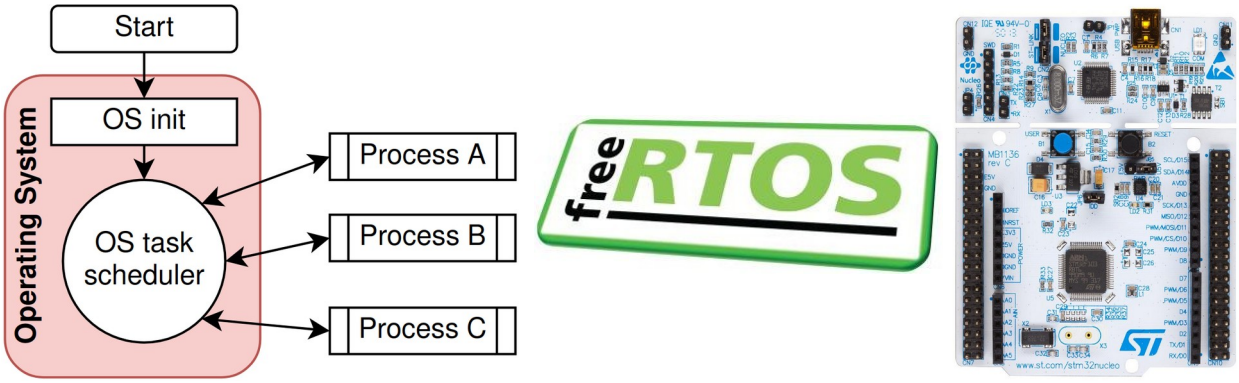
Systèmes Temps-Réel

Support de Travaux Pratiques



ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

2A SATE / ECSE

hugo.descoubes@ensicaen.fr

dimitri.boudier@ensicaen.fr

Contacts

Dimitri Boudier – CM, TP, Responsable du module en FISA
dimitri.boudier@ensicaen.fr

Hugo Descoubes – CM, TP, Responsable du module en FISE
hugo.descoubes@ensicaen.fr

Oumaima Assou – TP en FISA
oumaima.assou231@ensicaen.fr

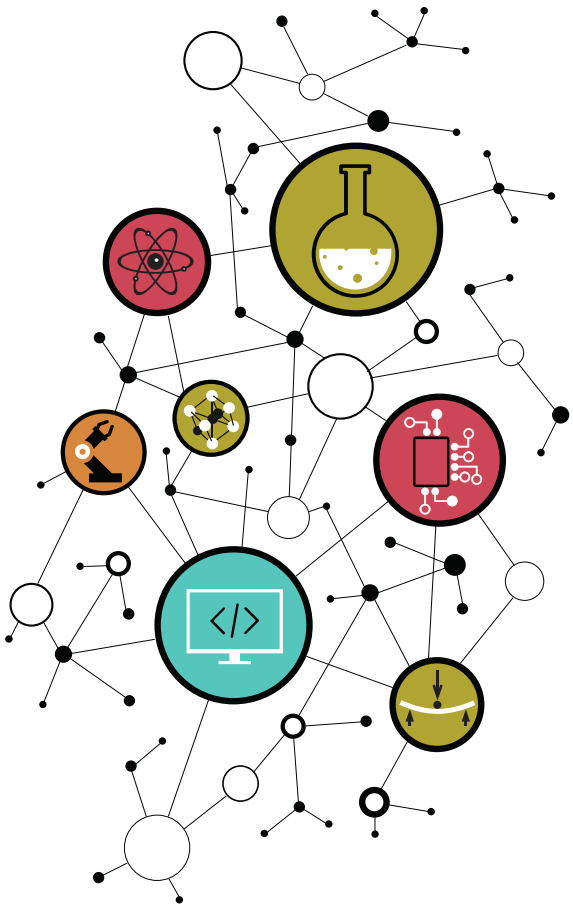
Ressources

Toutes les ressources (supports CM, TP et outils) sont sur la page Moodle du cours :
https://foad.ensicaen.fr/course/view.php?id=840

Sommaire

PARTIE 1

ENVIRONNEMENT DE TRAVAIL



I. Quels outils ?

Commençons par présenter les outils matériels et logiciels utilisés pendant ces séances de travaux pratiques.

I.1. Système d'exploitation

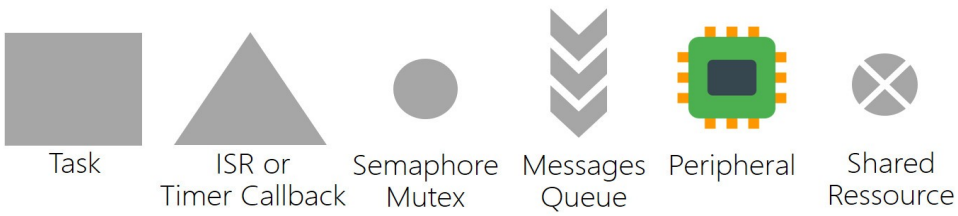
Cet enseignement est avant tout dédié à la compréhension des services logiciels proposés par un système d'exploitation, notamment ceux dits **temps-réel** (déterministes).

Les concepts seront illustrés sur la technologie **FreeRTOS**[1], solution *open source* (licence MIT) et leader actuel sur le marché des **RTOS (*Real Time Operating System*)** dans l'embarqué.

I.2. Représentation graphique

En parallèle, l'utilisation de ces outils amène à définir une architecture logicielle de sorte à répondre à un cahier des charges. Bien évidemment, ceci doit être fait de manière rigoureuse et claire. Cependant, il n'existe pas de consensus (et encore moins de norme) au niveau international ou inter-professionnels pour décrire une architecture logicielle construite autour d'un RTOS.

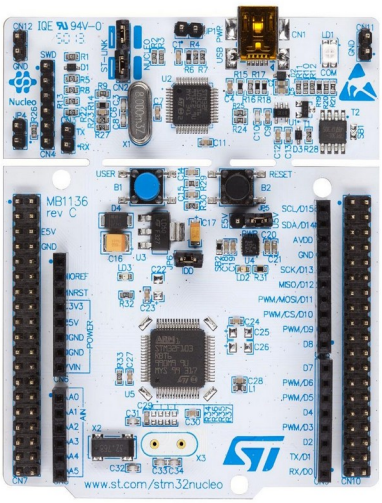
Nous utiliserons donc un formalisme maison, présenté ci-dessous pour illustrer nos propos et/ou décrire un cahier des charges. Bien que celui-ci soit propre à cet enseignement, il ne faut pas oublier qu'une des missions premières d'un ingénieur est de savoir communiquer efficacement (c'est-à-dire simplement et clairement).



1 https://www.freertos.org/

I.3. Matériel

La trame de TP se fera sur **micro-contrôleur STM32**. C'est donc l'occasion de découvrir un processeur de la famille **ARM**, plus précisément de la gamme Cortex-M, à travers une solution de **STMicroelectonics**.



Nous travaillerons avec une **carte d'évaluation SMT32 Nucleo-L073RZ**[2] ou **NUCLEO-L476RG**[3] (selon les stocks disponibles). Cette carte fait partie de la famille STM Nucleo-64, principale famille de kit d'évaluation de ST pour ses processeurs ARM.

Chaque carte d'évaluation comporte notamment le micro-contrôleur STM32 cible ainsi qu'une sonde de programmation et de *debug*. De plus, elle embarque un bouton poussoir, une LED, ainsi que des connecteurs ST morpho (76 broches) et Arduino. Elle est équipée du STLink VCP (*Virtual COM Port*), permettant une communication série via le port USB. Le tout permet ainsi de développer rapidement des applications embarquées.

I.4. Environnement de développement

Pour nos développements sur STM32, nous utiliserons la suite logicielle proposée par STMicroelectonics : **STM32Cube**[4]. Il s'agit d'un écosystème complet qui propose plusieurs logiciels, même si nous nous concentrerons sur les deux principaux :

- **STM32CubeIDE**, un IDE ;
- **STM32CubeMX**, un outil de configuration et de génération de code.

L'environnement de développement intégré (qu'on appellera désormais **IDE** pour *Integrated Development Environment*) s'appelle donc **STM32CubeIDE**. Il est construit à partir de l'IDE Eclipse, *aka* le plus gros projet d'IDE libre et multi-plateforme. Son fonctionnement s'articule autour de *perspectives* (des vues) correspondant à des usages spécifiques (par ex : *edit*, *debug*).

Inclus dans l'IDE, **STM32CubeMX** est un outil graphique qui permet de configurer le MCU et ses périphériques pour une utilisation en quelques minutes. Hors du TP, l'objectif de ce genre d'outils est de réduire le *Time-to-market* (temps de développement) des solutions logicielles embarquées. L'équivalent Microchip s'appelle MCC (*MPLAB Code Configurator*).

STM32Cube étant *cross-platform*, les TP peuvent être réalisés sous Windows et/ou Linux. Pour plus d'informations sur l'utilisation de ces logiciels (version à télécharger notamment), se référer à l'annexe dans l'archive de TP.

2 https://www.st.com/en/evaluation-tools/nucleo-l073rz.html
3 https://www.st.com/en/evaluation-tools/nucleo-l476rg.html
4 https://www.st.com/en/ecosystems/stm32cube.html

II. Présentation de l'archive de TP

L'archive de TP suit l'arborescence suivante :

- `tp_rtos/` : la racine de l'archive de TP
  - `documentations/` : documentations techniques (cartes NUCLEO, STM32CubeIDE)
  - `tutorials/` : tutoriels pour l'utilisation des outils
  - `sources/` : fichiers fournis pour les exercices
  - `workspace/` : dossier qui contiendra les projets STM32CubeIDE

Idéalement, cette archive de TP doit être stockée sur votre disque dur.

> Le chemin complet ne doit comporter ni accent, ni espace, ni caractère spécial.
>
> Si vous travaillez sur les sessions Windows de l'école, tous vos projets devront être créés dans un sous-répertoire du disque `Z:/`.
>
> Aucune erreur ne sera pardonnée par l'IDE !

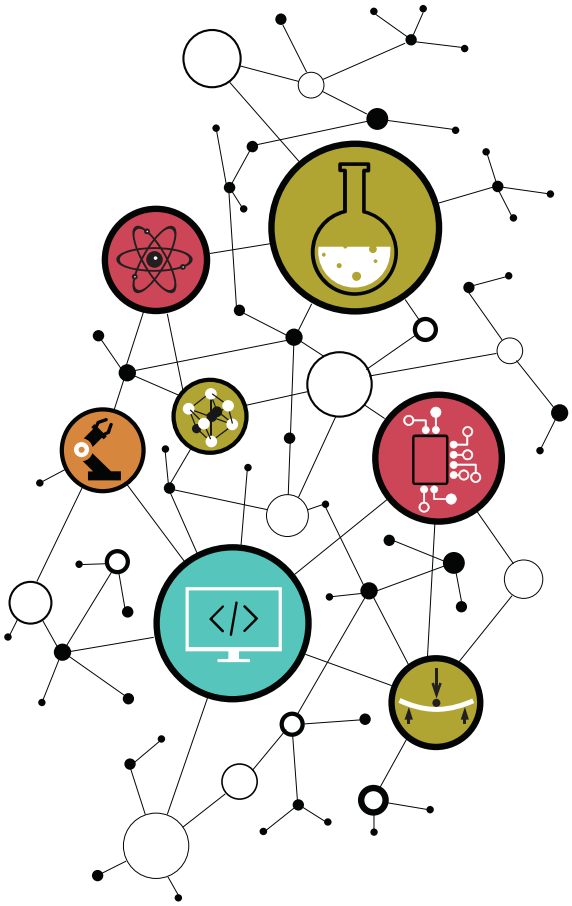
Dans la suite de ce document, il sera souvent fait référence aux annexes. Celles-ci sont stockées dans le répertoire `tutorials/`, qu'il convient d'aller consulter en cas de doute. Vous y trouverez des documents sur les cartes NUCLEO, l'IDE, FreeRTOS et des logiciels de terminal série.

À la racine de l'archive de TP se trouvent deux PDF :

- `introduction_aux_stm32.pdf`, à faire si vous ne l'avez pas déjà fait ;
- `tp_rtos_stm32.pdf` (ce fichier), à faire si vous savez manipuler la NUCLEO et l'IDE.

PARTIE 2

STRATÉGIES D'ORDONNANCEMENT



I. Tâches et ordonnanceur

La partie centrale d'un système d'exploitation est son **ordonnanceur (ou *scheduler*)**, qui fait d'ailleurs à proprement parler partie du noyau (ou *kernel*) de l'OS. C'est en effet l'ordonnanceur qui est chargé de répartir le temps d'utilisation du CPU entre les différentes **tâches (ou *process*, ou *thread*)** qui évoluent en parallèle (tout du moins en apparence). Pour cela le *scheduler* doit faire des choix en fonction de **l'état des tâches**, ce qui reflète les besoins d'une tâche à un instant donné. Comme d'autres systèmes d'exploitation temps-réel, FreeRTOS propose pour le *scheduler* un mode **coopératif** et un mode **préemptif**.

En mode **coopératif**, l'ordonnanceur n'a pas le pouvoir absolu sur les tâches : les tâches sont programmées de sorte à coopérer de manière explicite, avec seulement une faible participation de l'ordonnanceur. Concrètement, une tâche en cours d'exécution le restera jusqu'à ce qu'elle quitte d'elle-même l'état *running*. L'ordonnanceur est alors appelé pour choisir la prochaine tâche à exécuter, mais ne fait plus rien tant que la nouvelle tâche reste à l'état *running*. Le mode coopératif est très peu utilisé du fait des problèmes de robustesse et de partage du temps de CPU, ce qui sera d'ailleurs illustré dans le premier exercice de cette partie.

Le mode **préemptif** quant à lui donne le droit à l'ordonnanceur d'interrompre n'importe quelle tâche à intervalles réguliers. De cette manière, le *scheduler* va périodiquement décider quelle tâche sera exécutée (en fonction des tâches à l'état *ready*/*running* et de leur priorité). Ce mode est de loin le plus répandu, car il apporte une grande flexibilité et des performances temps-réel intéressantes. Ce mode est étudié en deuxième partie de ce chapitre.

Les objectifs de cette partie sont les suivants :

- comprendre les états d'une tâche ;
- comprendre l'interaction de l'ordonnanceur avec les tâches en fonction de leur état ;
- prendre en main les principales fonctions de l'API de FreeRTOS.

II. Travail préliminaire

Pour répondre à ces questions, aidez-vous de la documentation de FreeRTOS (*Developper Docs*[5] et *API Reference*[6]).

NB : la page suivante est vierge pour y écrire vos réponses.

1. Quels états peut prendre une tâche sous FreeRTOS ?
2. Ces états sont-ils les mêmes quel que soit l'OS ou le RTOS ? Donnez un exemple pour un autre RTOS.
3. Donnez le prototype de la fonction permettant de créer une tâche, et précisez le rôle de ses paramètres.
4. Quelle fonction permet le démarrage de l'ordonnanceur ?
5. Que se passe-t-il sous FreeRTOS lorsqu'aucune des tâches précédemment créées n'est en cours d'exécution (état *running*) ?
6. Qu'est-ce qu'un TCB ? Que trouve-t-on dans le TCB sous FreeRTOS ?
7. Quels sont les inconvénients et avantages d'un OS coopératif (aidez-vous du Web) ?
8. Quels sont les inconvénients et avantages d'un OS préemptif (aidez-vous du Web) ?
9. Quelle macro permet de choisir entre le mode coopératif et le mode préemptif sous FreeRTOS ? Précisez les valeurs associées.

5 https://www.freertos.org/features.html
6 https://www.freertos.org/a00106.html

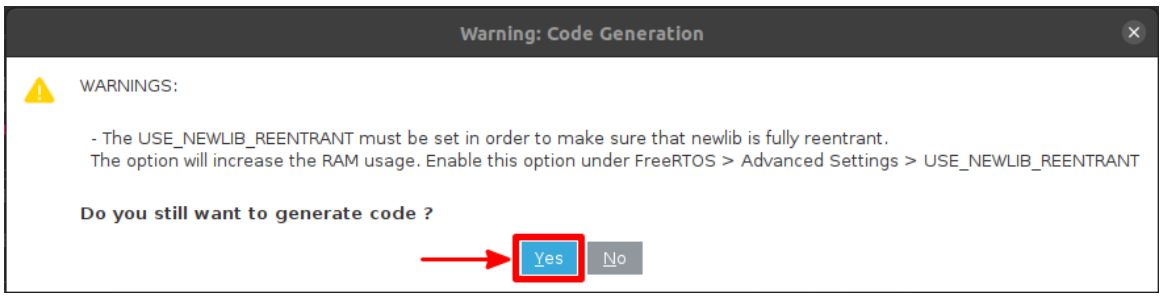
III. Créer un projet intégrant FreeRTOS

Les sources de FreeRTOS sont directement téléchargeables depuis leur site[7]. Il s'agit d'un ensemble de fichiers C, h et asm, ce qui permet au développeur de ne sélectionner que les fonctionnalités essentielles au projet et ainsi limiter la taille du firmware généré. Quant aux questions « quels fichiers intégrer au projet et comment ? », elles sont traitées dans l'annexe dédiée à FreeRTOS (dossier `tp_rtos/tutorials/`).

Cependant, STM32Cube permet nativement d'intégrer les sources de FreeRTOS à un projet, en s'affranchissant des fastidieuses étapes d'importation. Nous suivons donc cette méthode pour ces TP, également détaillée dans cette même annexe.

- En vous aidant des annexes dédiées à CubeIDE et à FreeRTOS, créez un projet répondant aux spécifications suivantes :
  - emplacement : dans le répertoire `workspace/rtos02/` de l'archive de TP.
  - nom du projet : `votrenom_rtos02_scheduler`
  - **ATTENTION** : ne pas configurer les périphériques par défaut !
  - `System Core/GPIO` : broches PC4, PB13 et PB15 en mode `GPIO_Output`, respectivement nommées Task1, Task2 et Task3 (champ `User Label`).
  - `Middleware/FreeRTOS` : Interface CMSIS v1
    - Onglet *Config parameters* : mettre le champ *Memory Allocation* à *Dynamic*, tous les autres champs à *Disable* (sauf *USE_TASK_NOTIFICATIONS*)
    - Onglet *Include parameters* : mettre tous les champs à *Disable*

Une fois la configuration terminée, cliquez sur `Project → Generate Code` (ou l'icône).



Avec ce projet nouvellement créé, nous pouvons noter dans l'explorateur de projet l'apparition du répertoire `Middleware/Third_Party/FreeRTOS/Source/`. Celui-ci contient bien entendu les sources (c, asm, h) de FreeRTOS pour notre MCU. Cette arborescence est détaillée dans l'annexe FreeRTOS.

Le fichier `Core/Inc/FreeRTOSConfig.h` a également été ajouté au projet. Notez qu'il s'agit du **seul fichier** orienté FreeRTOS à **ne pas être situé parmi les sources du RTOS** (dans Middleware/Third_Party/FreeRTOS/), mais bien **présent avec les sources de l'application** (dans Core/). Pour cause, ce fichier contient les paramètres permettant d'ajuster les composants de FreeRTOS pour l'adapter à l'**application**. Pour information, les réglages que vous avez réalisés lors de la configuration du projet (*Config parameters* et *Include parameters*) sont tous traduits sous forme de constantes prédéfinies dans ce fichier. Pour plus d'informations, rendez-vous encore une fois dans l'annexe FreeRTOS.

7 https://www.freertos.org/a00104.html

Enfin, le fichier `Core/Src/main.c` est lui aussi fourni dans le projet, pré-rempli avec quelques instructions que nous nous empressons d'étudier. Observons donc ce que STM32CubeMX a ajouté au `main.c` (en comparaison à un projet sans OS, ou *bare-metal*).

- Que fait la fonction `osThreadDef()` appelée dans la fonction `main()` ?

- Que fait la fonction `osThreadCreate()` ? Quelle autre fonction appelle-t-elle ?

- Que fait la fonction `osKernelStart()` ? Quelle autre fonction appelle-t-elle ?

En l'état actuel, la seule opération réalisée par les fonctions `osThreadCreate()` et `osKernelStart()` est d'appeler les fonctions `vTaskCreate()` et `vTaskStartScheduler()`, qui elles sont des fonctions de l'**API FreeRTOS**. En bref, les deux premières fonctions encapsulent (*wrap*) les deux dernières.

Les deux premières fonctions sont définies par **CMSIS-RTOS**. Il s'agit d'une API développée par la société ARM pour rendre homogène le développement sur processeur Cortex quel que soit l'OS ou RTOS embarqué. FreeRTOS a fait le choix de se rendre compatible avec cette API.

Pour des raisons pédagogiques, nous nous focaliserons uniquement sur FreeRTOS et n'aborderons pas plus l'API CMSIS-RTOS. D'ailleurs, CMSIS-RTOS semble peu adoptée par les industriels, ceux-ci préférant l'API de leur RTOS habituel (source : échange avec deux ingénieurs STMicroelectonics, dont un responsable de développement de l'environnement STM32Cube et un diplômé SATE).

IV. Drivers UART

Toute la partie précédente ne fait que créer un projet générique, il faut ensuite que nous développions l'application propre à notre « produit ». Commençons par les drivers.

Comme il est souvent l'usage, nous utiliserons un périphérique UART (ici USART2) afin de disposer d'une interface de debug de notre application. Cependant pour des raisons particulières[8], nous n'utiliserons pas la HAL pour manipuler ce périphérique USART2, mais un driver « maison ».

- Depuis votre explorateur de fichiers Windows ou Ubuntu, placez-vous dans l'archive de TP, copiez le dossier `<archive-TP>/sources/rtos02/Drivers/ENSI/` et collez-le dans le dossier `<projet>/Drivers/` de votre projet CubeIDE. Les fichiers drivers apparaissent maintenant dans l'explorateur de projet de l'IDE (si besoin, pressez `F5` sur le dossier `Drivers`).

- Ouvrez le fichier `ensi_uart.h` et sélectionnez la carte utilisée en dé-commentant la ligne correspondante.

```
/* Includes ------------------------------------------------------------------*/
//#define USE_NUCLEO_L073RZ             //!< Select here your NUCLEO board
//#define USE_NUCLEO_L476RG             //
//#define USE_NUCLEO_F411RE             //
```

- Dans le `main()`, initialisez le périphérique UART.

```
    /* Initialize all configured peripherals */
    MX_GPIO_Init();
    /* USER CODE BEGIN 2 */
->  ENSI_UART_Init();
->  ENSI_UART_PutString((uint8_t*)"\r\nEt z'est partiiii");
    /* USER CODE END 2 */
```

- Compilez. Une première erreur arrive :

```
ensi_uart.c:13:10: fatal error: ensi_uart.h: No such file or directory
```

- La toolchain ne connaît pas le chemin dans lequel se trouve le fichier d'en-tête. Résolvez cette erreur fournissant à la toolchain le chemin d'inclusion vers ce fichier. Si besoin, consultez le tutoriel consacré à STM32CubeIDE.

- Compilez à nouveau. Résolvez les derniers *warnings*.

```
../Core/Src/main.c: In function 'main':
../Core/Src/main.c:91:3: warning: implicit declaration of function
'ENSI_UART_Init' [-Wimplicit-function-declaration]
   91 |   ENSI_UART_Init();
      |   ^~~~~~~~~~~~~~
```

8 Les fonctions de la HAL bloquent l'accès au périphérique tant qu'un travail déjà engagé n'est pas fini.

V. Ordonnancement en mode coopératif

V.1. Rédaction de l'application

Après avoir intégré les sources FreeRTOS et le driver UART au projet, il est maintenant temps d'écrire notre application.

Pour concentrer en un seul fichier les tâches FreeRTOS propres à l'application et les outils associés, CubeMX a créé un fichier `<project>/Core/Src/freertos.c`. Ce n'est donc pas un fichier de FreeRTOS à proprement parler. En ouvrant et parcourant ce fichier, vous verrez qu'il ne contient aucun code, juste des balises de commentaires afin d'y placer vos tâches.

- Prenez le fichier `<archive-TP>/sources/rtos02/Core/Src/freertos.c` depuis l'archive de TP, et remplacez le fichier `<project>/Core/Src/freertos.c` existant dans votre projet.

Vous pourrez constater que ce nouveau fichier contient du code applicatif. Vous y verrez notamment la définition des fonctions `task1()`, `task2()` et `task3()`.

- Ces fonctions sont déclarées en `static`. Qu'est-ce qu'une ***fonction*** statique en ***langage C*** ?

- Complétez la définition de la fonction `app_init()` pour y créer trois tâches de priorité 1. Ces tâches seront implémentées par les fonctions `task1()`, `task2()` et `task3()`.
- Puis appelez la fonction de création de vos tâches, tout en supprimant la création de la tâche par défaut (fournie par CubeMX).

```
    /* Create the thread(s) */
    /* definition and creation of defaultTask */
    osThreadDef(defaultTask, StartDefaultTask, osPriorityNormal, 0, 128);
    defaultTaskHandle = osThreadCreate(osThread(defaultTask), NULL);

    /* USER CODE BEGIN RTOS_THREADS */
    /* add threads, ... */
->  app_init();
    /* USER CODE END RTOS_THREADS */

    /* Start scheduler */
    osKernelStart();
```

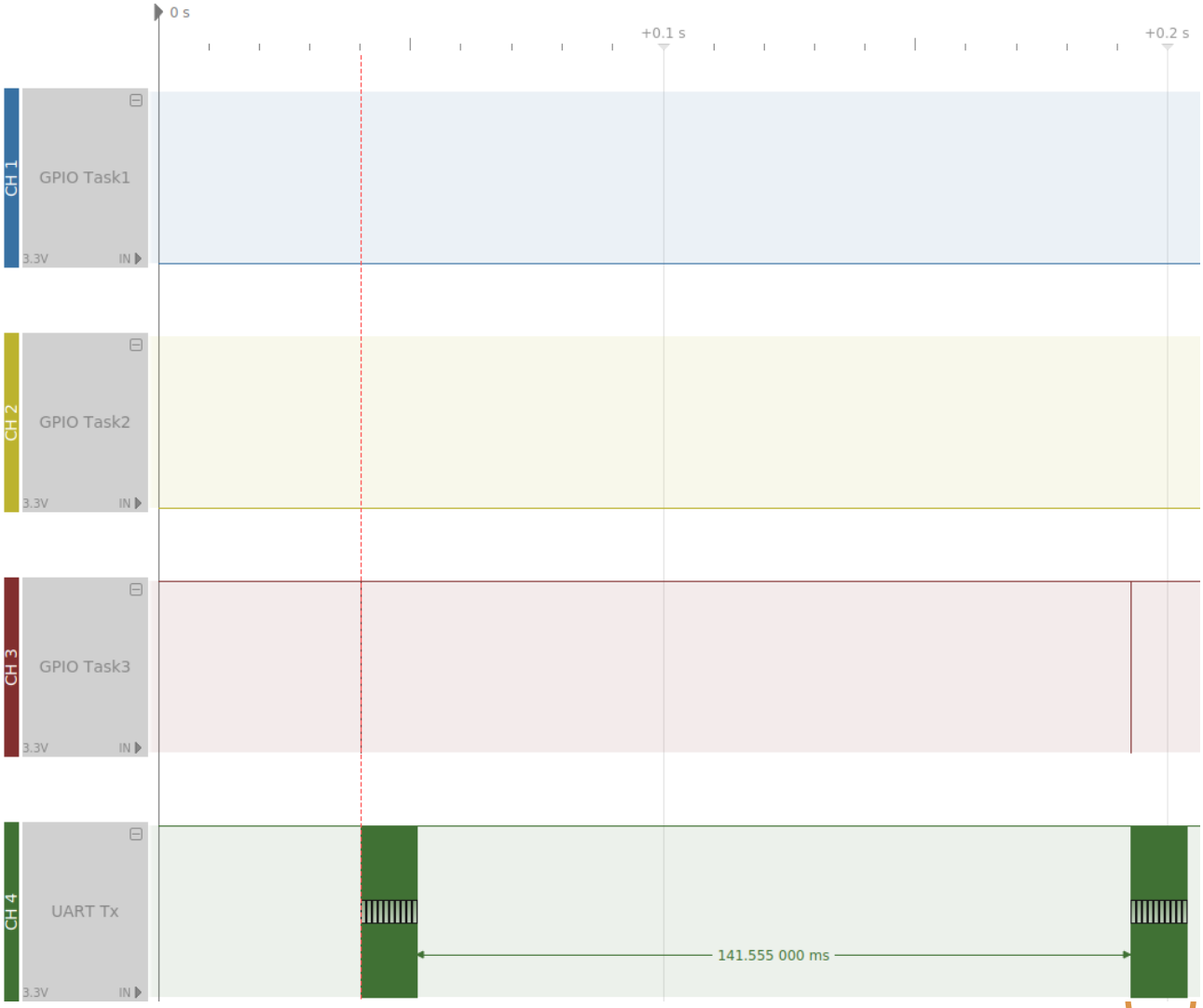
- Compilez, exécutez et analysez en visualisant sur ordinateur les données reçues par liaison série (voire observez les GPIO avec l'analyseur logique).

  Vous pouvez aussi travailler en mode *Debug* (et non en mode *Run*).

Sur la page suivante, vous trouverez les relevés qu'on peut faire avec un analyseur logique et un moniteur série. Ceux-ci vous permettront de tracer un chronogramme, travail d'analyse que vous devrez mener par vous-même dans la suite du TP.

Relevé avec un analyseur logique

La GPIO pilotée par chacune des trois tâches et la broche Tx de l'UART.

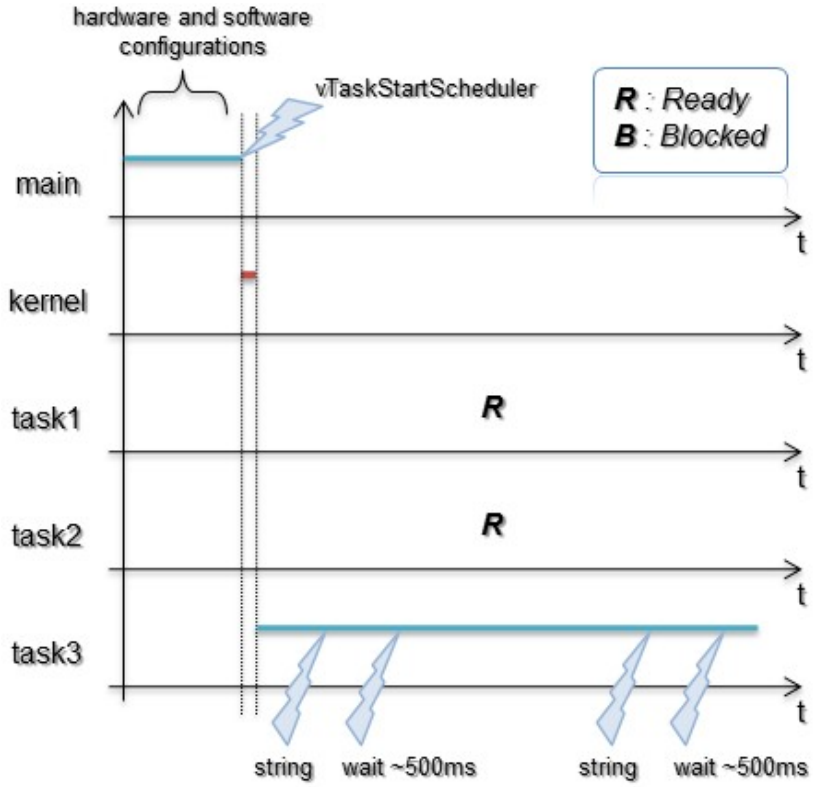


Relevé avec un terminal série observant la broche de transmission de l'UART.

```
Welcome to minicom 2.7.1

OPTIONS: I18n
Compiled on Dec 23 2019, 02:06:26.
Port /dev/ttyACM0, 11:38:59

Press CTRL-A Z for help on special keys


Et z'est partiiii
33333333
33333333
33333333
33333333
33333333
33333333
33333333
33333333
```

V.2. Analyse de la première application

Comme on peut le voir sur le relevé de l'analyseur logique et celui du terminal série, seule la tâche 3 semble être exécutée. On construit alors le chronogramme suivant.



➢ Pourquoi sommes-nous bloqués ?

Tout simplement car en mode coopératif (la macro `configUSE_PREEMPTION` vaut '0'), c'est la tâche qui doit appeler l'ordonnanceur en utilisant une fonction système. L'ordonnanceur ne peut interrompre une tâche de lui-même.

➢ Pourquoi dans la tâche 3 ?

Avec FreeRTOS, au démarrage si plusieurs tâches de même priorité sont susceptibles de prendre la main, c'est la dernière créée qui sera la première à démarrer. Cette politique est différente en fonction de l'OS rencontré.

➢ Dans quel état se trouvent les tâches 1 et 2 ?

En mode debug dans l'IDE, mettez le programme en pause. Puis cliquez sur *Window* → *Show View* → *FreeRTOS* → *FreeRTOS Task List*. Observez.

Les tâches 1 et 2 se trouvent à l'état prêt (*ready*). Elles sont toutes les deux prêtes à prendre la main si la tâche 3 la leur donne (tâche 3 qui elle est à l'état *running*).

Console | Problems | Executables | Debugger Console | Memory | FreeRTOS Task List ×

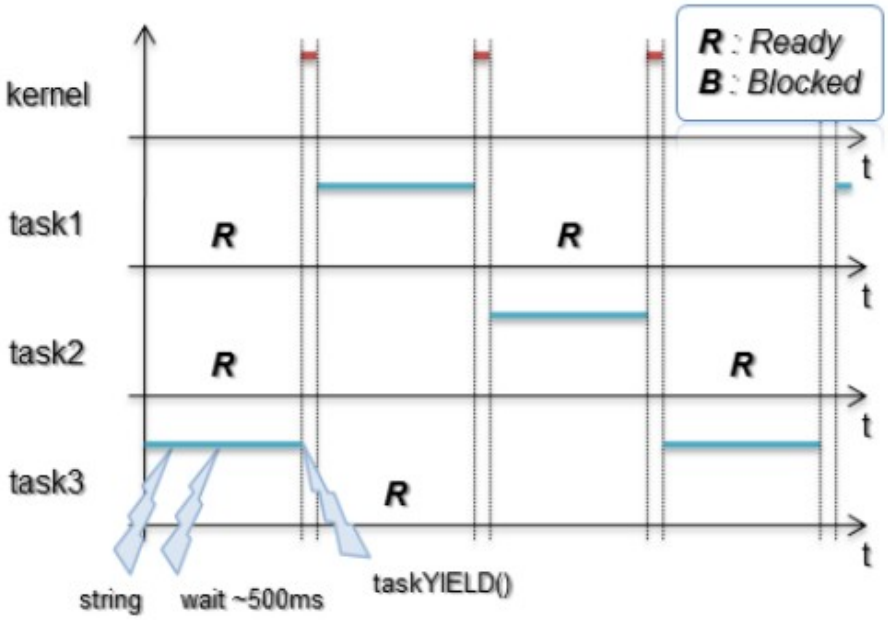
	Name	Priority (Base/Actual)	Start of Stack	Top of Stack	State
	IDLE	N/A/0	0x20000b20	0x20000cd4 <ucHeap+2876>	READY
	Tache 1	N/A/1	0x20000400	0x200005b4 <ucHeap+1052>	READY
	Tache 2	N/A/1	0x20000660	0x20000814 <ucHeap+1660>	READY
→	Tache 3	N/A/1	0x200008c0	0x20000a74 <ucHeap+2268>	RUNNING

V.3. Commutation des tâches

Forçons maintenant les tâches à donner la main en appelant l'ordonnanceur. Dans chaque fonction de tâche, appelez la fonction `taskYIELD()`[9] à la suite de l'extinction de la GPIO :

```
taskYIELD();
```

Vous constaterez que le noyau donne la main à chaque tâche à tour de rôle. Beaucoup d'OS temps réel légers travaillent ainsi, il s'agit de la technique dite du **round-robin** qui suit le principe de fonctionnement d'un tourniquet. Chaque tâche de même priorité prête ou en court d'exécution prendra la main à tour de rôle. Le principal avantage de cette technique est de ne nécessiter qu'une intelligence très réduite au niveau du code du *kernel*.



Nous venons d'illustrer le principe de la coopération entre tâches ainsi que le principal problème amené si une boucle infinie (bug) intervient dans le code d'une tâche. Les tâches bloquées ou prêtes ne peuvent plus prendre la main et l'application tombe !

Pour information, durant la totalité de la trame de TP, nous étudierons le fonctionnement de nos programmes à l'aide de chronogrammes comme ci-dessus. Sachez néanmoins qu'il existe des outils dédiés, souvent propriétaires et donc payants permettant ce type d'analyses. Prenez quelques minutes pour visualiser la vidéo présentant les outils de trace proposés par FreeRTOS (outils payants en fonction des services demandés) :

http://www.youtube.com/watch?feature=player_embedded&v=WTNc1PwoMG4

9 https://www.freertos.org/a00020.html#taskYIELD

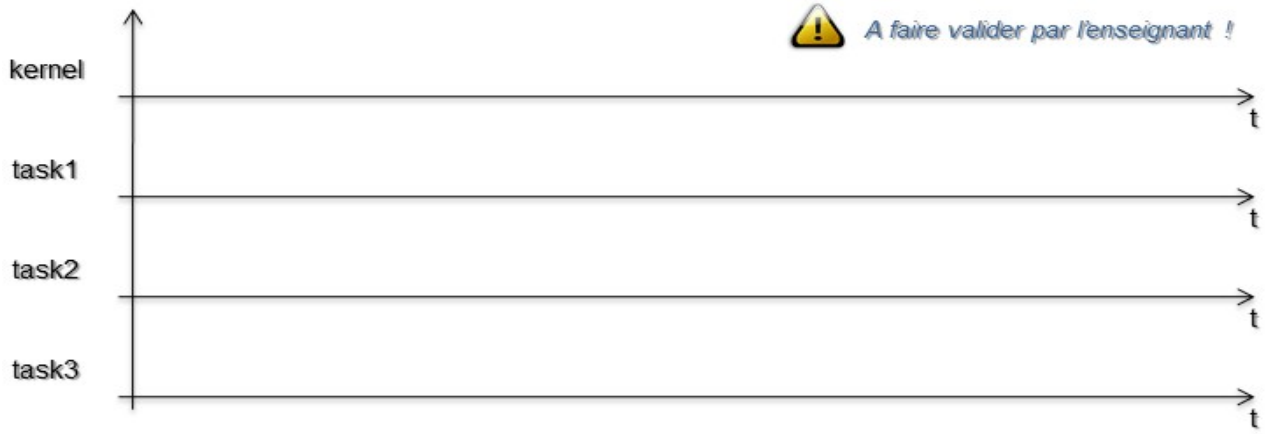
V.4. État bloqué

Intéressons-nous à la fonction bloquante `vTaskDelay()`[10].

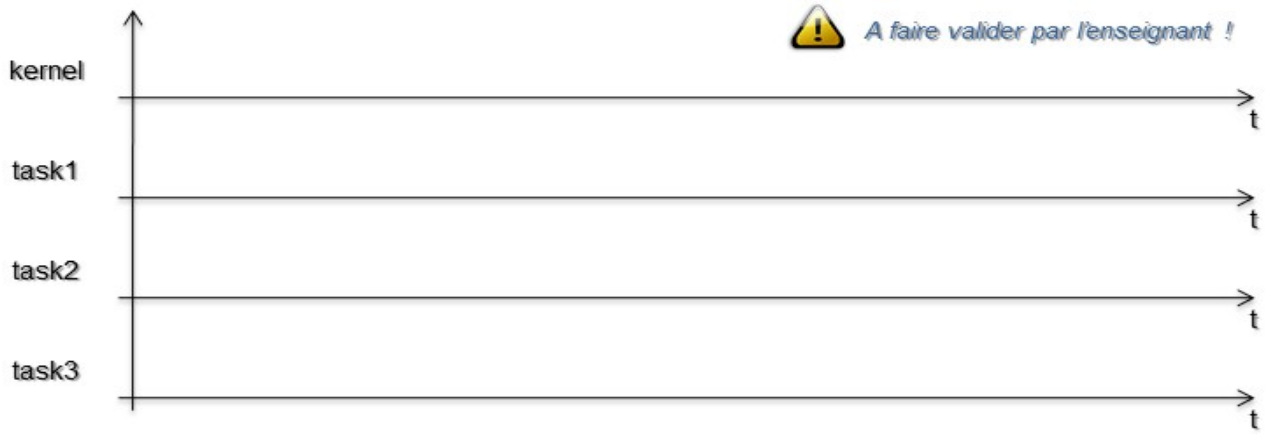
- Mettez à '1' la valeur de la macro `INCLUDE_vTaskDelay` du fichier `FreeRTOSConfig.h`. Pour rappel, ce fichier d'en-tête permet d'adapter FreeRTOS aux besoins de l'application, en n'intégrant au projet que le strict nécessaire.
- Modifiez le programme précédent en supprimant dans la tâche 1 la temporisation logicielle, puis remplacez l'appel à `taskYIELD()` par :

```
vTaskDelay(1000);
```

- Complétez le chronogramme ci-dessous en étant prudent aux quelques pièges pouvant apparaître. Pour information, 1000 signifie 1000 ticks donc 1 seconde dans notre cas (1 tick = 1ms, cf. FreeRTOSConfig.h). Précisez à chaque fois les appels des fonctions `taskYIELD()` et `vTaskDelay(1000)` ainsi que l'état pris par chaque tâche (R = *ready* et B = *blocked*) :



- Modifiez maintenant `task2()` et `task3()` de manière identique à `task1()`. Complétez le chronogramme suivant et préciser l'état pris par chaque tâche :



- Quel code s'exécute lorsque nous nous trouvons dans aucune des trois tâches ?

10 https://www.freertos.org/a00127.html

V.5. Tâche Idle

Enfin, intéressons-nous à la tâche *Idle* et découvrons comment la détourner afin d'y insérer du code utilisateur. Par défaut en mode coopératif la tâche *Idle* ne fait que forcer des commutations de contexte en appelant la fonction `taskYIELD()`.

- Définissez à '1' la macro `configUSE_IDLE_HOOK` présente dans `FreeRTOSConfig.h`.

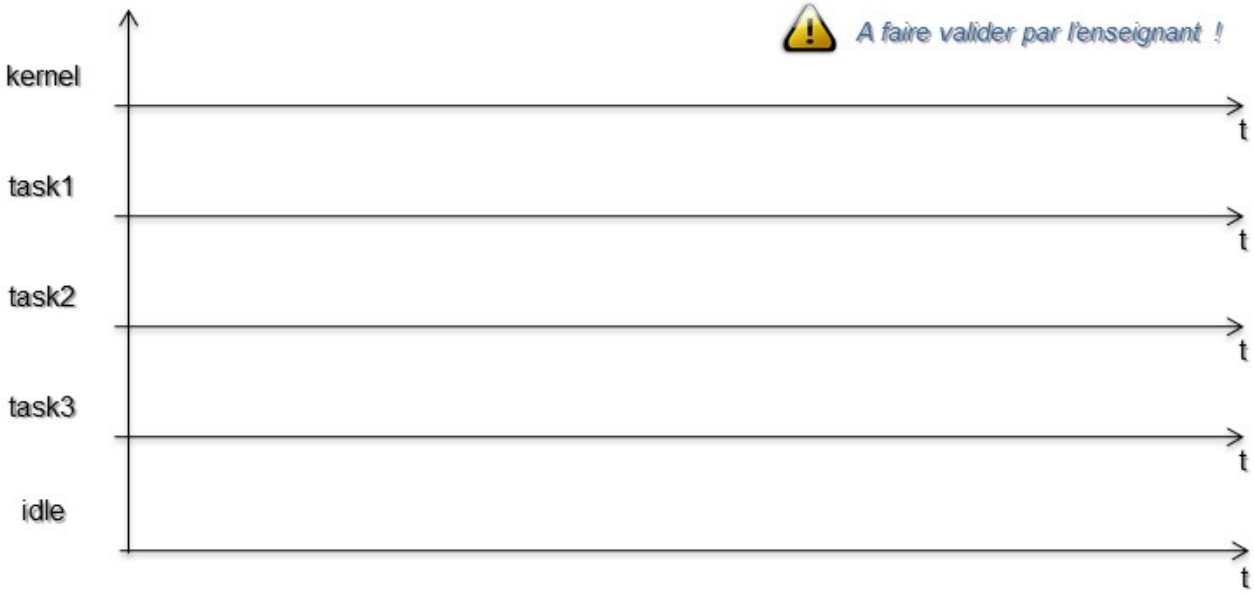
- Par qui la tâche *Idle* est-elle créée ? Retrouvez l'endroit où est créée cette tâche et expliquez.

- Cherchez le code de la fonction associée à la tâche *Idle*. Que fait-elle ?

- À la fin du fichier `freertos.c`, définissez une fonction (et non une tâche !) nommée `vApplicationIdleHook()`. Attention ce nom est imposé par le système. Cette fonction ne fera qu'envoyer une chaîne de caractères :

```
/* TODO: Idle task callback function */
/**
 * @brief function called by idle task
 */
void vApplicationIdleHook( void ){
    ENSI_UART_PutChar('i');
}
```

- Testez votre code et complétez le chronogramme ci-dessous :

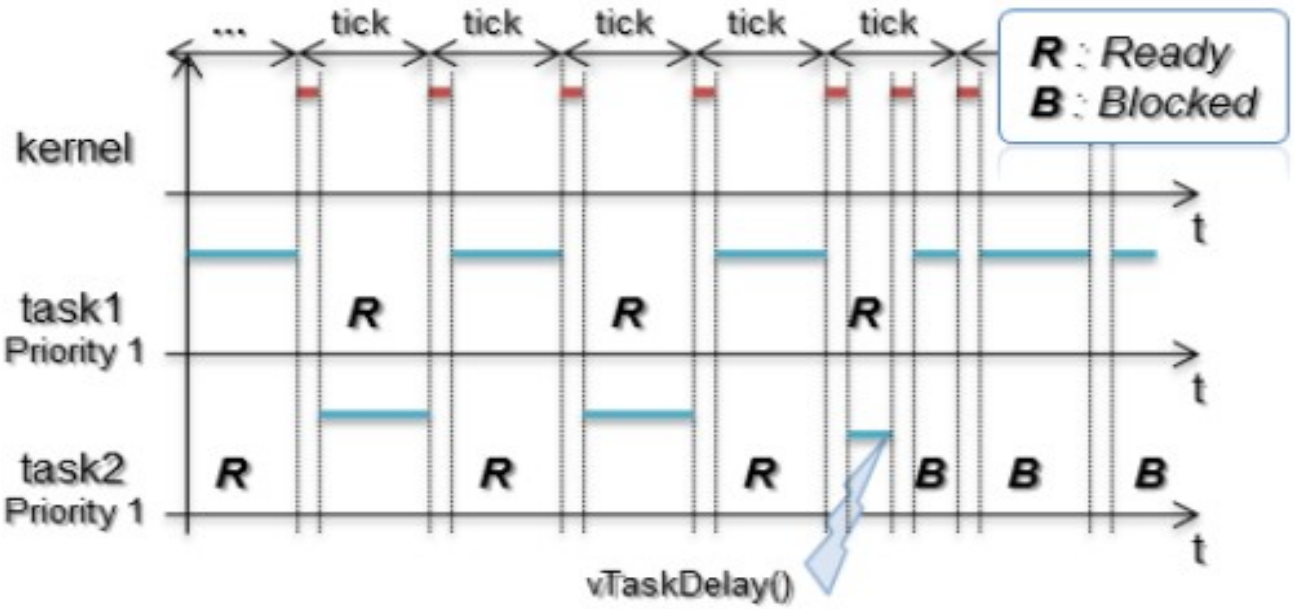


- Proposez des cas d'applications et exemples d'utilisation de la tâche *Idle*. Ne pas hésiter à s'aider du web et d'exemples de détournement de la tâche *Idle* sur d'autres noyaux temps réel.

VI. Ordonnancement en mode préemptif

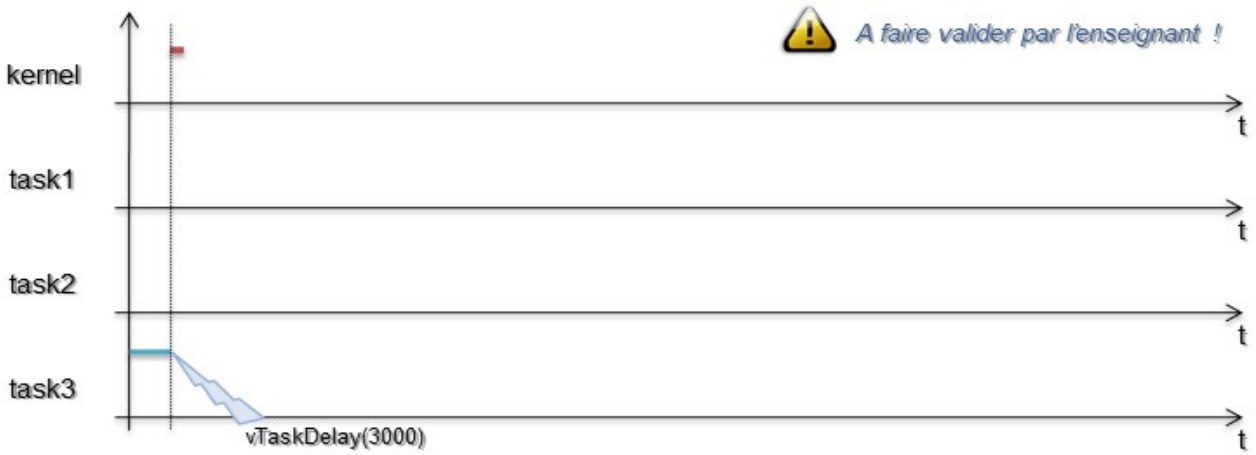
À partir de maintenant et jusqu'à la fin de la trame de TP nous travaillerons exclusivement en mode préemptif.

En mode préemptif, le *scheduler* prend périodiquement la main, interrompant ainsi une tâche en cours d'exécution, puis force un ré-ordonnancement (si nécessaire). Cette périodicité se nomme ***tick*** (le *tick* est généré par un timer matériel avec interruptions) et est configurable sous FreeRTOS à travers la macro `configTICK_RATE_HZ` présente dans `FreeRTOSConfig.h`.



- Dans `FreeRTOSConfig.h`, affectez à '1' la macro `configUSE_PREEMPTION`.
- Dans `FreeRTOSConfig.h`, affectez à '0' la macro `configUSE_IDLE_HOOK`.
- Dans la fonction `app_init()`, modifiez la priorité de la tâche 1 à la valeur 2.
- Compilez, exécutez.

- Complétez le chronogramme suivant et préciser l'état pris par chaque tâche (R = *ready* et B = *blocked*). Attention aux pièges :



- Le comportement du programme peut sembler étrange dans un premier temps (entrelacement de caractères), mais est normal ici. Expliquez vos relevés.

- Dans notre cas, la tâche 1 est-elle périodique ?
  De quoi dépend la périodicité d'une tâche appelant la fonction `vTaskDelay()` ?

- En vous aidant de la documentation FreeRTOS, quel est le traitement réalisé par la fonction `xTaskGetTickCount()` ? Quels sont ses paramètres ?

- Récupérez à chaque réveil (après avoir SET la GPIO) de la tâche 1 la valeur du *tick*, puis envoyez-la à l'ordinateur (s'aider de la fonction standard `sprintf`) :

```
HAL_GPIO_WritePin(Task1_GPIO_Port, Task1_Pin, GPIO_PIN_SET);
// @todo: read current tick value
// @todo: build a new string that contains the tick value
// @todo: send this string to UART
HAL_GPIO_WritePin(Task1_GPIO_Port, Task1_Pin, GPIO_PIN_RESET);
vTaskDelay(1000);
```

- Que fait et comment fonctionne la fonction `vTaskDelayUntil()` ?

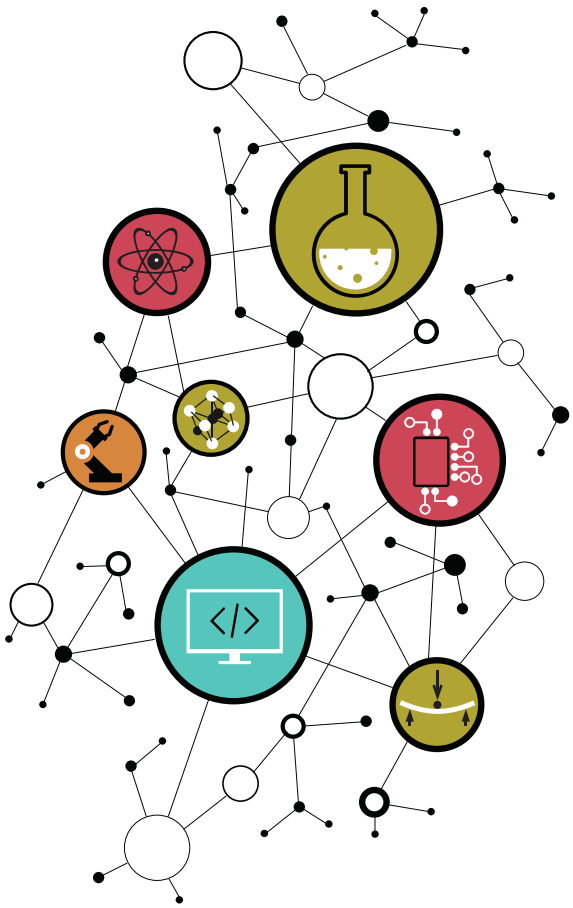
  Remplacez dans votre appel à la fonction `vTaskDelay()` par `vTaskDelayUntil()`.

- Si besoin, expliquez pourquoi votre projet ne compile pas dans un premier temps et comment a été résolu ce problème.

- La tâche 1 est-elle maintenant périodique ? Quelle est l'utilité de la fonction `vTaskDelayUntil()` en comparaison à `vTaskDelay()` ?

PARTIE 3

GESTION MÉMOIRE



I. Travail préliminaire

1. Qu'est-ce qu'une pile ou *stack* et que trouve-t-on sur la pile ?
2. Quelle est la taille par défaut de la pile de la tâche *Idle* dans le cadre de notre trame de TP ? Expliquez votre démarche pour répondre à cette question.
3. Que trouve-t-on généralement sur le tas (*heap*) ?

FreeRTOS propose différentes stratégies de gestion du tas, chacune implémentée dans un fichier parmi `heap_1.c`, `heap_2.c`, `heap_3.c`, `heap_4.c` et `heap_5.c`. Vous pouvez donc les retrouver dans les sources du kernel FreeRTOS, que nous avons également mises à disposition dans l'archive de TP `<archive-tp>/sources/FreeRTOS-kernel/Source/portable/MemMang/` (*Memory Management*). Lors de la configuration du projet avec CubeMX, le choix de la stratégie de gestion du tas (et donc l'inclusion du fichier `heap_X.c` correspondant) vous est laissé à travers le champ « *Memory Management scheme* ».

4. En parcourant ces fichiers (dans l'archive de TP) et la documentation associée[11], indiquez dans les grandes lignes ce que font les fonctions `pvPortMalloc()` et `vPortFree()` :
   - dans le fichier `heap_1.c`
   - dans le fichier `heap_2.c`
   - dans le fichier `heap_3.c`
5. Indiquez les avantages et inconvénients de ces trois stratégies.

11 https://www.freertos.org/a00111.html

II. Memory map (espaces mémoire)

La partie qui suit est extrêmement importante et sujette à énormément de bugs et mauvais développements en milieu industriel, notamment lorsque nous travaillons sur de petits exécutifs temps réel comme FreeRTOS. La problématique est différente sur OS évolué (GNU\Linux, Android …) et processeur équipé de MMU (*Memory Management Unit*). Sur RTOS, le développeur doit avoir une très bonne gestion et maîtrise des ressources mémoire utilisées par la chaîne de compilation et le système. Prenons un petit programme d'exemple et observons le mapping mémoire de données du processeur.

```
int gbl;

/**
 * @fn int main(void)
 */
void main(void){
    int lclMain;
    xTaskCreate(
        task, "task", 100, NULL, 1, NULL);
    vTaskStartScheduler();
}

/**
 * @fn  void task(void *pvParameters)
 */
void task(void *pvParameters){
    int lclTask;
}
```

- Représenter ci-contre un découpage du mapping mémoire en faisant apparaître :

  tas de FreeRTOS ; contexte de la fonction main ; pile de la tâche ; TCB de la tâche ; contexte de la fonction task

- Que trouve-t-on dans le reste de la mémoire ?

- Où se situe physiquement la chaîne de caractères `"task"` ?

- la variable globale `gbl` ?
- la variable locale `lclMain` ?
- la variable locale `lclTask` ?

III. Stack overflow (débordement de pile)

FreeRTOS propose une API de programmation permettant de détecter certaines exceptions du noyau et d'appeler des fonctions de *callback* en cas d'occurrence. Le *kernel* permet notamment de détecter certains *stack overflow* (débordement de pile) et *heap overflow* (débordement de tas). Pour information, les *stack overflows* font partie des bugs les plus répandus durant des développements sur STR et peuvent être délicats à mettre au jour dans certains cas. Il vous est très très très fortement conseillé d'implémenter ce type de détection durant vos phases de développement. Malheureusement, lorsque vous vous trouvez dans l'une de ces fonctions de *callback*, il est déjà trop tard !

- Créez un nouveau projet STM32CubeIDE répondant aux spécifications suivantes :
  - emplacement : dans le répertoire `workspace/rtos03/` de l'archive de TP.
  - nom du projet : `votrenom_rtos03_memory`
  - **ATTENTION** : ne pas configurer les périphériques par défaut !
  - ~~`System Core/GPIO`~~ : pas besoin de GPIO
  - `Middleware/FreeRTOS` : Interface CMSIS v1
    - Onglet *Config parameters* :
      - *Memory Allocation* = « *Dynamic* »
      - *USE_PREEMPTION* et *USE_TASK_NOTIFICATIONS* = « *Enabled* »
      - *Memory Management scheme* = « heap_4 »
      - tous les autres champs à « *Disable* »
    - Onglet *Include parameter*s :
      - *vTaskDelay* = « *Enabled* »
      - tous les autres champs à « *Disable* »

- Copiez-collez les fichiers de `<archive-TP>/sources/rtos03/` (drivers UART, `freertos.c`),vers le *workspace* du projet.
- Dans le `main()`, initialisez le périphérique UART, supprimez la tâche créée par défaut par CubeMX et appelez la fonction `app_init()`.
- Compilez et résolvez les erreurs s'il y en a.

- Dans `FreeRTOSConfig.h`, définissez la macro `configCHECK_FOR_STACK_OVERFLOW` (elle n'existe pas par défaut) à la valeur '1' ou '2'. La valeur de cette macro définit la stratégie de détection de *stack overflow* qui sera appliquée par le noyau de l'OS[12].

- Observez la fonction `vApplicationStackOverflowHook()` présente dans `freertos.c`.
  - Que fait cette fonction ?
  - Est-elle déclarée par le développeur ou par FreeRTOS ?
  - Est-elle définie par le développeur ou par FreeRTOS ?

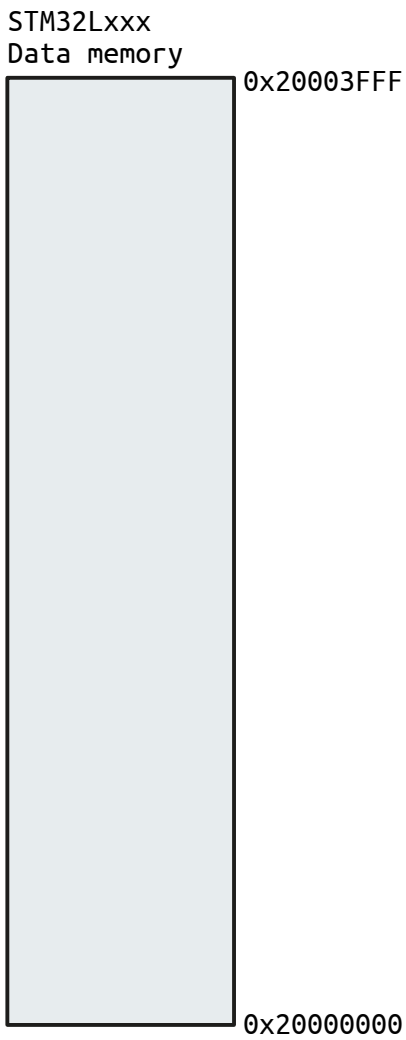
12 http://www.freertos.org/Stacks-and-stack-overflow-checking.html

- Dans `freertos.c` la fonction suivante a été définie et est appelée une fois par `task1()`. Interprétez et illustrez le comportement du programme sur le schéma en bas à gauche.

```
void growth( void ) {
    vTaskDelay(1);
    growth();
}
```

- Réitérez en supprimant l'appel à `vTaskDelay()`. Complétez le schéma de droite.

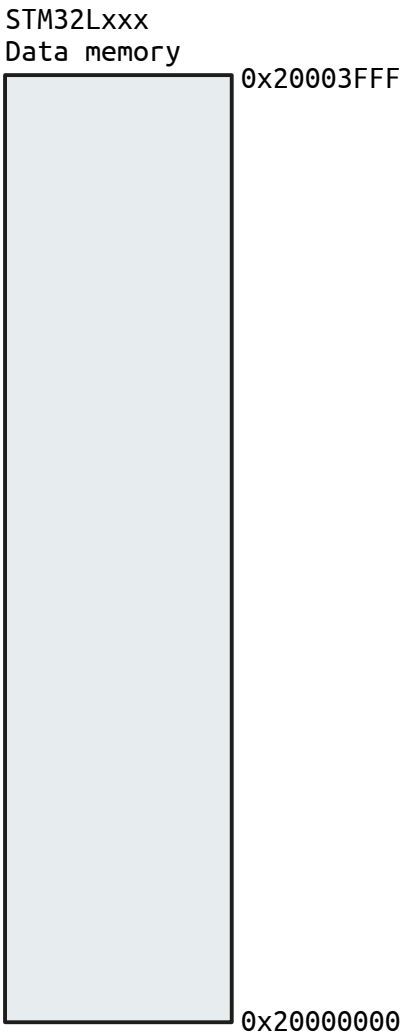
```
void growth( void ) {
    return growth();
}
```



IV. Heap overflow (débordement de tas)

Nous allons maintenant mettre en place les mécanismes de détection de débordement de tas.

- Définissez la macro `configUSE_MALLOC_FAILED_HOOK`[13] à '1' dans `FreeRTOSConfig.h`.
- Observez la fonction `vApplicationMallocFailedHook()` dans le fichier `freertos.c`.
  - Que fait cette fonction ?
  - Est-elle déclarée par le développeur ou par FreeRTOS ?
  - Est-elle définie par le développeur ou par FreeRTOS ?
- Créez une tâche dont la taille de pile dépasse celle du tas (la taille totale du tas est donnée par la macro `configTOTAL_HEAP_SIZE`).
- Interprétez et illustrez le comportement du programme sur le schéma ci-dessous.



13 https://www.freertos.org/a00110.html#configUSE_MALLOC_FAILED_HOOK

V. Voyage dans le mapping mémoire

À la compilation, la *toolchain* (et plus précisément le *linker*) génère un fichier comportant le mapping mémoire de l'application. Nous allons ouvrir ce fichier et en étudier certains points.

- Ouvrez le fichier `Debug/votrenom_rtos03_memory.map` dans l'IDE.

  Le fichier `<archive-TP>/sources/rtos03/Misc/memory_map.txt` est un condensé qui contient des extraits du fichier à étudier.

- Au début du fichier on peut remarquer la mention « *Discarded input sections* ». Il s'agit donc des fonctions compilées mais inutilisées dans le programme, elles ne sont donc pas intégrées à l'exécutable final.
  - Cherchez : `.text.ENSI_UART_GetChar` ; `.text.xTaskGetTickCount`.

- Cherchez la mention « *Memory Configuration* ».
  - Quelle est l'adresse de départ de la RAM ?
  - Quelle est sa taille ?
  - Quelle est l'adresse de départ de la Flash ?
  - Quelle est sa taille ?

- Cherchez la mention `.isr_vector`.
  - Dans quelle mémoire se situe le vecteur d'interruption ?
  - Quelle est sa taille ?

- Cherchez la mention `.text.app_init`.
  - Dans quelle mémoire se situe la fonction ?
  - Dans quelle section se situe la fonction ?
  - Quelle est la taille de la fonction ?
  - Notez que d'autres fonctions utilisées se situent dans la même section.

- Cherchez la mention `.rodata` associée à `./Core/Src/main.o`.
  - Dans quelle mémoire se situe la fonction ?
  - Quelle est l'adresse de départ de cette section ?
  - Quelle est sa taille ?
  - Peut-on expliquer la taille de cette section ?

- Cherchez la mention `.bss.uartRxCircularBuffer`.
  - Dans quelle mémoire se situe la fonction ?
  - Dans quelle section se trouve la variable ?
  - Quelle est sa taille ?
  - Peut-on expliquer la taille de cette variable ?

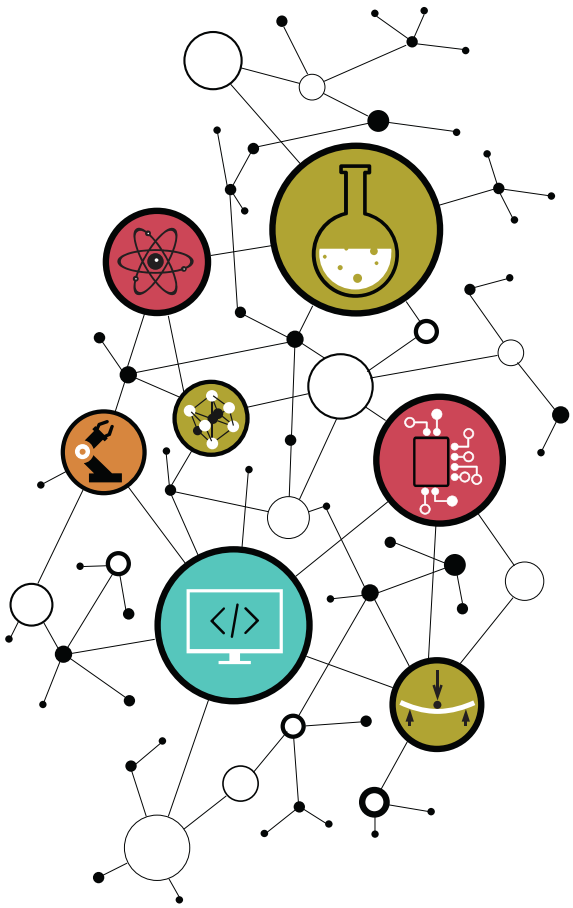
- Cherchez la mention `.bss.ucHeap`.
  - Dans quelle mémoire se situe la fonction ?
  - Quelle est l'adresse de départ de cette variable ?
  - Quelle est sa taille ?
  - Peut-on expliquer la taille de cette variable ?

Rappel : un programme (ou plus largement un fichier binaire, exécutable ou non) est décomposé en sections. Ces sections contiennent :
- `.text` : des instructions (du code) ;
- `.data` : des données initialisées (variables globales, statiques, …) ;
- `.bss` : des données non-initialisées (variables globales, statiques, …) ;
- `.rodata` : des données en *read-only* (variables `const`, chaînes de caractères, …).

FreeRTOS n'utilise pas la pile/*stack* ni le tas/*heap* mis à disposition par la chaîne de compilation. À la place FreeRTOS crée une grande variable non-initialisée, `ucHeap`, utilisée comme un conteneur. Dans cette variable sont stockés les stacks des tâches, leurs TCB, les outils de l'OS (sémaphores, mutex, …).
L'allocation dynamique par FreeRTOS n'est donc qu'un leurre, puisque le tas de FreeRTOS est en réalité une variable globale allouée statiquement (mais dont le contenu est géré dynamiquement par l'OS).

Partie 4

Outils de Synchronisation et Communication



I. Travail préliminaire

Les systèmes d'exploitation (temps-réel ou non) mettent à disposition des outils permettant aux tâches de collaborer. Ceci peut se faire par le biais d'une synchronisation, d'un échange sécurisé de données, de la gestion des accès à une ressource partagée … Ce sont ces outils que nous allons aborder dans cette partie du TP. N'hésitez pas à comparer ces outils avec ceux d'un OS GNU/Linux, qui sont au programme de l'enseignement « Système d'exploitation et Réseau » (S8 en FISE, S9 en FISA).

Aidez-vous de la documentation de FreeRTOS (*Developper Docs* et *API Reference*).

1. En une phrase, à quoi sert un sémaphore ?

   Donnez et présentez succinctement le prototype des fonctions de création, prise et restitution de sémaphore sous FreeRTOS.

2. En une phrase, à quoi sert une queue de messages (ou file d'attente) ?

   Donnez et présentez succinctement le prototype des fonctions de création, envoi et réception de message par file d'attente (*message queue*) sous FreeRTOS.

3. En une phrase, à quoi sert un mutex ?

   Donnez et présentez succinctement le prototype des fonctions de création, prise et restitution de mutex sous FreeRTOS.

II. Synchronisation par sémaphore

Le premier outil inter-tâches que nous allons étudier est le **sémaphore**. Il s'agit d'un mécanisme de synchronisation qui se comporte comme un *flag* (indicateur). Relâcher ou libérer un sémaphore équivaut à donner un « top départ ». Une autre tâche qui voulait prendre ce sémaphore se trouve ainsi débloquée et peut continuer son traitement.

- Créez un nouveau projet STM32CubeIDE répondant aux spécifications suivantes :
  - emplacement : dans le répertoire `workspace/rtos04/` de l'archive de TP.
  - nom du projet : `votrenom_rtos04_tools`.
  - **ATTENTION** : ne pas configurer les périphériques par défaut !
  - `System Core/GPIO` : broches PC4, PB13, et PB15 en mode `GPIO_Output`, respectivement nommées Task1, Task2 et Task3 (champ `User Label`).
  - `Middleware/FreeRTOS` : Interface CMSIS v1
    - Onglet *Config parameters* :
      USE_PREEMPTION = « *Enable* », *USE_TASK_NOTIFICATIONS* = « *Enable* »,
      Memory Allocation = « *Dynamic* », tous les autres champs à « *Disable* ».
    - Onglet *Include parameters* : tous les champs à « *Disable* ».
- Copiez-collez les fichiers de `<archive-TP>/sources/rtos04/` dans le workspace.

- Copiez-collez les fichiers de `<archive-TP>/sources/rtos04/` (drivers UART, `freertos.c`),vers le *workspace* du projet.
- Dans `main()`, initialisez l'UART, vos tâches et supprimez la tâche créée par CubeMX.
- Compilez et résolvez les erreurs s'il y en a.

- Dans `freertos.c` créez une tâche de priorité 2 (tâche 1) et deux tâches de priorité 1.
- Puis éditez les fonctions correspondantes :
  - La tâche 1 devra être de période 3 s. À chaque réveil, elle libérera un sémaphore.

```
HAL_GPIO_WritePin(Task1_GPIO_Port, Task1_Pin, GPIO_PIN_SET);
/* Give semaphore HERE */
HAL_GPIO_WritePin(Task1_GPIO_Port, Task1_Pin, GPIO_PIN_RESET);
vTaskDelay(3000);
```

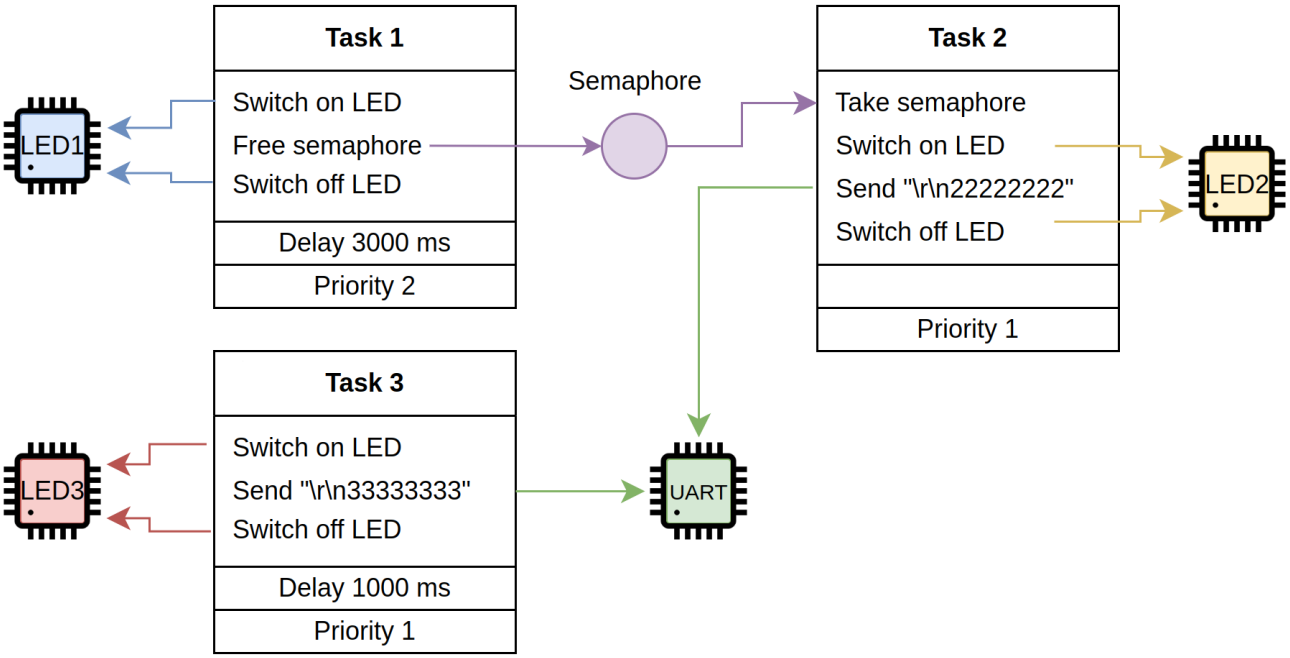
  - La tâche 2 devra attendre la libération du sémaphore commun avec la tâche 1, puis enverra un message via l'UART.

```
/* Take semaphore HERE */
HAL_GPIO_WritePin(Task2_GPIO_Port, Task2_Pin, GPIO_PIN_SET);
ENSI_UART_PutString((uint8_t*)"\r\n22222222");
HAL_GPIO_WritePin(Task2_GPIO_Port, Task2_Pin, GPIO_PIN_RESET);
```

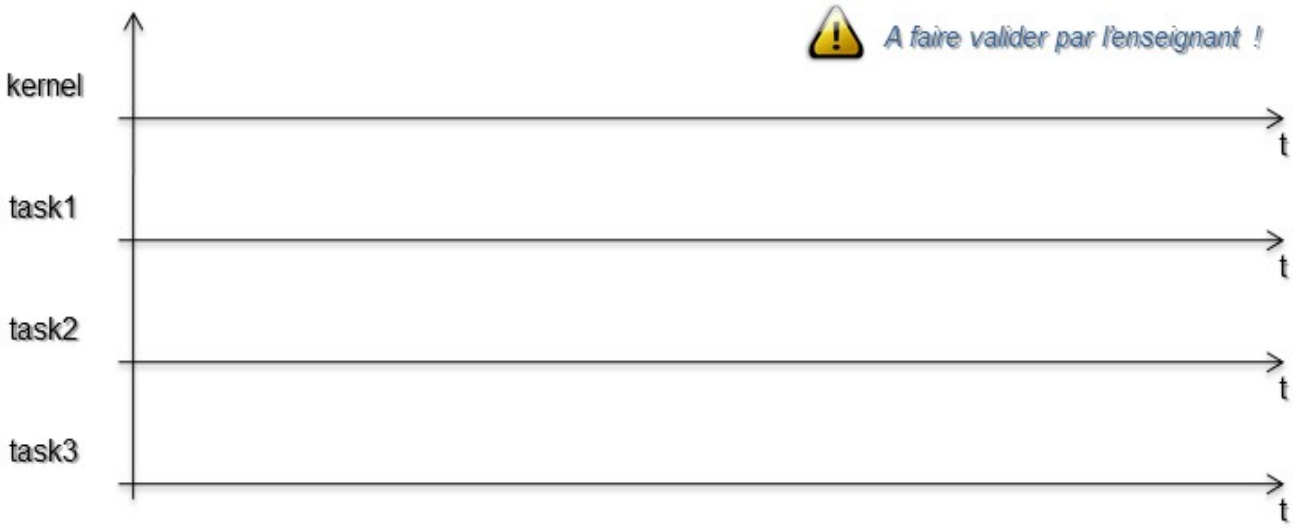
  - La tâche 3 devra être périodique de 1 s et envoyer une chaîne de caractères à l'UART à chaque réveil.

```
HAL_GPIO_WritePin(Task3_GPIO_Port, Task3_Pin, GPIO_PIN_SET);
ENSI_UART_PutString((uint8_t*)"\r\n33333333");
HAL_GPIO_WritePin(Task3_GPIO_Port, Task3_Pin, GPIO_PIN_RESET);
vTaskDelay(1000);
```

Voici ci-dessous un exemple de diagramme permettant de représenter l'application, en utilisant le formalisme maison décrit dans la section I.2 Représentation graphique page 6. Ce schéma est *OS-agnostic* et *hardware-agnostic* (indépendant et de l'OS et du matériel), contrairement au code précédents. Analysez bien ce schéma et établissez le lien avec les codes précédents, car les prochaines fois vous aurez à dessiner ces diagrammes.



- Interprétez les relevés et complétez le chronogramme ci-dessous.



- Quelle est la périodicité de la tâche 2 ?

Les sémaphores sont ainsi utilisés pour synchroniser deux tâches, ou bien demander le démarrage d'une tâche par une autre tâche ou fonction. On les rencontre typiquement en association avec les ISR (*Interrupt Service Routine*).

En effet tant que l'ISR est en cours d'exécution, les interruptions matérielles sont désactivées, y compris celle du *timer* gérant le *tick* de l'ordonnanceur : il est donc « en pause ». Le *scheduler* ne peut plus prendre le contrôle des tâches et la caractéristique temps-réel de l'application se dégrade.

Pour limiter le temps passé dans l'ISR, celle-ci ne fait que libérer un sémaphore. Le réel traitement de l'évènement se fait alors dans une tâche qui attend ce même sémaphore. On peut ainsi considérer que l'ISR délègue le traitement à une tâche dans le but de désactiver le moins longtemps possible l'ordonnanceur.

III. Communication par queue de messages

Comme nous l'avons vu les sémaphores permettent de synchroniser (au moins) deux tâches, mais cet outil relève plus de l'indicateur (*flag*) que de l'échange d'information à proprement parler.

Le mécanisme des **queues de messages** (files d'attente, *queues* ou encore *mailboxes* dans d'autres OS) apporte cette fonctionnalité supplémentaire d'échange de données tout en gardant l'aspect synchronisation de tâches. D'ailleurs, les sémaphores sous FreeRTOS sont implémentés comme des queues de messages vides. Vous pouvez le constater car il n'existe pas de `semphr.c` dans les sources de FreeRTOS, mais seulement un `semphr.h` qui ne fait que *wrapper* l'API de gestion des queues de messages (*simple skin*) et utilise donc les définitions de `queue.c` et `queue.h`.

- Modifiez le code précédent pour supprimer (ou commenter) les instructions concernant les sémaphores, puis éditez le code pour correspondre au cahier des charges suivant :
  - La tâche 1 devra être de période 3 s. À chaque réveil, elle récupérera la valeur du *tick* et l'enverra dans une queue de message.
  - La tâche 2 devra attendre l'arrivée d'un message émis par la tâche 1, puis afficher celui-ci sur la console via l'UART.
  - La tâche 3 devra être périodique de 1 s et envoyer une chaîne de caractères à l'UART à chaque réveil.

- Tracez le diagramme d'application correspondant (cf I.2 Représentation graphique page 6), sur la page suivante (pour avoir de la place).

- Complétez le chronogramme suivant et interprétez le résultat.

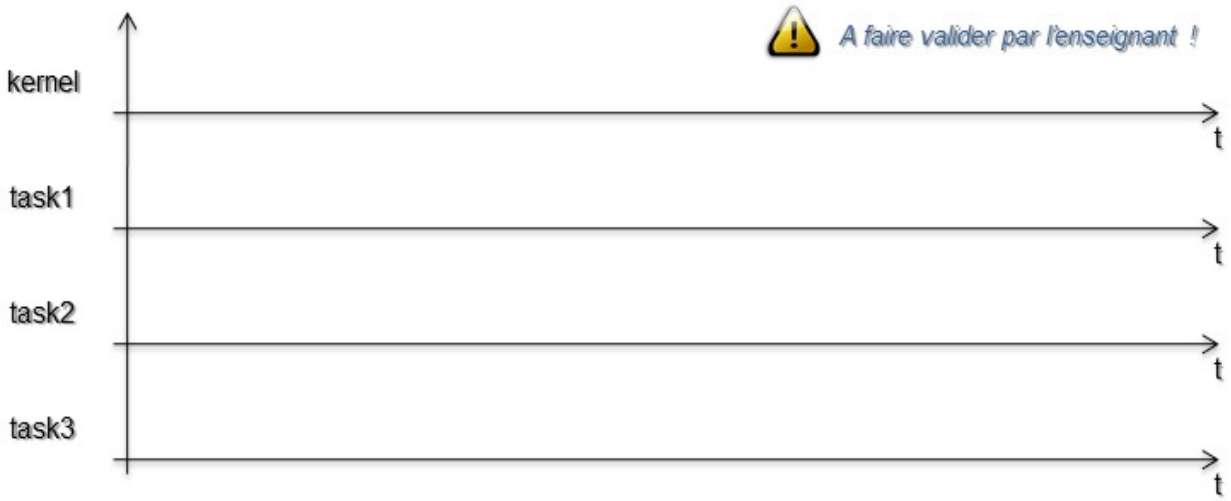
- Tracez le diagramme d'application correspondant (cf I.2 Représentation graphique page 6).

La communication inter-tâches est le deuxième concept important vu dans ce chapitre. Elle consiste en la capacité d'effectuer des échanges sécurisés d'informations entre différentes tâches de l'application. Notez que ces échanges peuvent se produire avec plusieurs écrivains et/ou plusieurs lecteurs.

IV. Timeout

Certaines fonctions pour la gestion de queue de messages ou de sémaphores utilisent un *Timeout*. La notion de *timeout* ne s'applique qu'à des appels système bloquants. Lorsqu'une tâche est bloquée, 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 (libération de sémaphore, écriture dans une file d'attente ...).

- Reprenez l'exercice précédent et forcez le réveil de la tâche 2 toutes les secondes en utilisant le *Timeout* associé à la fonction `xQueueReceive()`.
  - Soit la tâche s'est réveillée après réception d'un message, auquel cas elle transmet le message reçu vers l'UART.
  - Soit la tâche s'est réveillée après *timeout*, auquel cas elle transmet un message `"\r\n22222222 : timeout"` à l'UART.
- Complétez et interprétez le chronogramme ci-dessous.



- Que se passe-t-il si le *timeout* d'une fonction bloquante est mis à '0' ?

- Que se passe-t-il si le *timeout* d'une fonction bloquante est mis à `portMAX_DELAY` (la macro `INCLUDE_vTaskSuspend` doit être définie à '1') ?

- À quelle valeur théorique de *timeout* correspond cet argument ?

- Peut-on trouver un timeout sur une fonction système non bloquante ?

V. Protection de ressource par section critique

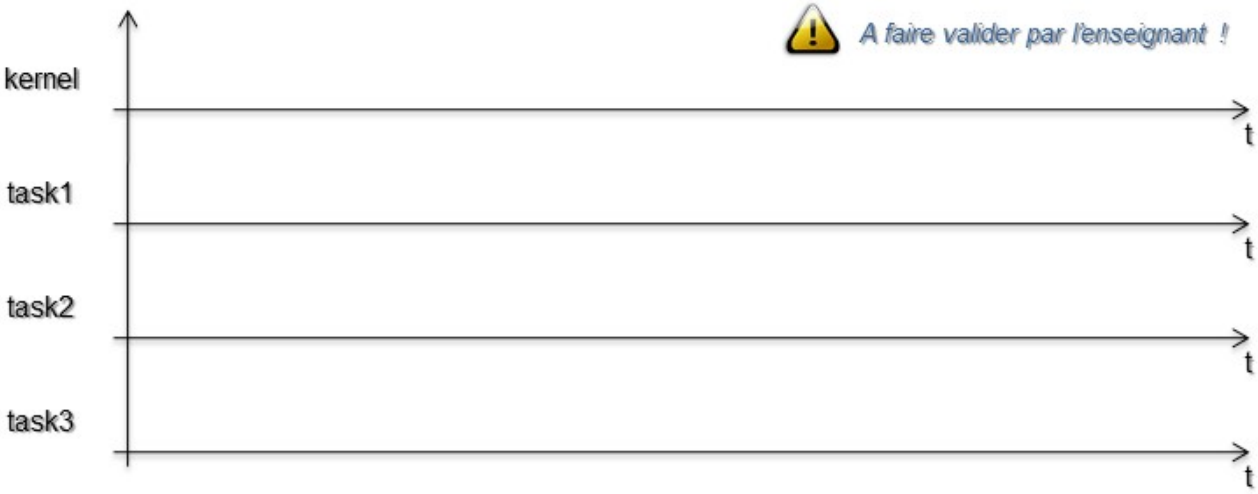
Avec l'utilisation de systèmes multi-tâches, il est très fortement probable qu'une ressource partagée (donnée, périphérique, …) soit manipulée par plusieurs tâches distinctes de manière quasi-simultanée. Ces manipulations simultanées peuvent mener à des comportements hasardeux voire dangereux qui en plus sont parfois difficilement détectables. Nous appelons **section critique** toute portion de code pour laquelle la bonne exécution et l'intégrité des données doivent être garanties. Il s'agira le plus souvent de protéger une ressource partagée (variable globale, accès à un périphérique …), comme décrit plus haut. Une section critique peut être protégée par différents outils système :

- Sémaphores (en synchronisant les tâches, cf section précédente)
- Mutex (*mutual exclusion*)
- Fonctions dédiées, le plus souvent par masquage d'interruption.

Vous avez normalement dû constater que les tâches 2 et 3 étant de même priorité, elles se partagent à tour de rôle l'accès à l'UART. Nous allons donc protéger l'accès à ce périphérique. Cela signifie qu'une tâche ayant pris cette ressource matérielle la gardera jusqu'à ce qu'elle ait fini le traitement en cours.

- Pour les tâches 2 et 3, placez la fonction d'envoi de données à l'ordinateur dans une section critique. Utilisez les macros `taskENTER_CRITICAL()` et `taskEXIT_CRITICAL()`.
- Tracez ici le diagramme équivalent.

- Complétez et interprétez le chronogramme ci-dessous.



- Parcourez les sources de FreeRTOS pour retrouver la définition des deux macros utilisées. Que font ces deux macros ? Quel problème cela peut-il poser ?

Cette implémentation des sections critiques par FreeRTOS est assez dangereuse, notamment dans l'exemple actuellement présenté, vu que les Ticks (générés par timer matériel) ne sont plus vus de l'ordonnanceur pendant la durée d'exécution de la section (section longue en temps d'exécution).

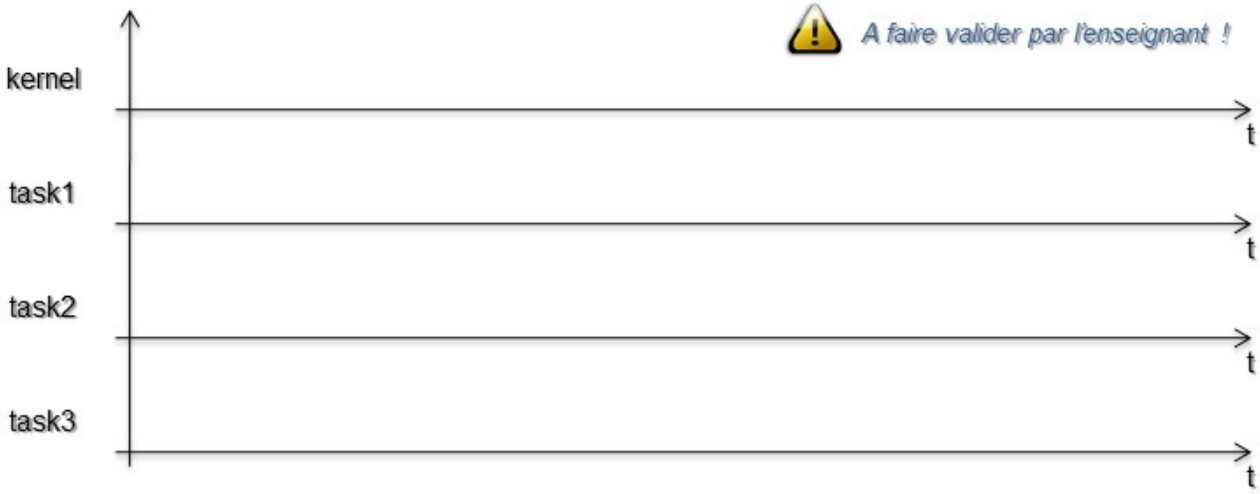
VI. Protection de ressource par mutex

Durant l'exercice précédent vous avez été amené à manipuler des sections critiques en utilisant les fonctions `taskENTER_CRITICAL()` et `taskEXIT_CRITICAL()`. Cependant ces deux fonctions sont à manier avec précaution car une région critique ainsi créée ne peut plus être préemptée par le noyau, ni par les interruptions matérielles en dessous d'un certain niveau de priorité système (section non interruptible). Ceci peut donc devenir très dangereux en cas de mauvaise programmation et en fonction de la criticité de l'application.

À l'aide de **mutex**, nous allons créer des sections critiques pouvant être préemptées par le système et également interrompues par les périphériques matériels. Gardez tout de même en tête que les sections critiques doivent être les plus courtes possible.

- Reprendre le diagramme de l'exercice précédent et adaptez-le de sorte à protéger l'accès à l'UART non plus par une section critique, mais par un mutex.
  - On rappelle que « protéger l'accès » signifie d'empêcher l'accès à une ressource si celle-ci est déjà en cours de manipulation.
- Adaptez maintenant votre programme de la même manière : supprimez les sections critiques et utilisez un mutex.
  - Il y aura sûrement des ajustements à faire : lisez bien les messages d'erreur et la documentation sur les mutex !

- Complétez et interprétez le chronogramme ci-dessous.



- Sous FreeRTOS, quelle différence existe-t-il entre un sémaphore binaire et un mutex ?

- Dans notre cas, pourrait-on protéger la ressource par un sémaphore plutôt que par un mutex ? Lequel est le plus intéressant ici ?

La gestion de ressources partagées est plus sûre en utilisant des mutex que des sections critiques. Pour rappel, les sections critiques désactivent les interruptions et donc le kernel lui-même !
Au contraire, le mutex est un mécanisme géré par le kernel ce qui permet de laisser la main à l'ordonnanceur, qui gérera l'exécution en fonctions des tâches bloquées (en attente du mutex) et de celles prêtes (mutex disponible).

VII. Bibliothèque UART avec appels système

Nous allons, dans cet ultime exercice, modifier plus profondément les sources de la librairie de gestion du module UART. Le but étant d'obtenir une bibliothèque optimisée pour travailler avec FreeRTOS (pas en vitesse d'exécution mais en robustesse et efficacité).

À titre indicatif, beaucoup d'applications font cohabiter bibliothèque réseau (ou *stack* réseau) et système d'exploitation. STMicroelectronics propose par exemple une librairie réseau indépendante de tout OS, qui n'est donc pas optimisée pour travailler avec notre kernel. FreeRTOS propose en revanche une librairie réseau implémentant des appels système qui est donc optimisée pour cohabiter avec le noyau, néanmoins cette *stack* est un outil propriétaire. Observons rapidement le coût de certains de ces services en 2014 (gratuit en 2018 depuis le rachat par AMAZON) :

- Supprimez dans les fichiers `ensi_uart.c` et `ensi_uart.h` toute référence à l'utilisation du *buffer* circulaire permettant l'échange d'information entre l'ISR de réception de l'UART et la fonction `ENSI_UART_GetChar()`.
- Modifiez l'ISR ainsi que la fonction `ENSI_UART_GetChar()` en synchronisant par queue de message les réveils de la fonction d'interruption avec l'appel de la fonction `ENSI_UART_GetChar()`. Chaque caractère reçu sera posté dans la file d'attente et la fonction de réception de caractères implémentera donc un appel système bloquant en vidant cette queue de message.
- Une fois ce travail réalisé, modifiez le code de la tâche 3 de façon à réceptionner puis renvoyer des chaînes de caractères envoyées depuis l'ordinateur. Assurez-vous du bon fonctionnement du programme.
- Ultime test : envoyez un fichier texte depuis l'ordinateur et assurez-vous de sa bonne réception et renvoi par l'application embarquée. Le fichier est présent dans le répertoire `<archive-TP>/sources/rtos04/Misc/`. Sous TeraTerm, aller dans Fichier → Envoyer un fichier ...

```
ENSI_UART_PutString( "\r\ndisco# " );
while (1) {
     if( ENSI_UART_GetString(str_tmp) ) {
         ENSI_UART_PutString(str_tmp);
         ENSI_UART_PutString("\r\ndisco# ");
     }
}
```

- Tracez le diagramme correspondant à l'application ainsi développée.