rxbuffer,1);
/* USER CODE END 2 */
```

On terminera par créer la fonction CallBack qui sera exécutée à chaque réception du nombre de caractère indiqué dans HAL_UART_Receive_IT(..) dans la zone de code 4. (Autour de la ligne 1660)

```
/* USER CODE BEGIN 4 */
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
{
    uint8_t Message[2];
    if(rxbuffer[0]=='a') HAL_GPIO_WritePin(LED_1_GPIO_Port, LED_1_Pin,1);
    if(rxbuffer[0]=='e') HAL_GPIO_WritePin(LED_1_GPIO_Port, LED_1_Pin,0);

    HAL_UART_Receive_IT(&huart1,rxbuffer,1);
}
/* USER CODE END 4 */
```

1.7 Zone critique

Pour éviter qu'une courte section de code ne soit interrompue, on peut bloquer l'ordonnanceur avec :

```
taskENTER_CRITICAL();
...
//Code à ne pas interrompre
...
taskEXIT_CRITICAL();
```

2 Fiches basées sur le cours de M. Juton

Vocabulaire

1. Tâche : Task
2. Système d'exploitation : Operating system (OS)
3. Pile : Stack
4. Tas : Heap
5. Sémaphore : Semaphore
6. Temps réel dur : Hard Real Time
7. Temps réel souple : Soft Real Time
8. Ordonnanceur : Scheduler

2.1 Enjeux et problématique des OS temps réel

Un OS permet de faciliter la programmation Multitâche et le travail collaboratif (Une personne/équipe par tâche simplifie la mise en commun du programme sans pour autant passer par un partage tour par tour ou un partage par Contrôle de version par exemple Github).

Un OS temps réel va, lui, permettre de respecter les échéances de tâches critiques (Imposer des temps de fonctionnement), garantir le déterminisme fonctionnel et temporel (L'utilisateur contrôle tous les paramètres de ses tâches). On peut par exemple retrouver un RTOS dans une voiture (Ordinateur de bord) qui doit gérer plein de micro-contrôleurs dédiés eux-mêmes à plein de périphériques différents.

On distingue le temps réel strict ou dur pour des systèmes critiques (les échéances doivent êtres respectées) et le temps réel souple ou mou (on peut tolérer un retard limité). **Un système est dit critique si son fonctionnement met en jeu la vie de personnes. (Une voiture, un avion ou Aurore par exemple)** Le systèmes non critiques peuvent souvent se suffirent de temps réel souple (Équipement réseau du Crous par exemple).

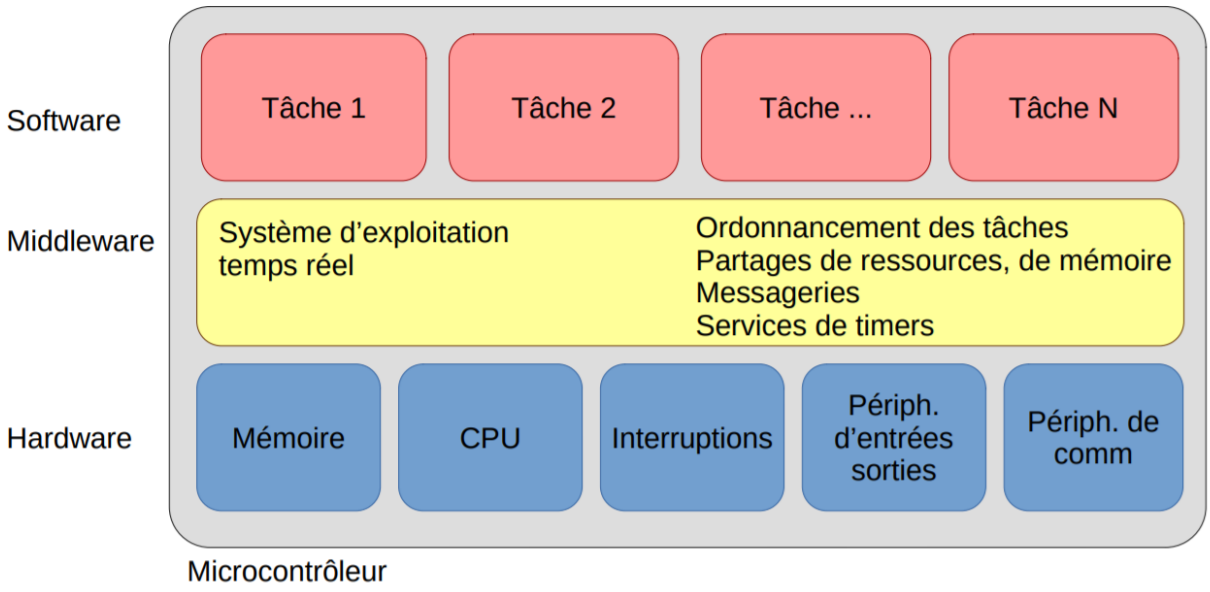

Figure 19: Service d'un OS temps réel

Nous utiliserons l'OS temps réel Free RTOS, d'une part car il est gratuit mais également car il est compatible avec l'embarqué (très faible empreinte mémoire, entre 4 et 9 Ko) dont l'architecture ARM Cortex M4.

On distingue 3 types de systèmes temps réels :

- **Les systèmes transformationnels :** Systèmes transformant des données d'entrée en données de sortie à leur propre rythme, sans interaction avec leur environnement.
- **Les systèmes interactifs :** Systèmes avec souvent une interface graphique, devant répondre rapidement aux demandes d'un utilisateur.
- **Les systèmes réactifs :** Systèmes contrôlant des procédés. Un procédé est un système physique à contrôler, possédant son propre environnement. Donc avec une importance du temps de réaction.

2.2 Ordonnancement

Qu'est-ce que l'ordonnancement ?

L'odonnancement est un système de file d'attente basé sur les Sémaphores et les Mutex. Il est basé sur le modèle Round-Robin avec gestion des priorités.

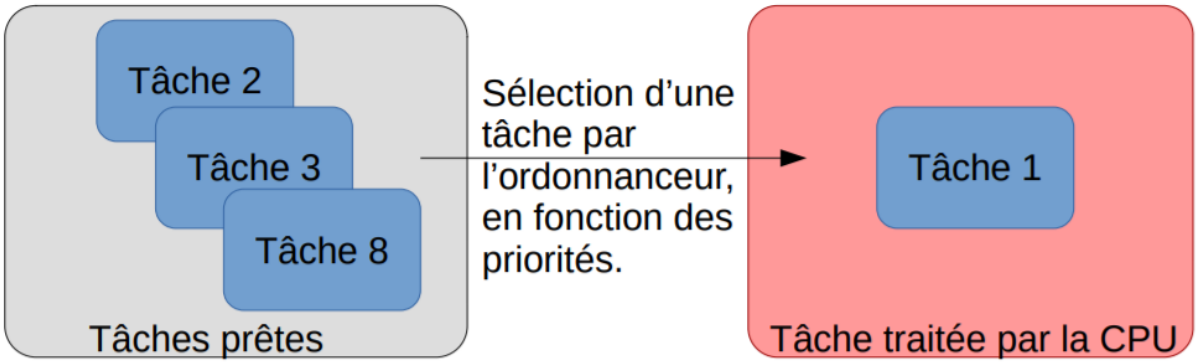

Figure 20: Ordonnancement des tâches

On distingue 3 types de tâches :

- **Tâche périodique :** Tâche qui demande à être exécutée régulièrement, en étant réveillée périodiquement par l'horloge du calculateur. (Exemple : Tâche périodique d'une boucle d'asservissement)
- **Tâche sporadique :** Tâche qui demande à être exécutée sur événement, à des instants non prévisibles. Ses réveils sont séparés par un délai minimal, que l'on appelle période. (Exemple : Tâche liée à la réception d'un message via une communication régulée)
- **Tâche apériodique :** Tâche qui demande à être exécutée sur événement, à des instants non prévisibles. Ses réveils ne sont pas séparés par un délai minimal. On ne peut prouver leur ordonnancement. (Exemple : Tâche liée à une alarme)

Modélisation des tâches

Soit un ensemble de N tâches à échéance contrainte et à départ simultané, $\forall tache$ notée t_i, $i \in [1,\ N]$, , on peut noter :

- C_i : Temps de calcul (WCET Worst-Case Execution Time, dans le pire des cas)
- S_i : Date de démarrage

- r_i : Date de réveil
- d_i : Date d'échéance
- D_i : Date d'échéance relative
- T_i : Période de réveil
- U_i : $\frac{C_i}{T_i}$ utilisation du processeur

Ainsi que la condition nécessaire à l'ordonnancement **pour les algorithmes vus en classe** :

$$\boxed{\sum_{i=1}^{N} U_i \leq 1} \tag{1}$$

Sinon, il y a famine.

Dans notre cas, l'ordonnancement est préemptif, c'est-à-dire qu'une tâche en cours d'exécution peut être interrompue si une autre tâche est prête. un ordonnancement non-préemptif bloque l'ordonnancement jusqu'à ce que la tâche termine son cycle.

Validation temporelle

On valide l'ordonnancement par l'essai de l'ordonnancement jusqu'à l'hyper-période, soit la période égale au ppcm des périodes des tâches du système.

Avec la condition suffisante de Liu et Layland **pour certains algorithmes** :

$$\boxed{\sum_{i=1}^{N} U_i \leq N.(2^{1/N-1})} \tag{2}$$

$$\lim_{N \to \infty} N.(2^{1/N-1}) = ln(2) \simeq 0.69$$

Algorithme d'ordonnancement

Il dicte à l'ordonnanceur quelle tâche exécuter parmi les tâches prêtes avec un critère de choix dépendant de :

- L'importance de la tâche (caractéristique intrinsèque de la tâche)
- Son évolution dynamique (son échéance)

Types de priorités :

- **Priorités fixes aux tâches :** La tâche a une priorité qui ne varie pas, définie avant l'exécution du sytème(sauf en cas d'accès à une ressource, voir fiche services). C'est le mode utilisé par FreeRTOS
- **Priorités fixes aux travaux :** La tâche a une priorité qui peut varier d'un travail à un autre (Travail : Un cycle d'exécution de la tâche)
- **Priorités dynamiques :** A tout moment, l'ordonnanceur peut recalculer les priorités des tâches. Ces algorithmes ne sont pas prédictibles (risque d'anomalie d'ordonnancement).

Le tourniquet (Round-Robin) attribue un quota de temps identique à chaque tâche prête. Chacune des tâches est alors exécutée pendant ce quota de temps, à tour de rôle et selon le niveau de priorité.

2.3 Algorithmes d'ordonnancement

On utilisera pour les exemples des algorithmes la situation suivante :

Tâche	Début S_i	Calcul C_i	Echéance D_i	Période T_i
t_1	0	1	3	3
t_2	0	1	4	4
t_3	0	2	6	6

Rate monotonic (RM)

Pour les tâches à échéance implicite (échéance relative égale à la période), l'algorithme consiste à affecter aux tâches une priorité inversement proportionnelle à leurs périodes.

Exemple :

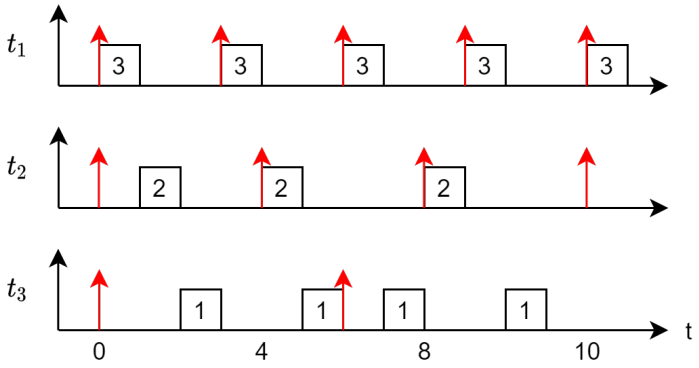

Figure 21: Algorithme RM

La condition suffisante pour l'ordonnançabilité de RM est la condition **(2)**. Si les périodes des tâches sont harmoniques, alors la condition devient nécessaire et suffisante **(1)**.

Deadline Monotonic (DM)

Pour les tâches à échéance contrainte (échéance relative inférieure ou égale à la période), DM consiste à affecter aux tâches une priorité inversement proportionnelle à leur échéance relative. Dans le cas particulier où les tâches sont à échéances implicites (échéances relatives ordonnées comme les périodes) alors DM est identique à RM.

Exemple :

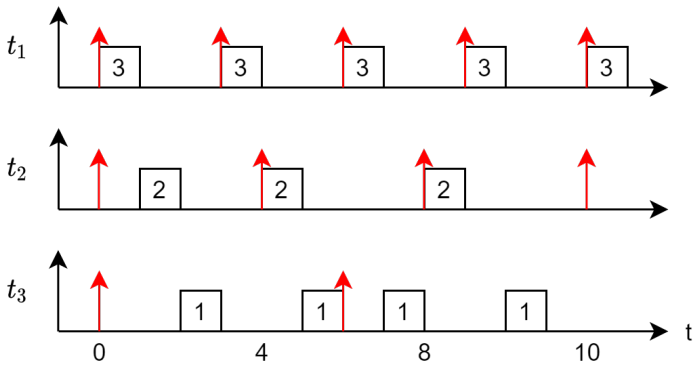

Figure 22: Algorithme DM = RM

La condition suffisante pour DM est similaire à RM, **(2)**.

Earliest Deadline First (EDF)

EDF consiste à donner une priorité à une instance de tâche proportionnelle à son urgence. L'urgence d'une instance est connue à son réveil, soit $E_i(t) = D_i - t$, tel que $E_i(t)$ est l'échéance de la tâche i à l'instant t. Plus l'échéance est grande moins la tâche est prioritaire.

Exemple :

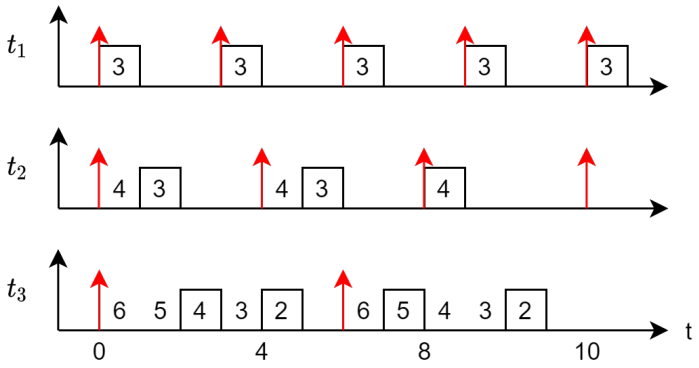

Figure 23: Algorithme EDF

La condition nécessaire et suffisante de EDF est **(1)** pour des tâches périodiques ou sporadiques à **échéance implicite**.

Least Laxity First (LLF)

LLF est basé sur un quantum de temps. A chaque quantum, les priorités des tâches sont calculées sur la laxité. C'est-à-dire : $Latence_i(t) = D_i - t$, $Laxite_i(t) = Latence_i(t) - c_i(t)$ avec $c_i(t)$ le temps de calcul restant à l'instant t de la tâche i.

Exemple :

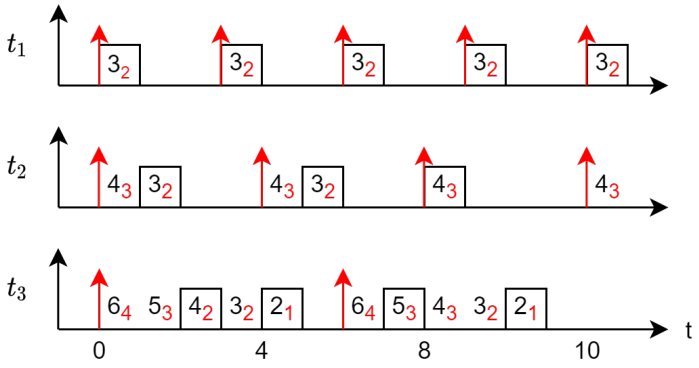

Figure 24: Algorithme LLF

On a alors des priorités dynamiques. LLF est optimal pour ordonnancer des systèmes de tâches indépendantes.

2.4 Le partage de ressources

Mutex

Problématique : Si deux tâches utilisent une même ressource, des incohérences peuvent se produire, surtout lorsqu'une tâche à une période beaucoup plus courte que la seconde (exemple de l'écran TP2).

Pour résoudre le problème, on utilise un procédé d'exclusion mutuelle, c'est-à-dire qu'une tâche bloque l'utilisation de la ressource en question tant qu'elle n'as pas terminer. Ce procédé est appelé MUTEX ou sémaphore binaire (0—1) voir **1.4**.

Inversion de priorité

Le risque avec les mutex (sémaphores en général) est qu'il bloque une tâche plus prioritaire plus longtemps que prévu avec l'exemple suivant :

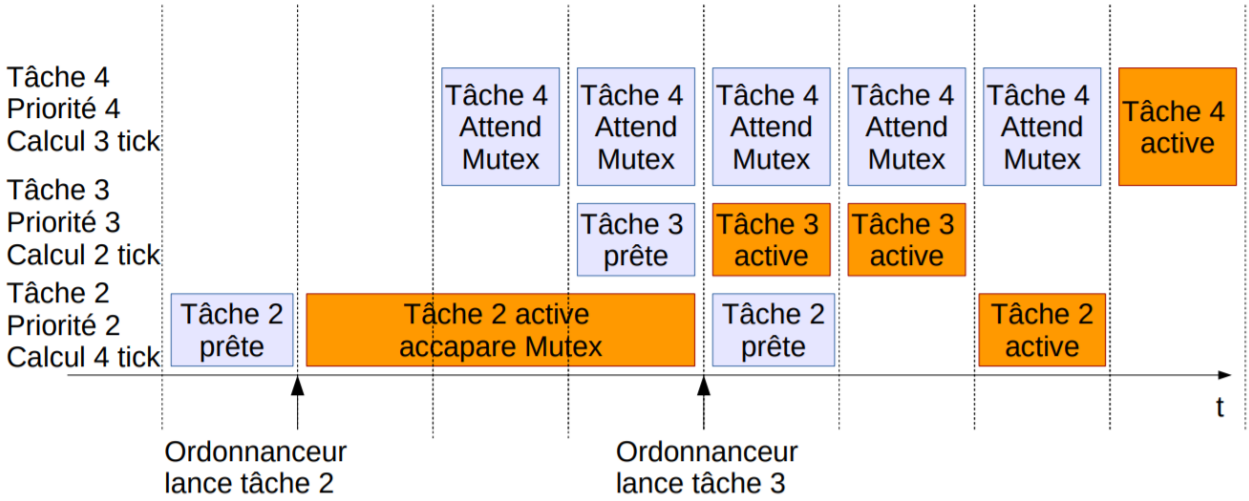

Figure 25: Risque des sémaphores

où la tâche 4 est bloquée par la tâche 2 qui elle même est retardée par la tâche 3 à un certain moment. Une solution à ce problème est temporairement d'augmenter la priorité de la tâche utilisant le Mutex (ou sémaphore) au niveau de la priorité la plus haute des tâches utilisant ce même mutex.

D'autres problèmes peuvent survenir mais ils surviendront lorsqu'on utilise plusieurs mutex entre les mêmes tâches.

2.5 Services

Messagerie

Nous voulons communiquer entre deux tâches mais les variables ne sont pas partagées ! Comment faire ? On utilise une messagerie de type Queue, il existe également les types Blackboard, Mailbox et Pipe dont on parlera très rapidement (au cas où vous en ayez besoin).

Queue : Les messages sont placés les uns derrière les autres dans une queue. Les tâches peuvent ainsi venir lire dans la queue dans un mode first-in-first-out FIFO. Les limitations de ce type seront au niveau de la taille des données à transmettre.

Blackboard : Les nouveaux messages écrasent les anciens. On utilise une zone mémoire partagée par toutes les tâches (c'est une sorte de variable globale mais il ne faut pas le dire parce qu'on n'aime pas ça).

Mailbox : C'est un peu comme une Queue sauf que l'on a le contrôle sur la taille de données maximale ainsi que définir des priorités (intra-code) pour des zones de la mailbox.

Pipe : Pareil qu'une Queue mais à message d'une taille variable !

Gestion des interruptions

Les interruptions matérielles gardent leurs intérêts. Elles sont prioritaires sur le fonctionnement normal des tâches (priorité par rapport à l'ordonnanceur est à choisir). Les fonctions utilisées dans les interruptions par FreeRTOS utilisent un suffixe FromISR pour communiquer. Par exemple :

```
1 xQueueSendFromISR(myQueueU2HHandly, &Message, 0);
```

Pour la validation temporelle, soit les interruptions lancent des tâches apériodiques non prioritaires, soit on crée un tâche périodique chargée de la gestion des interruptions, quand elles arrivent.

Section critique, création et destruction de tâches

Pour éviter qu'une courte section de code ne soit interrompue, on peut bloquer l'ordonnanceur avec :

```
1 taskENTER_CRITICAL();
2 ...
3 //Code à ne pas interrompre
4 ...
5 taskEXIT_CRITICAL();
```

Une tâche peut créer, détruire, suspendre et relancer d'autre tâches avec respectivement :

```
1 XtaskCreate()
2 VtaskDelete()
3 VtaskSuspend()
4 VtaskResume()
```

Bibliographie

- **Cours RTOS M. Juton :** Ecampus>RTOS
- **Free RTOS :** https://www.freertos.org/a00106.html
- **Ordonnancement temps réel :** https://hal.archives-ouvertes.fr/hal-00662741/document#:~:text=Chaque%20t%C3%A2che%20a%20une%20certaine,la%20vie%20du%20syst%60eme.&text=L'ordonnanceur%20Rate%20Monotonic%20(RM,est%20fonction%20de%20leur%20p%C3%A9riode.
- **Optimisation consommation :** https://hal.archives-ouvertes.fr/tel-01332440/document