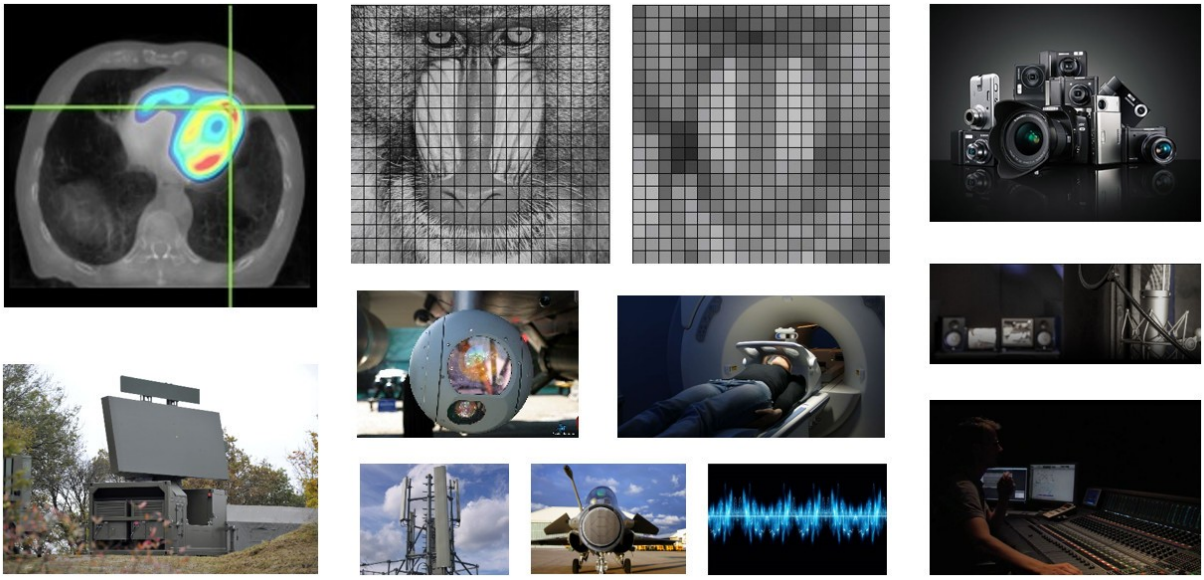
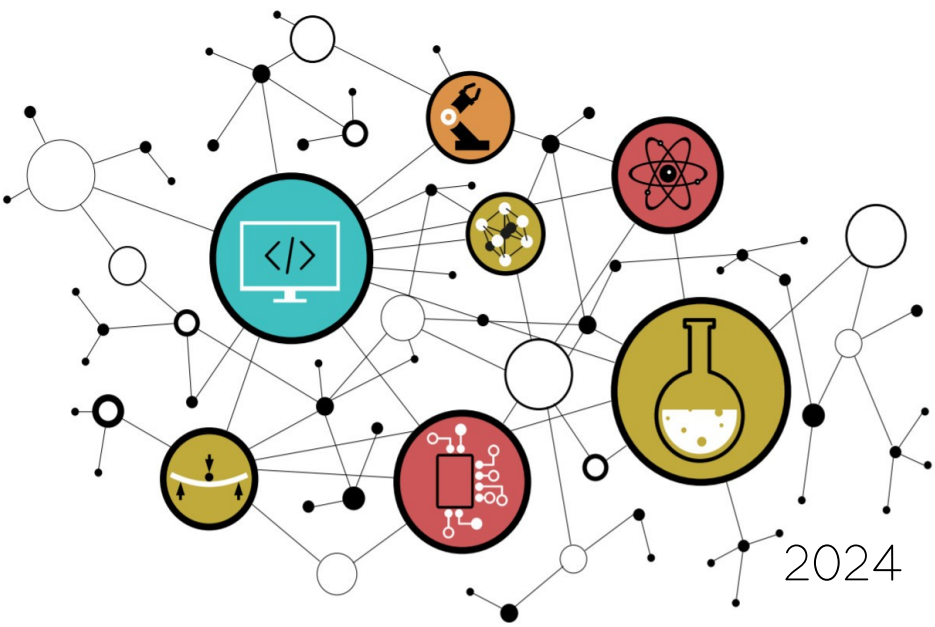
ENSI CAEN

2A FISE GPSE SATE

SYSTÈMES TEMPS RÉEL

TRAVAUX PRATIQUES





2024

CONTACTS

Équipe enseignante

hugo descoubes
hugo.descoubes@ensicaen.fr
+33 (0)2 31 45 27 61

Dimitri Boudier
dimitri.boudier@ensicaen.fr

ENSICAEN
6 boulevard Maréchal Juin
CS 45053
14050 CAEN cedex 04

RESSOURCES ET OUTILS DE DEVELOPPEMENT

Les différentes ressources numériques sont accessibles sur la plateforme pédagogique de l'ENSICAEN. Télécharger l'archive complète de travail rts.zip

https://foad.ensicaen.fr/course/view.php?id=118

ÉVALUATION

L'enseignement comportera 2 évaluations distinctes. Une évaluation sur table et une évaluation pratique. Voici le détail des points sur lesquels vous serez évalués :

ÉVALUATION THEORIQUE

- SAVOIR – 7pts : Questions de culture générale pouvant traiter sur tout point abordé en séance de cours présentiel ou présent dans le support de travail. *Connaissances fondamentales et culture scientifique de l'ingénieur électronicien autour des systèmes temps réel*

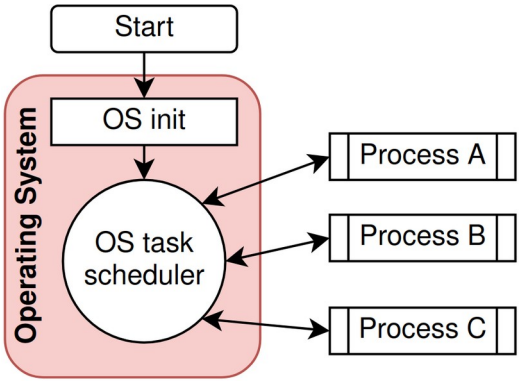
- ANALYSER – 13pts : Exercice d'analyse d'une application simple portée sur une technologie de système temps réel non vu en enseignement. *Évaluation des capacités de l'élève ingénieur à adapter les concepts étudiés en enseignement sur de nouvelles technologies*

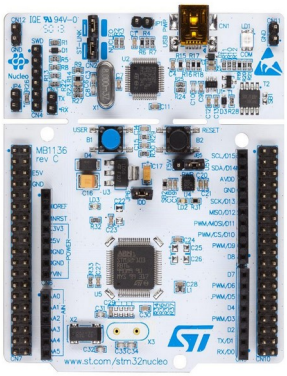
ÉVALUATION PRATIQUE

- DEVELOPPER – 20pts : Réalisation d'un projet simple imposant des contraintes temps réel sous FreeRTOS, technologie étudiée en enseignement

Systèmes Temps-Réel

Support de Travaux Pratiques





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

Ressources

Toutes les ressources (supports CM, TP et outils) sont sur la page Moodle du cours :

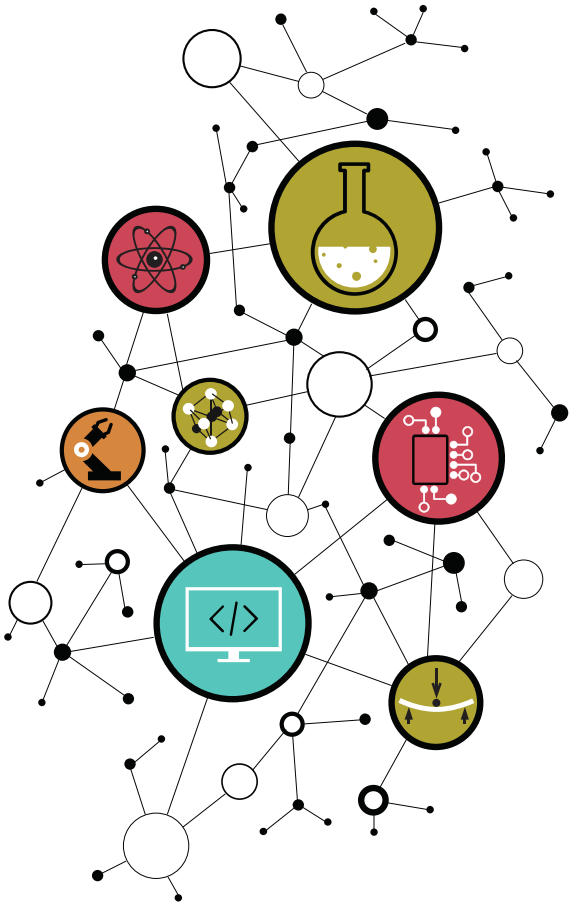
https://foad.ensicaen.fr/course/view.php?id=840

Except where otherwise noted, this work is licensed under
https://creativecommons.org/licenses/by-nc-sa/4.0/

Sommaire

Partie 1

Environnement de travail



I. Quels outils ?

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

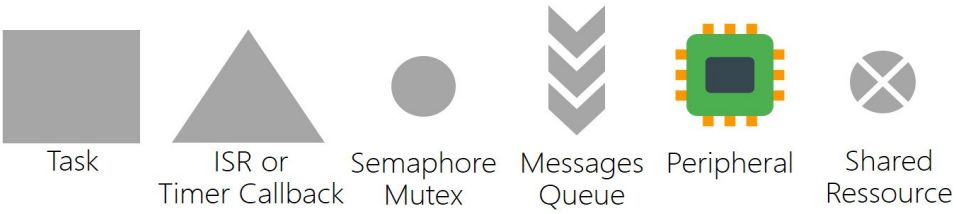
I.1. Système d'exploitation

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.

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.2. Matériel

La trame de TP se fera sur **micro-contrôleur STM32L073RZ**[2]. C'est donc l'occasion de découvrir un processeur de la famille **ARM**, à travers une solution de **STMicroelectonics**.

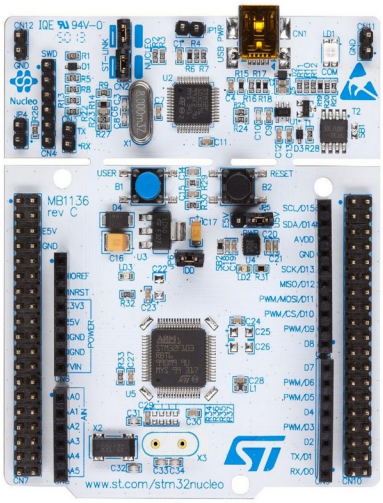
La gamme STM32L0 fait partie des nombreuses solutions STMicroelectonics qui se basent sur un micro-contrôleur à cœur ARM Cortex-M 32-bit. L'intérêt de cette gamme est de proposer des composants *Ultra-low power*, en comparaison aux autres MCU STM32.

STM32L0 MCU Series - 32-bit Arm® Cortex®-M0+

• Ultra low leakage process • Dynamic voltage scaling • 14 to 100-pin • 5 clock sources • Advanced RTC w/ calibration • 12-bit ADC 1.14 Msps • Multiple USART, SPI, I²C • Multiple 16-bit timers • LP UART1 • LP Timers1 • 2 watchdogs • Reset circuitry POR/PDR • Brown-out Reset • DMA • AES-128	Product line	Flash (KB)	RAM (KB)	EE - PROM (Bytes)	Power supply	PVD [2]	TEMP sensor	2x ULP COMP	2x 12-bit DAC	Touch sense	TRNG	USB 2.0 FS Crys - tal-less	Segment LCD Driver
	STM32L0x0 Value line	Up to 128	Up to 20	Up to 512	Down to 1.8V								
	STM32L0x1 Access	Up to 192	Up to 20	Up to 6K	Down to 1.65V	•	•	•					
	STM32L0x2 USB	Up to 192	Up to 20	Up to 6K	Down to 1.65V	•	•	•	•	•	•	•	
	STM32L0x3 USB & LCD	Up to 192	Up to 20	Up to 6K	Down to 1.65V	•	•	•	•	•	•	•	Up to 4x52 or 8x48

Note 1: Low-power peripherals available in ultra-low-power modes
Note 2: PVD = Programmable voltage detector

En TP, nous utiliserons donc une **carte d'évaluation SMT32 Nucleo-L073RZ**[3], qui comporte notamment le micro-contrôleur STM32L073RZ ainsi qu'une sonde de programmation et de *debug*.



Cette carte fait partie de la famille STM Nucleo-64, principale famille de kit d'évaluation de ST pour ses processeurs ARM.

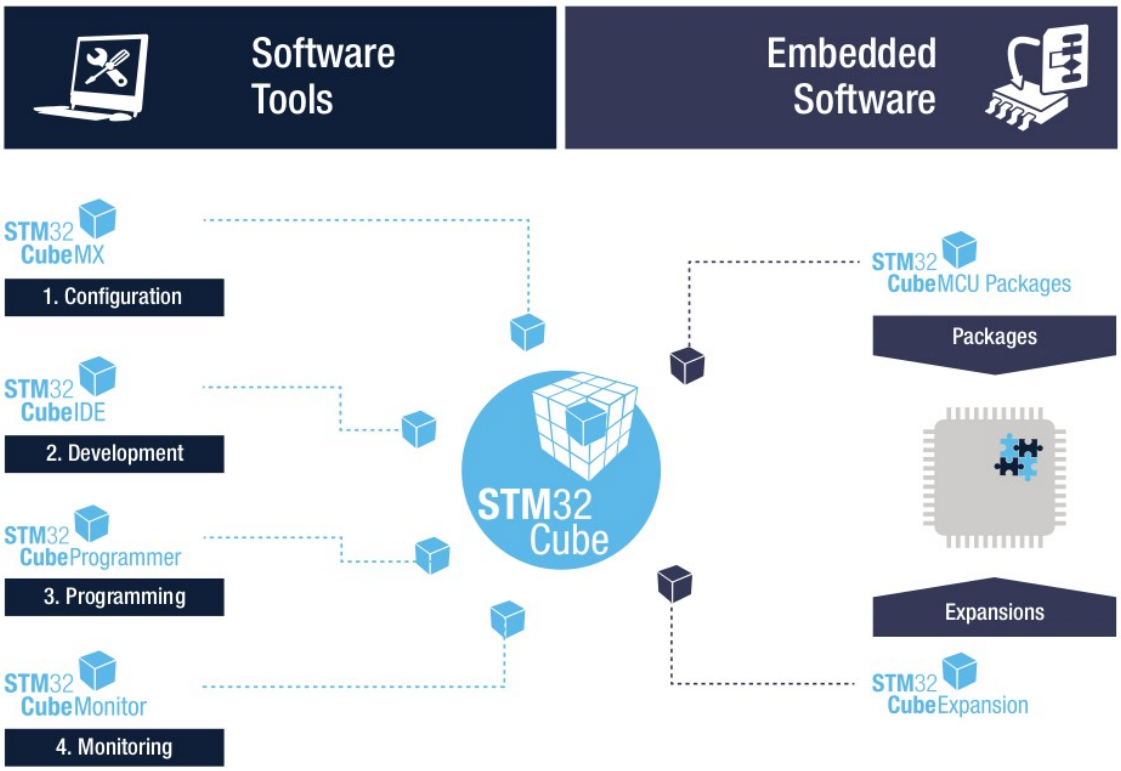
En plus du micro-contrôleur et de la sonde de programmation, cette carte embarque un bouton poussoir et une LED, ainsi que des connecteurs ST morpho (76 broches) et Arduino. Elle est équipée du STLink VCP (*Virtual COM Port*), permettant une communication série via le port USB. Le tout permet ainsi de développer rapidement des applications embarquées.

2 https://www.st.com/en/microcontrollers-microprocessors/stm32l073rz.html
3 https://www.st.com/en/evaluation-tools/nucleo-l073rz.html

I.3. 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*).

Si on décidait de partir sur de la programmation du micro-contrôleur à l'étage registre (comme en TP MCU de première année), il faudrait ajouter quelques heures à la formation tant les Cortex-M sont complexes (en comparaison à des PIC18). Nous utiliserons donc **STM32CubeMX**, qui permet de configurer graphiquement 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 « I. STM32CubeIDE ».

4 https://www.st.com/en/ecosystems/stm32cube.html

II. Prise en main des outils

Commençons les TP en douceur, en créant un projet simple pour STM32. Avec celui-ci, nous manipulerons des **GPIOs** et mettrons en place une **liaison série (UART)**.

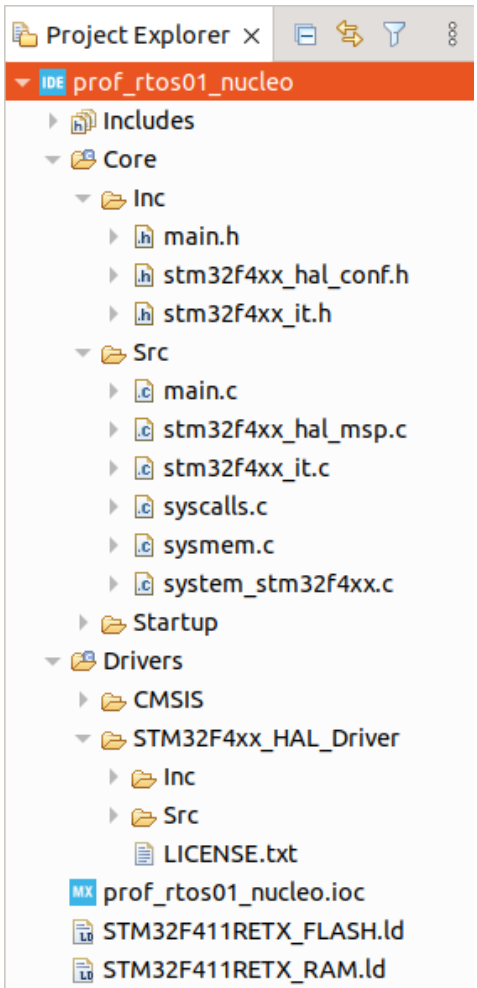
II.1. Création d'un projet STM32CubeIDE

- En vous aidant du chapitre « I.1 Création d'un projet » de l'annexe, créez un projet répondant aux spécifications suivantes :
  - emplacement : dans le répertoire `workspaces/` de l'archive de TP.
  - nom du projet : `votrenom_rtos01_nucleo`
  - `System Core/GPIO` : broches PC4 en mode `GPIO_Output`, renommée Task1 (champ `User Label`).

Une fois le code généré ( ), le projet est créé et contient des fichiers sources pré-remplis. Vous devriez maintenant voir l'explorateur de projet (onglet de gauche dans l'IDE), sinon fermez la sous-fenêtre de configuration que vous venez d'utiliser.

Observons rapidement l'arborescence du projet :

- `Core/Src/` : répertoire contenant les fichiers sources (en C et asm) propres à l'application
- `Core/Inc/` : répertoire contenant les *headers* associés aux sources de l'application.
- `Drivers/STM32L0xx_HAL_Driver/Src/` et `Inc/` : répertoires contenant les sources et *headers* propres à la HAL (*Hardware Abstraction Layer*) des **MCU STM32**. Autrement dit, ce sont les « pilotes » (ou BSP) fournis par CubeMX pour utiliser les périphériques des **MCU STM32**.
- `Drivers/CMSIS/Src/` et `Inc/` : répertoires contenant les sources et headers propres à la couche d'abstraction des **cœurs ARM**. Autrement dit, c'est une interface de programmation ARM (donc **indépendant du fabricant** du MCU).



Pour plus d'informations sur ces deux derniers points (HAL et CMSIS), se référer au chapitre « I.3 HAL – Hardware Abstraction Layer » de l'annexe.

II.2. Manipuler les GPIOs avec la HAL

II.2.a) BP et LED de la Nucleo-64

La carte Nucleo-64 (famille dont la Nucleo-LR073 fait partie) intègre un bouton poussoir et une LED directement utilisables car ils sont configurés par défaut lors de la création du projet. Pour prendre en main la carte, nous allons les utiliser.

- Dans la boucle `while(1){...}` de la fonction `main()`, écrire le code suivant.

```
if( HAL_GPIO_ReadPin(B1_GPIO_Port, B1_Pin) == GPIO_PIN_RESET ) {
      HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_SET);
}
else {
      HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_RESET);
}
```

Attention : écrivez entre les balises dédiées !
Les fichiers générés par STM32CubeMX contiennent des balises de commentaires indiquant les zones dans lesquelles le développeur doit écrire.
En cas de reconfiguration projet (changement de paramètres d'un périphérique par ex), STM32CubeMX écrasera toutes les instructions qui ne sont pas comprises entre ces balises, entraînant une potentielle perte du code déjà rédigé !

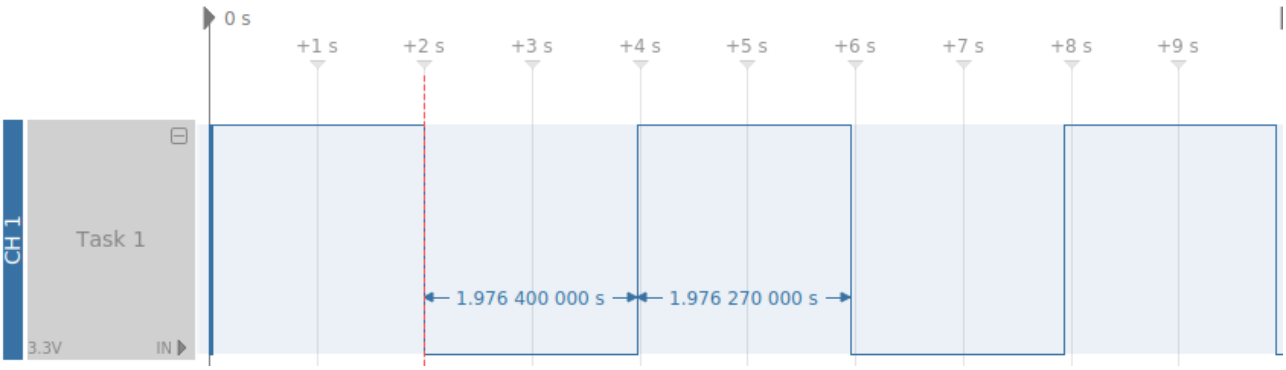
- Compilez (*Project* → *Build All*) et lancez l'exécution du programme (*Run* → *Run*).

Vous aurez vite deviné comment interagir avec la carte pour valider le fonctionnement du programme.

II.2.b) GPIOs configurées manuellement

En plus du BP et de la LED de la Nucleo, nous avons demandé la configuration de quatre broches en sortie **GPIO**. Vérifions leur fonctionnement.

- Commentez (ou effacez) les lignes de code de l'exercice précédent.
- Écrivez un programme qui fait clignoter la broche Task1 à une période de 2 secondes. Vous aurez besoin de consulter certains fichiers de la HAL :
  - Quel est le nom du port et le nom de broche associés à Task1 ? Voir `main.h`.
  - Quelle fonction permet de changer une GPIO d'état ? Regardez parmi les nombreuses fonctions déclarées dans `stm32l0xx_hal_gpio.h`.
  - Quelle fonction permet d'imposer une temporisation logicielle (et donc bloquante) ? Regardez dans `stm32l0xx_hal.h`.
- Vous pouvez aussi faire clignoter la LED pour un résultat visuel.
- Validez la période de votre signal avec un oscilloscope ou un analyseur logique.



Exemple de résultat attendu avec un analyseur logique.

II.3. Liaison série

Maintenant que les GPIOs sont fonctionnelles, il reste à tester notre **interface UART**. Aujourd'hui encore, il s'agit de l'une des premières interfaces mises en place lors de développement sur micro-contrôleur. La rapidité de mise en œuvre, la simplicité du protocole et sa robustesse en font sa force. Testons son fonctionnement en s'aidant de la HAL fournie par STM32Cube.

II.3.a) STLink Virtual COM Port

Pour ces TP nous utiliserons l'UART2, ce périphérique étant relié au **STLink *Virtual COM Port* (VCP)**. Il s'agit d'une liaison série émulée à travers la communication USB de la Nucleo-64. Elle ne nécessite aucun matériel supplémentaire et s'avère donc être un outil pratique, qui plus est utile pour un travail en dehors des séances.

Cependant, à cause de son lien avec le STLink VCP, l'UART2 n'est pas physiquement relié aux connecteurs Morpho (PA2/PA3) et Arduino (D0/D1) de la Nucleo-64. Si vous voulez observer les signaux de l'UART2, il faut brancher l'analyseur logique sur les broches RX et TX du connecteur CN3 en haut de carte. Attention : ces broches correspondent au Rx et Tx du STLink (donc de l'ordinateur), pas de l'UART2 ! Si vous voulez impérativement utiliser un UART sur les connecteurs classiques (Morpho ou Arduino) de la Nucleo-64, référez-vous à l'annexe « II.2 UART et STLink Virtual Com Port ».

II.3.b) UART avec la HAL

– Message à caractère informatif –

Comme tout autre périphérique, les UART (donc l'UART2) sont configurables par l'assistance STM32CubeMX. Cependant, les fonctions de la HAL ainsi générées mettent en œuvre des mécanismes de protection d'accès au périphérique, ce que nous voulons éviter pour des raisons pédagogiques. Nous allons donc vous fournir d'autres fichiers sources permettant l'utilisation de l'UART (cf. section suivante).

Si vous souhaitez apprendre à manipuler les périphériques UART avec la HAL, sachez que de nombreux tutoriels existent sur les Internets. Pour vous aider, voici les principales fonctions que vous pourriez rencontrer :

- `HAL_UART_Init()`
- `HAL_UART_Transmit()` et `HAL_UART_Transmit_IT()`
  - La première fonction étant bloquante, la deuxième non-bloquante (et appelle la *callback* `HAL_UART_TxCpltCallback()` après transmission).
- `HAL_UART_Receive()` et `HAL_UART_Receive_IT()`
  - La première fonction étant bloquante, la deuxième non-bloquante (et appelle la *callback* `HAL_UART_RxCpltCallback()` après réception).

– Fin de message –

II.3.c) UART avec bibliothèque maison

Comme expliqué dans la page précédente, une bibliothèque UART maison (écrite par vos enseignants) vous est fournie : `ensi_uart.c` et `ensi_uart.h`. Mettons-la en œuvre.

- Commentez (ou effacez) les lignes de code de l'exercice précédent.
- Effectuez l'importation de ces fichiers dans votre projet :
  - Copiez les fichiers fournis, répertoire `<archive TP>/resources/rtos01/Drivers`.
  - Collez-les dans le répertoire de votre projet (à un endroit judicieux ...)
  - Indiquez à la *toolchain* le répertoire dans lequel se trouve le *header* ajouté (annexe « I.2.c Intégrer les fichiers d'en-tête (headers) au chemins d'inclusion »).

- Dans le `main()`, procédez à l'initialisation de l'UART.

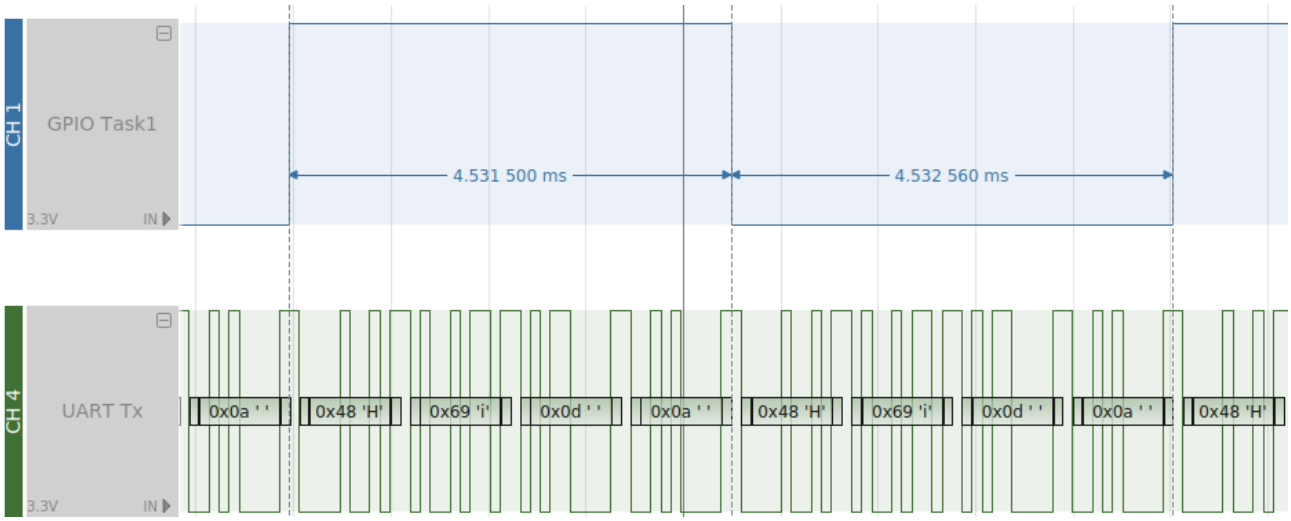
```
ENSI_UART_Init();
```

- Dans la boucle principale, envoyez une chaîne de caractères via UART, puis changez l'état d'une GPIO.

```
ENSI_UART_PutString("Hi\r\n");
HAL_GPIO_TogglePin(Task1_GPIO_Port, Task1_Pin);
```

- Confirmez la réception de la chaîne de caractères grâce à un analyseur logique, puis configurez un terminal série (Teraterm, PuTTy, minicom, …) pour disposer dès maintenant d'une interface de communication entre le STM32 et l'ordinateur.

Vous remarquerez dans la capture ci-dessous que la broche Task1 ne change d'état que quand tous les caractères (ici `"Hi\r\n"`) ont été effectivement transmis : la fonction de transmission de l'UART est donc une **fonction bloquante**.



Relevé de l'analyseur logique IKA Logic, et décodage de trame en UART (hex et ASCII).

Pour aller plus loin (1/3) :

- Testez la réception d'un caractère à la fois par l'UART. La manière la plus simple de coder un écho :
  - Effectuez la réception d'un caractère avec `ENSI_UART_GetChar()`.
  - *optionnel : passez ce caractère de minuscule en majuscule.*
  - Retransmettez le caractère : `ENSI_UART_PutChar()`.
- Sur le moniteur série, tapez un caractère. Celui-ci n'est pas affiché sur la console mais envoyé à l'UART. Ce dernier répond avec le même caractère (ou sa version majuscule) qui lui se retrouve affiché sur la console.

Pour aller plus loin (2/3) :

- Testez la réception de chaîne de caractères, toujours en codant un écho.
  - Vous ne pourrez pas taper directement de chaîne de caractères dans le terminal série, car chaque appui sur une touche conduit à l'émission du caractère seul.
  - Il est quand même possible d'envoyer une chaîne de caractère avec le terminal, en envoyant un fichier texte. Quelques fichiers tests vous sont donnés dans le répertoire de TP : `<archive TP>/resources/Misc`.
- Confirmez. Ce qui s'affiche sur le terminal est la réponse du STM32, et non le texte d'origine.

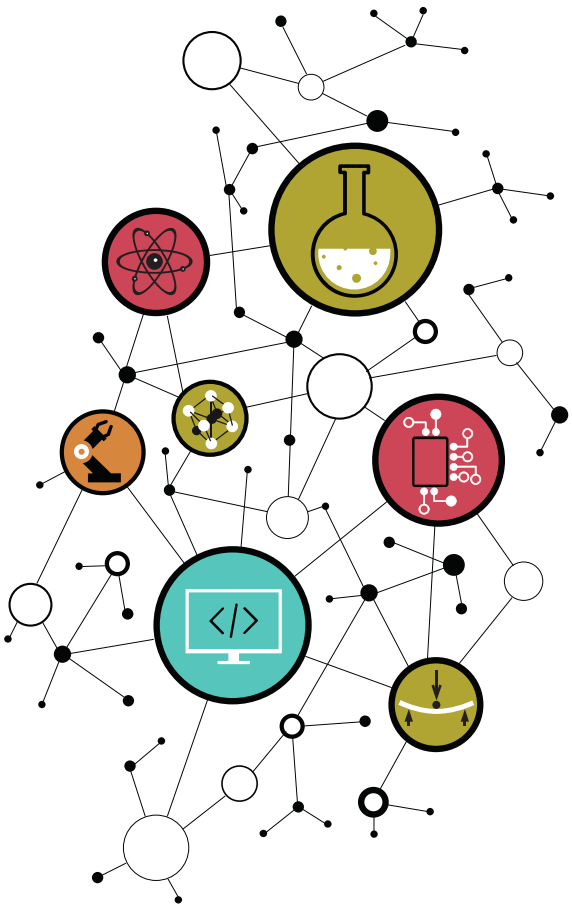
Pour aller plus loin (3/3) :

- Dans cette librairie maison, la réception s'articule autour d'un *buffer* circulaire. Rappelez comment cela fonctionne (cf. TP Systèmes Embarqués en 1A).
- Ouvrez le **Reference Manual RM0367** (*Ultra-low-power STM32L0x3 advanced Arm®-based 32-bit MCUs*) et décrivez le comportement du périphérique UART provoqué par les instructions de la fonction de transmission `ENSI_UART_PutChar()`.

Pour la suite des TP, l'utilisation de l'UART sera primordiale tandis que celle des GPIOs sera optionnelle.
L'UART, à travers le STLink VCP, permet de debugger et analyser simplement un programme sans matériel additionnel, ce qui permet de travailler hors des salles de TP. Les GPIOs apportent des informations complémentaires (principalement temporelles) mais nécessitent l'utilisation d'un analyseur logique ou un oscilloscope. Vous ne pourrez donc vous en servir qu'en salles de TP.

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 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 « III.2 Porter FreeRTOS dans un projet ».

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 en annexe « III.2.b Ajouter FreeRTOS depuis STM32CubeMX ».

- En vous aidant du chapitre « I.1 Création d'un projet » de l'annexe, créez un projet répondant aux spécifications suivantes :
  - emplacement : dans le répertoire `workspaces/` de l'archive de TP.
  - nom du projet : `votrenom_rtos02_scheduler`
  - `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*

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 « III FreeRTOS », section « III.2.a Sources et arborescence ».

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 dans l'annexe « III.2.d FreeRTOSConfig.h ».

Enfin, le fichier `Core/Src/main.c` est lui aussi fourni, et même pré-rempli avec quelques instructions que nous nous empressons d'étudier.

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

Observons 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 v1**. 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. Ordonnancement en mode coopératif

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

- Copiez-collez les fichiers de `<archive TP>/resources/rtos02/` dans le *workspace*.
  - Notez la présence de la librairie `ensi_uart` déjà utilisée.
  - Notez la présence du fichier `freertos.c`. Par défaut, CubeMX fourni ce fichier vierge (sauf commentaires) suite à la configuration de FreeRTOS. Comme il est censé contenir les fonctions propres à l'application que nous souhaitons développer, il a été partiellement complété par les enseignants en ce sens.
- 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()`.
- Dans le `main()`, initialisez le périphérique UART.

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

- 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();
```

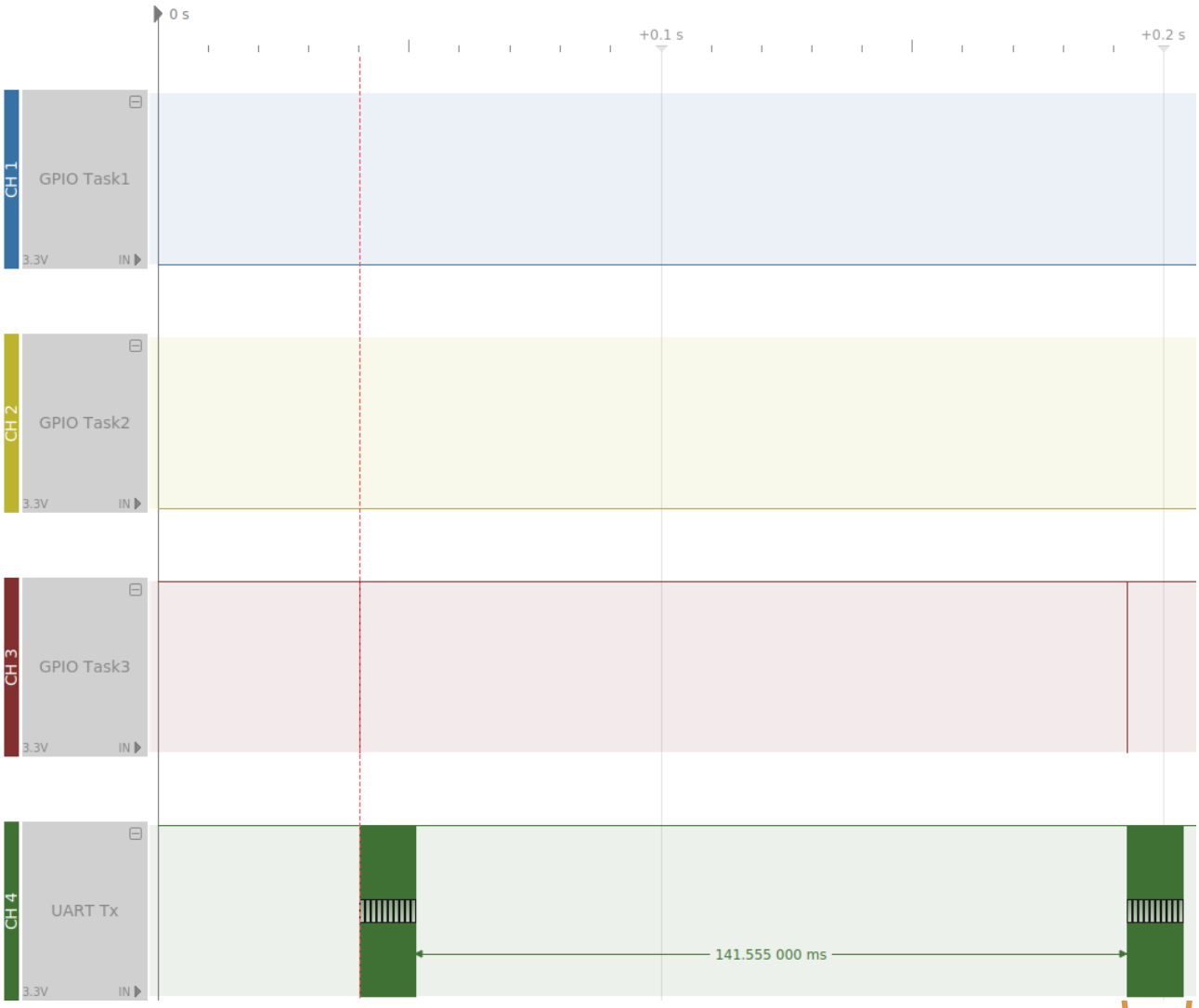
- Compilez, exécutez et analysez en visualisant sur ordinateur les données reçues par liaison série (voire observez les GPIO avec l'analyseur logique).

  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.

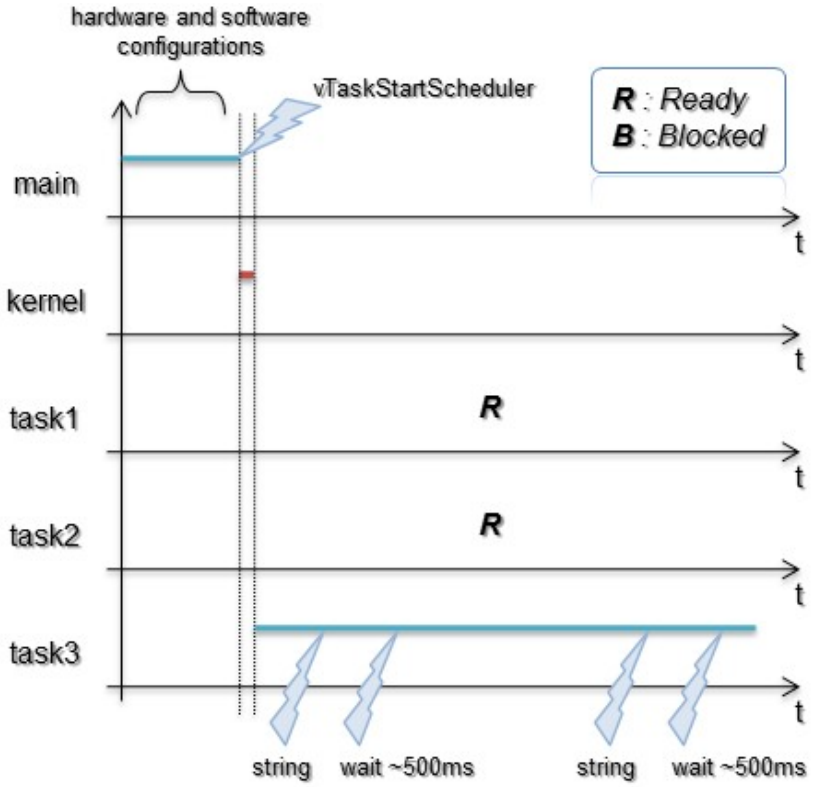


Relevé avec un terminal série observant la broche de transmission de l'UART.

```
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
```

IV.1. 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

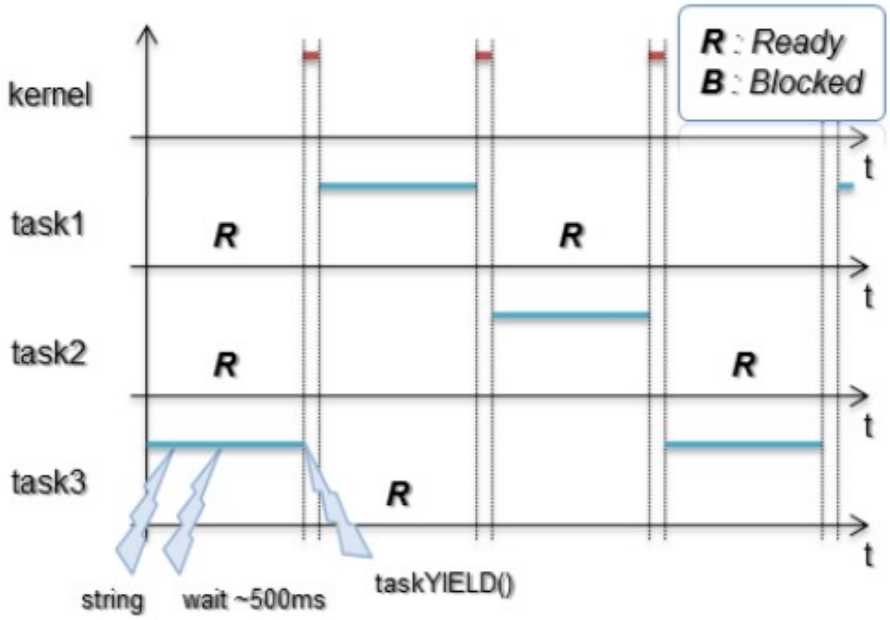
	Name	Priority (Base/Actual)	Start of Stack	Top of Stack	State
	IDLE	N/A/0	0x20000b20	0x20000cd4 <ucHeap+2876>	READY
	Tache 1	N/A/1	0x20000400	0x200005b4 <ucHeap+1052>	READY
	Tache 2	N/A/1	0x20000660	0x20000814 <ucHeap+1660>	READY
→	Tache 3	N/A/1	0x200008c0	0x20000a74 <ucHeap+2268>	RUNNING

IV.2. Commutation des tâches

Forçons maintenant les tâches à donner la main en appelant l'ordonnanceur. Dans chaque fonction de tâche, appelez la fonction suivante 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

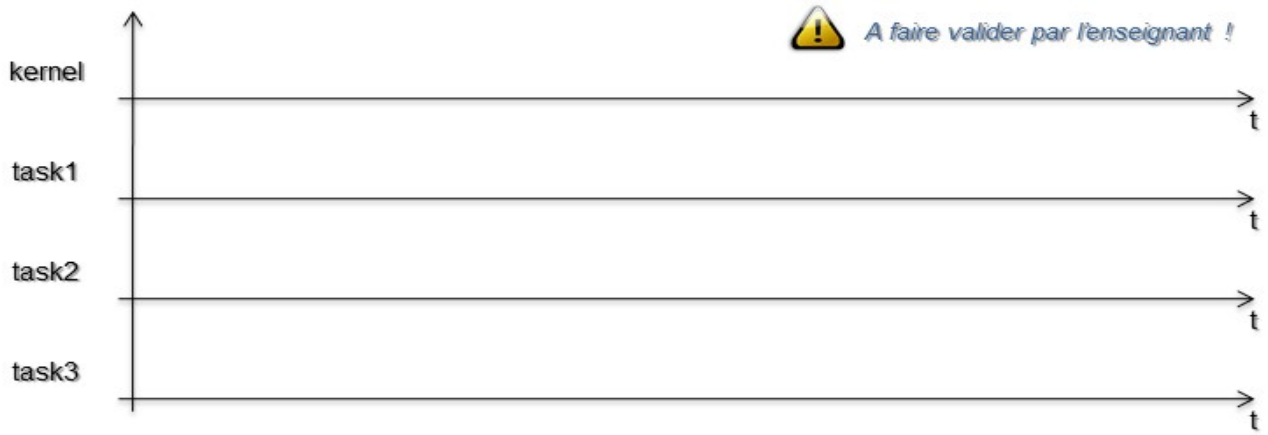
IV.3. État bloqué

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

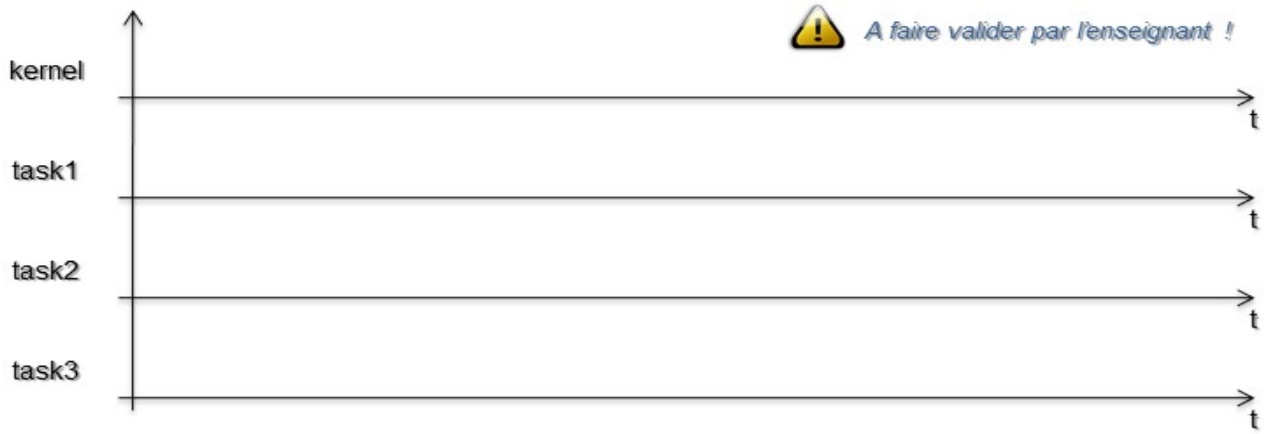
- Mettez à '1' la valeur de la macro `INCLUDE_vTaskDelay` du fichier `FreeRTOSConfig.h`. Pour rappel, ce fichier d'en-tête permet d'adapter FreeRTOS aux besoins de l'application, en n'intégrant au projet que le strict nécessaire.
- Modifiez le programme précédent en supprimant dans la tâche 1 la temporisation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

```
vTaskDelay(2000);
```

- Complétez le chronogramme ci-dessous en étant prudent aux quelques pièges pouvant apparaître. Pour information, 2000 signifie 2000 ticks donc 2 secondes dans notre cas (1 tick = 1ms, cf. FreeRTOSConfig.h). Précisez à chaque fois les appels des fonctions `taskYIELD()` et `vTaskDelay(2000)` 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 ?

IV.4. Tâche Idle

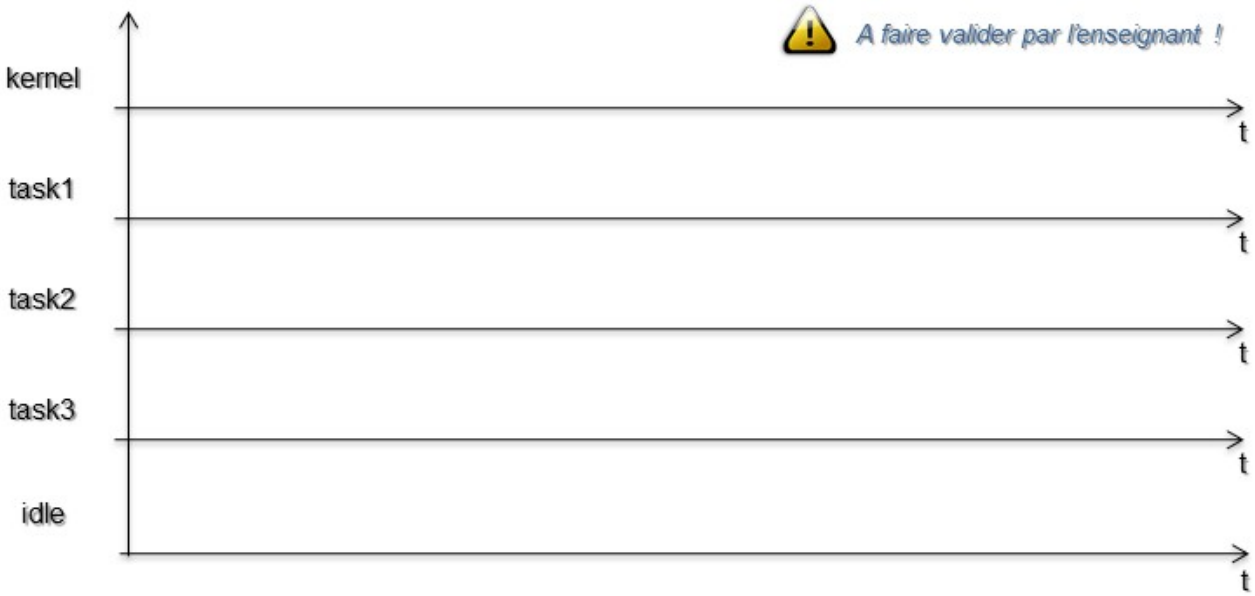
Enfin, intéressons-nous à la tâche *Idle* et découvrons comment la détourner afin d'y insérer du code utilisateur. Par défaut en mode coopératif la tâche *Idle* ne fait que forcer des commutations de contexte en appelant la fonction `taskYIELD()`.

- Définissez à '1' la macro `configUSE_IDLE_HOOK` présente dans `FreeRTOSConfig.h`.
- Cherchez le code de la tâche *Idle* dans les sources de FreeRTOS. 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 :

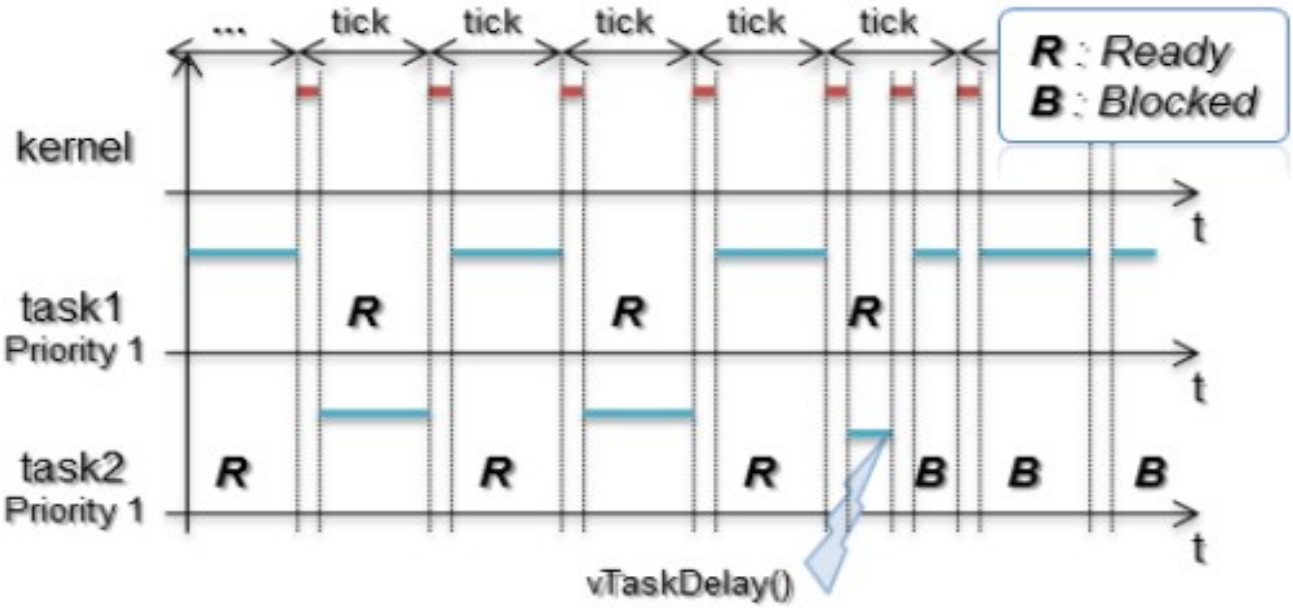


- Proposez des cas d'applications et exemples d'utilisation de la tâche *Idle*. Ne pas hésiter à s'aider du web et d'exemples de détournement de la tâche *Idle* sur d'autres noyaux temps réel.

V. Ordonnancement en mode préemptif

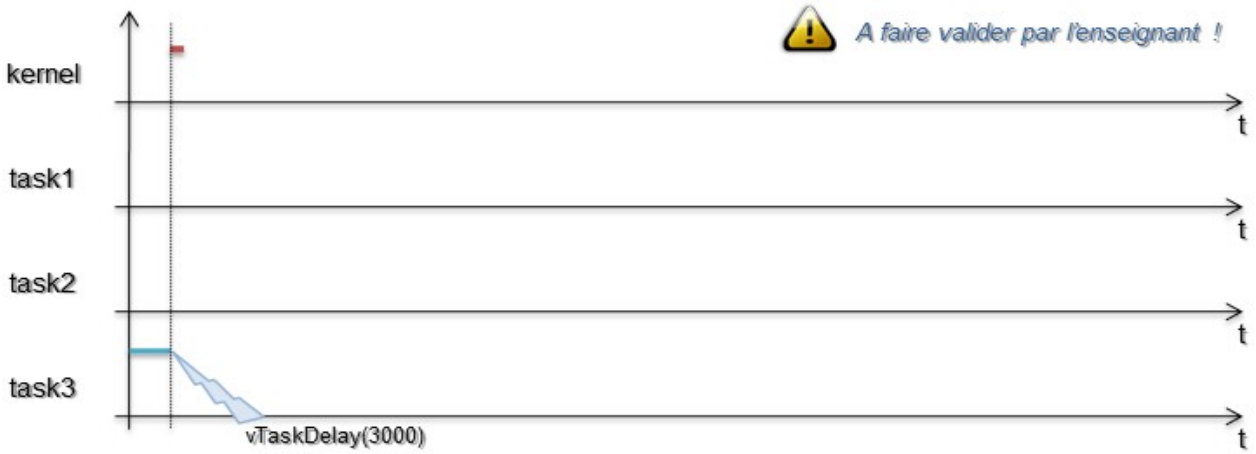
À partir de maintenant et jusqu'à la fin de la trame de TP nous travaillerons exclusivement en mode préemptif.

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`) :

```
ENSI_UART_PutString("\r\n1111111 - tick: ");
// read and send by UART current tick value
```

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

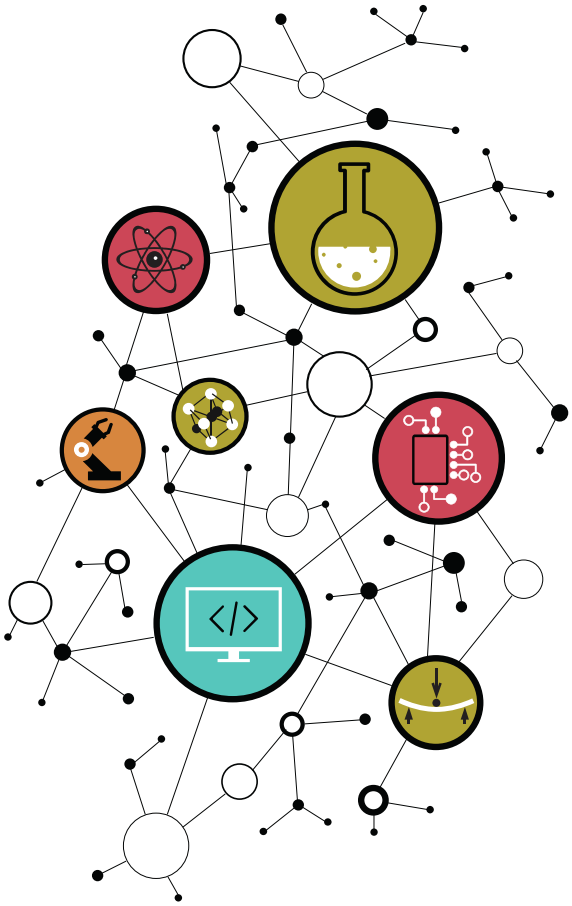
  Remplacez dans votre appel à la fonction `vTaskDelay()` par `vTaskDelayUntil()`.

- 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*) ?

Les stratégies de gestion du tas par FreeRTOS sont implémentées dans les fichiers `heap_1.c`, `heap_2.c`, `heap_3.c`, `heap_4.c` et `heap_5.c`. Ces fichiers sont présents dans les sources du *kernel* de FreeRTOS, et sont également mis à disposition dans l'archive de TP, répertoire `resources/FreeRTOS-kernel/portable/MemMang/` (*Memory Management*). De plus, le choix du fichier (et donc de la stratégie de gestion du tas) vous est laissé à la création d'un projet avec STM32CubeMX (champ « *Memory Management Scheme* »).

4. En parcourant ces fichiers (dans l'archive de TP) et la documentation associée[8] :
   - Quelles sont les différences entre les stratégies utilisant `heap_1.c` et `heap_2.c` ?
   - Quelles sont les différences entre les stratégies utilisant `heap_2.c` et `heap_3.c` ?

8 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 h*eap overflow*. 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 !

- En vous aidant du chapitre « I.1 Création d'un projet » de l'annexe, créez un projet répondant aux spécifications suivantes :
  - emplacement : dans le répertoire `workspaces/` de l'archive de TP.
  - nom du projet : `votrenom_rtos03_memory`
  - ~~`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* »
      - 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>/rtos03/` 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` 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[9].

- 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 ?

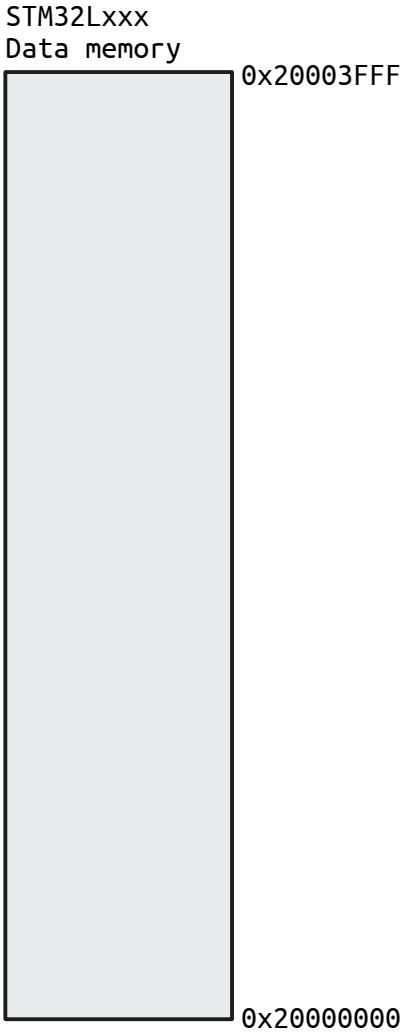
9 http://www.freertos.org/Stacks-and-stack-overflow-checking.html

- 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.

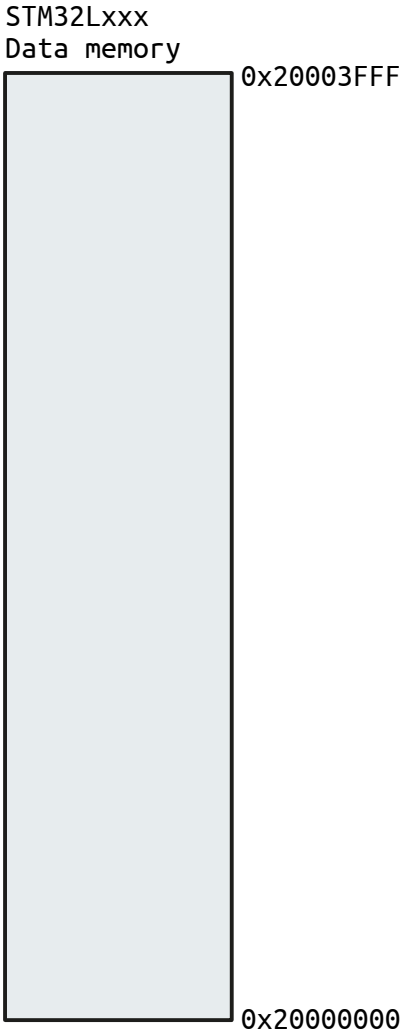
```
void growth( void ) {
    return growth();
}
```



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` 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.



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>/resources/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 l'adresse de départ de la Flash ?

- 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 ?

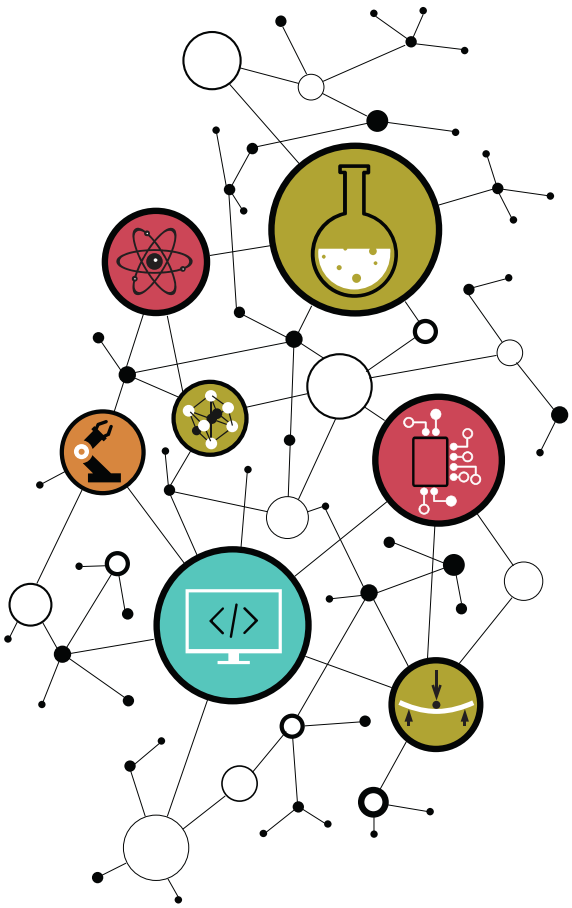
- Cherchez la mention `.bss.ucHeap`.
  - Dans quelle mémoire se situe la fonction ?
  - Quelle est l'adresse de départ de cette variable ?
  - Quelle est sa taille ?
  - Peut-on expliquer la taille de cette variable ?

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).

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.

- En vous aidant du chapitre « I.1 Création d'un projet » de l'annexe, créez un projet répondant aux spécifications suivantes :
  - emplacement : dans le répertoire `workspaces/` de l'archive de TP.
  - nom du projet : `votrenom_rtos04_tools`.
  - `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>/resources/rtos04/` dans le workspace.

- Dans `main()`, initialisez l'UART, vos tâches et supprimez la tâche créée par CubeMX.
- Dans `freertos.c` créez une tâche de priorité 2 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);
```

- Les tâches devront toutes allumer la GPIO correspondante à leur « réveil » (après une fonction bloquante) et l'éteindre avant d'appeler une fonction bloquante.

➢ 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.
- Complétez le chronogramme suivant et interprétez le résultat.

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 ?

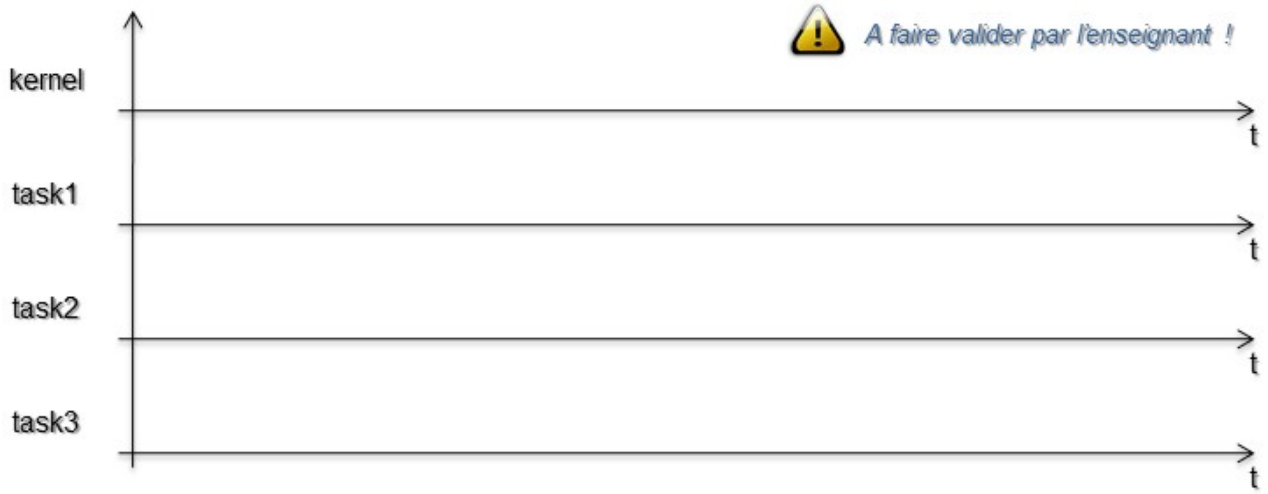
V. Protection de ressource par section critique

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()`.
- ➢ 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 ?

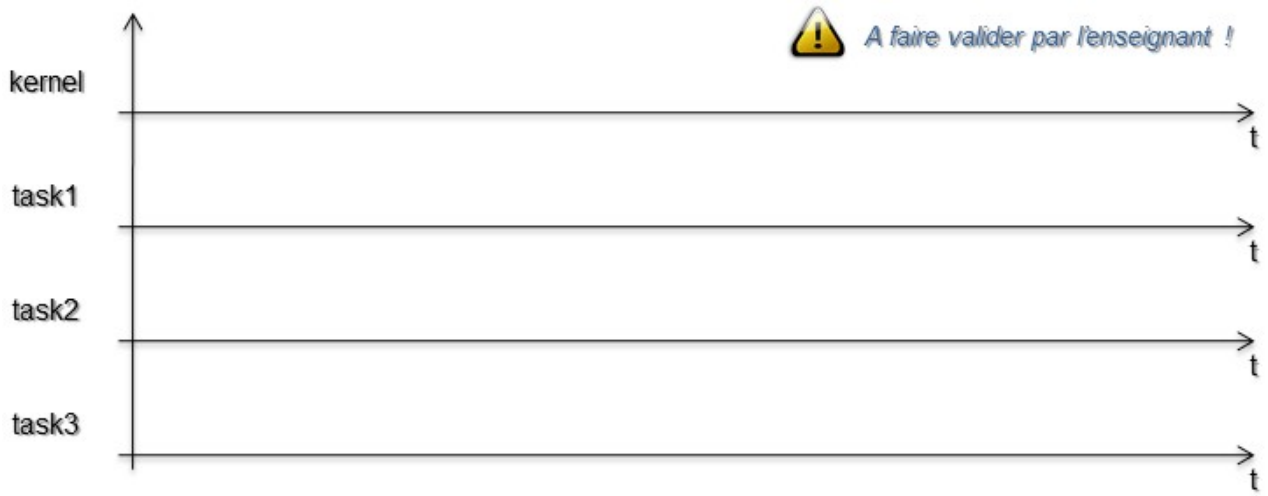
> Cette implémentation des sections critiques par FreeRTOS est assez dangereuse, notamment dans l'exemple actuellement présenté, vu que les *Ticks* (générés par timer matériel) ne sont plus vus de l'ordonnanceur pendant la durée d'exécution de la section (section longue en temps d'exécution).

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.

- Reprenez l'exercice précédent et retirez les sections critiques insérées.
- Créez un mutex qui se chargera de sécuriser la transmission de donnée par l'UART, de sorte à ce qu'une seule tâche à la fois puisse en prendre le contrôle. Attention à ne protéger que la partie critique du code (où est-elle d'ailleurs ?) !
- 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 ?

VII. Bibliothèque UART avec appels système

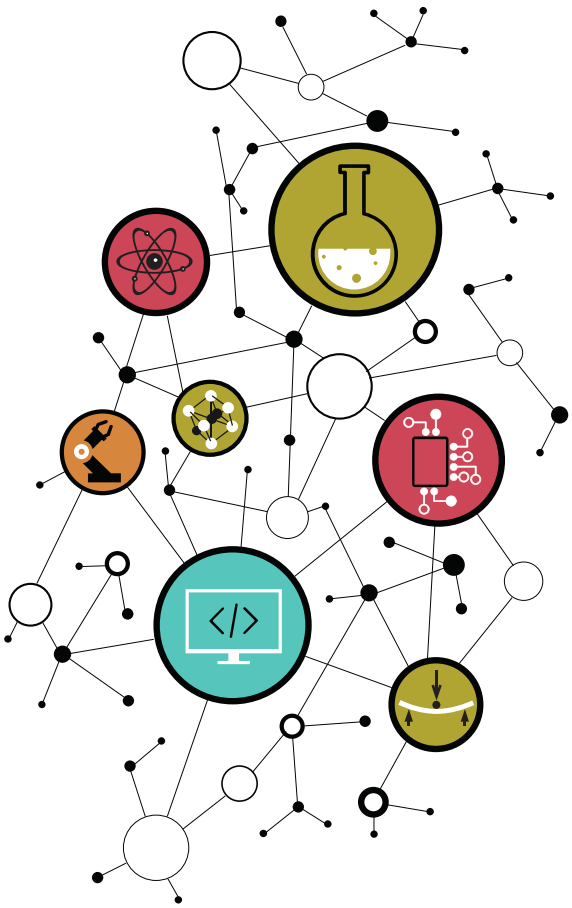
Nous allons, dans cet ultime exercice, modifier plus profondément les sources de la librairie de gestion du module UART. Le but étant d'obtenir une bibliothèque optimisée pour travailler avec FreeRTOS (pas en vitesse d'exécution mais en robustesse et efficacité).

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>/resources/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# ");
     }
}
```

Partie 5
Annexes



I. STM32CubeIDE

STM32CubeIDE est téléchargeable depuis le site STMicroelectronics et est *cross-platform* (Mac/Windows/Linux). Cette trame de TP a été validée avec la **version 1.11.0**.

https://www.st.com/en/development-tools/stm32cubeide.html

Une fois installé (pas de difficulté particulière), vous pouvez ouvrir l'IDE et en faire le tour avec une vidéo de 5 minutes, depuis STM32CubeIDE :

`Help → Tutorial Video → Discover your STM32 with SMT32CubeIDE → How to use STM32CubeIDE`

I.1. Création d'un projet

1. Dans **STM32CubeIDE** (pas STM32CubeMX) : `File → New → STM32 Project`.
2. Dans l'onglet « `Board Selector` », retrouver avec la barre de recherche la carte de TP (l'étiquette est dessus), la sélectionner et cliquer sur « `Next` ».
   - Notez qu'il est possible de sélectionner des projets exemples fournis par STMicroelectronics (avec ou sans OS).
3. Renseigner le nom[10] du projet, vérifier son emplacement, laisser les options telles quelles (`C / Executable / STM32Cube`) et cliquer sur `Finish`.
4. « `Initalize all peripherals with their default Mode ?` »

   → « `No` », pour éviter la configuration de périphériques inutiles.
5. La fenêtre de configuration graphique s'ouvre. Il s'agit d'un fichier `*.ioc`, géré par le logiciel STM32CubeMX. Dans l'onglet « `Pinout & Configuration` », configurer les composants **en fonction du cahier des charges**.
6. Une fois la configuration effectuée : `Project → Generate Code` pour générer le code correspondant aux configurations des périphériques (ou l'icône).

Le projet est créé et les instructions de configuration du micro-contrôleur sont générées (démarrage, configuration des horloges, interruptions, …). Les périphériques sélectionnés dans l'étape précédente sont également prêts à l'emploi. Pour les utiliser, voir section « I.3 HAL – Hardware Abstraction Layer » page 60.

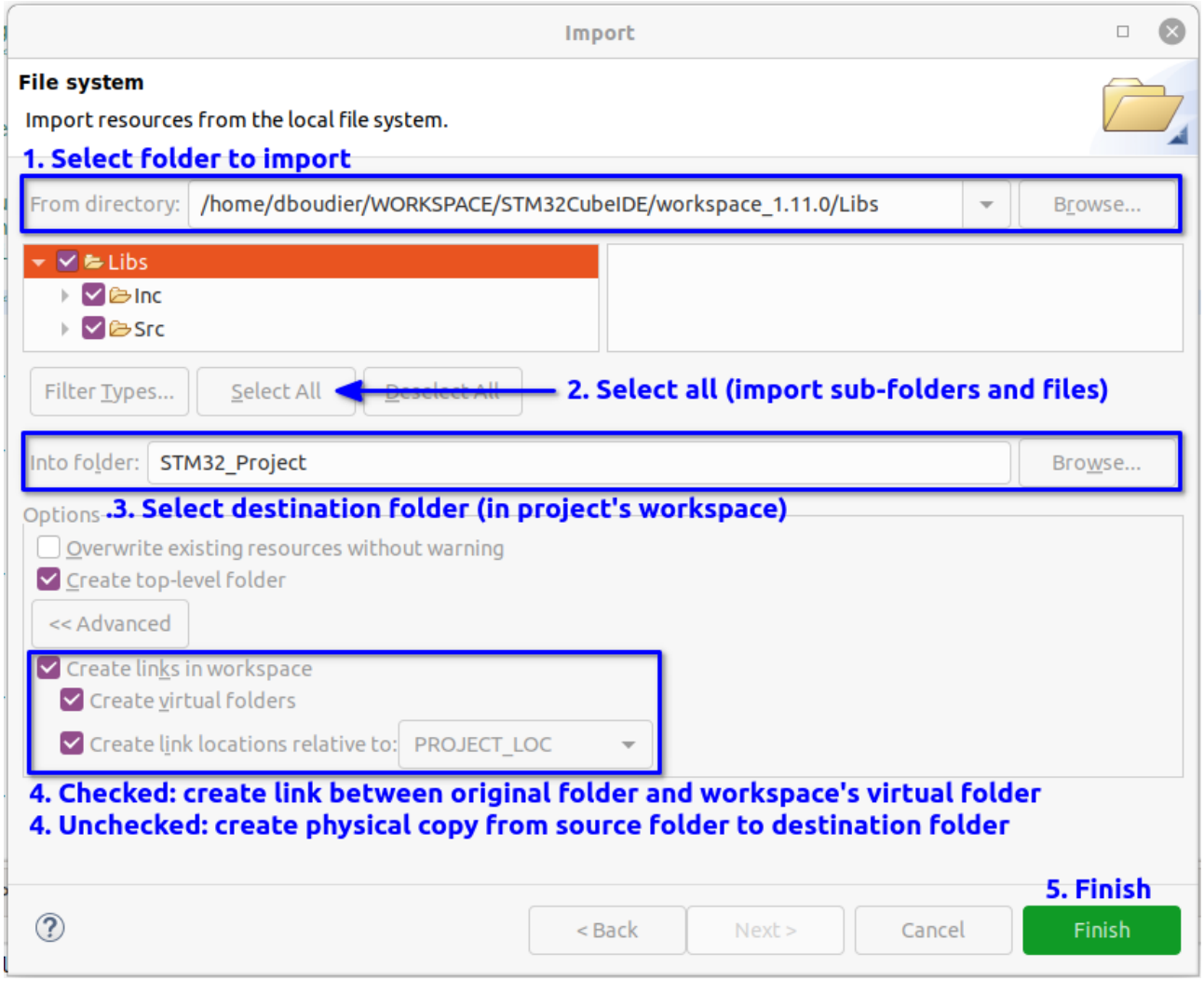
> **Attention :** Après avoir créé un projet en passant par STM32CubeMX (fichier de configuration *.ioc), faîtes attention à bien coder entre les balises de commentaires fournis par l'IDE.
> En cas de reconfiguration d'un périphérique et de re-génération du code, STM32CubeMX supprimera tout ce qui se trouve en dehors de ces balises !

10 Attention : choisir un nom sans accent, sans caractère spécial, sans espace.

I.2. Intégrer des sources existantes à un projet

I.2.a) Importer les sources dans le workspace

- Pour importer des sources (répertoire ou fichier) à un projet, on peut simplement copier-coller les sources dans un des répertoires déjà existants, depuis le système d'exploitation.
- Il est également possible de réaliser cette étape à partir de l'IDE, depuis l'explorateur de projet (onglet de gauche de l'IDE, en perspective *Edit*).
  - Clic droit sur projet `→ Import... → General → File System → Next`
  - Suivre la configuration suivante.



Le répertoire (ou le fichier) importé doit maintenant apparaître dans l'arborescence du projet, dans l'onglet de l'explorateur de projet. De plus si une copie physique a été réalisée, les fichiers se trouvent physiquement sur le disque, dans le répertoire du projet.

Remarque
Importer un répertoire à un projet n'a pour seul effet que de rendre son contenu (fichiers sources et *headers*) accessible depuis l'explorateur de projet, afin de faciliter la consultation ou l'édition de ses fichiers.

Ceci implique que les fichiers importés ne sont pas directement prêts à être utilisés par la chaîne de compilation. Selon qu'il s'agit de sources C/C++ ou de headers, deux manipulations distinctes doivent être réalisées. Celles-ci sont expliquées sur les pages suivantes.

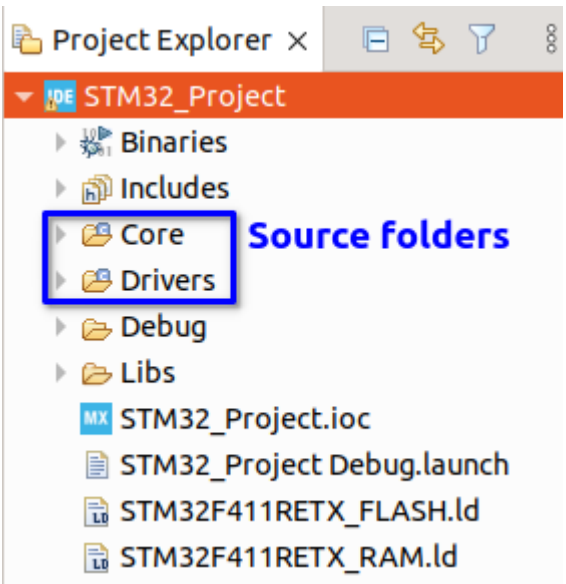
I.2.b) Intégrer les fichiers C/C++ au processus de compilation

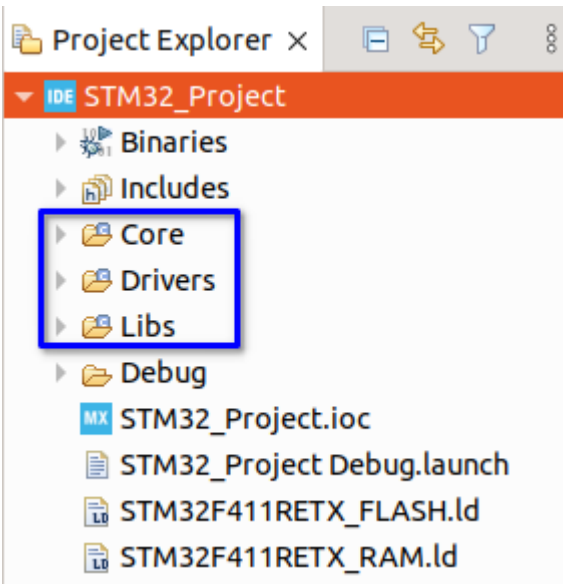
Pour être considéré comme un fichier source à compiler, le-dit fichier doit faire partie d'un « ***Source Folder*** », reconnaissable à son icône . Or par défaut un dossier importé avec la méthode précédente n'est qu'un dossier ordinaire. Mais en convertissant un dossier quelconque en **Source Folder**, les fichiers C/C++ qui en font partie sont automatiquement considérés comme sources à compiler.

Pour convertir un dossier en ***Source Folder*** :

Clic droit sur le projet → `New` → `Source Folder` → Renseigner le champ `Folder Name` ou cliquer sur `Browse` pour le trouver → `Finish`.

On observe la transformation de l'icône du répertoire `Libs`, importé au projet dans l'étape-exemple précédente.

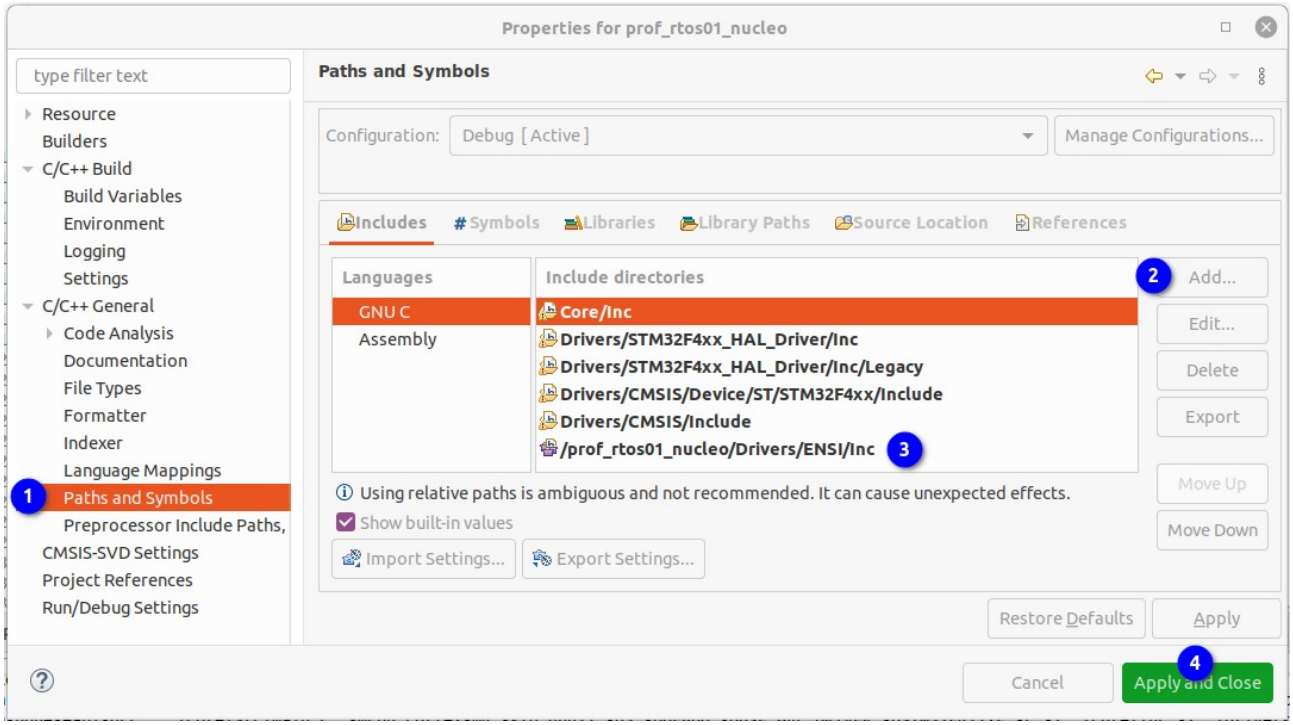




I.2.c) Intégrer les fichiers d'en-tête (headers) au chemins d'inclusion

Comme pour les fichiers sources, les *headers* doivent être connus de la chaîne de compilation (*toolchain*). Si ces fichiers ne sont pas dans un répertoire déjà référencé, alors il faut préciser à la chaîne de compilation les nouveaux chemins d'inclusion (pour la *toolchain* GCC, il s'agit de l'option `-I`).

0. Clic droit sur projet → `Properties`
1. `C/C++ General → Paths and Symbols`
2. `Add`, puis renseigner le(s) répertoire(s) contenant les *headers*
   - *Note : si le répertoire a été ajouté par lien symbolique (et non par copie physique des fichiers) alors il faut renseigner le répertoire d'origine.*
3. Exemple de chemins d'inclusion fournis par STM32CubeIDE à la création du projet et d'un chemin d'inclusion ajouté à la main.
4. Appliquer et fermer



Note : les chemins d'inclusion doivent également être spécifiés pour les fichiers assembleur, si ceux-ci utilisent les headers nouvellement intégrés.

> **Conseil**
> Ces étapes d'importation et d'inclusion sont souvent sources d'erreurs.
> Ne pas hésiter à importer et spécifier les chemins d'inclusion un à un, en compilant régulièrement le projet pour analyser les messages d'erreurs de la *toolchain*.

I.3. HAL – Hardware Abstraction Layer

I.3.a) Couches logicielles

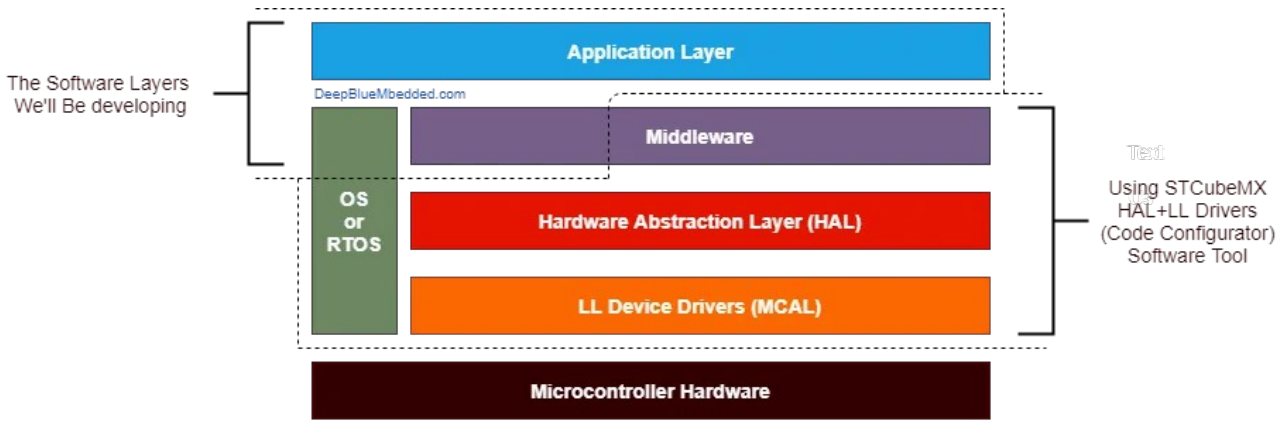
Les TP Systèmes Embarqués de première année portaient sur la programmation bas niveau (à l'étage registre) d'un micro-contrôleur (Microchip PIC18), avec pour premier objectif de produire un BSP (*Board Support Package*), puis en second objectif de développer une application (enceinte Bluetooth) en utilisant ce BSP.

Reproduire ce travail en TP RTOS serait encore plus long, étant donné la complexité d'un ARM Cortex-M en comparaison à un PIC18. Nous allons donc utiliser l'équivalent du BSP pour STM32 : la HAL et la LL. D'autant plus que ceci nous est gracieusement fourni par STM32CubeMX (cf. annexe « I.1 Création d'un projet »).

La **HAL (*Hardware Abstraction Layer*)** est une **API (*Application Programming Interface*)**. À ce titre elle fourni un jeu de fonctions au développeur, fonctions ayant pour principal objectif d'être haut-niveau et portables (entre les MCU STM32). Ainsi une application utilisant la HAL pourra tourner sur n'importe quel STM32, du L0 au F4, sans ré-écriture du code (sous réserve d'avoir le même matériel). La HAL couvre tous les périphériques matériels des STM32.

La **LL (Low-Layer)** est également une API, mais orientée registres. Elle est donc plus légère et rapide que la HAL, mais n'est valable que pour certains périphériques et n'est pas forcément portable d'un STM32 à l'autre. Elle est donc utilisée à titre d'optimisation des performances.

Ces deux API sont proposées par STMicroelectonics afin de répondre à la norme MISRA-C, norme de programmation en C pour le monde de l'automobile. Notons que l'utilisation de la HAL ou de la LL permet de grandement réduire le temps de développement, mais un développeur aguerri peut modifier ou passer outre certaines fonctions afin d'optimiser le fonctionnement de certains périphériques.



thttps://deepbluembedded.com/adding-ecual-drivers-to-your-stm32-project-configurations-options/

https://deepbluembedded.com/stm32-hal-library-tutorial-examples/

Sur la figure précédente, on peut noter que le Middleware peut en partie être généré par STM32CubeMX, tout comme le RTOS. L'application reste quant à elle entièrement entre les mains du développeur.

I.3.b) Utiliser la HAL

Prenons en exemple un extrait du « User Manual UM1749 – Description of STM32L0 HAL and Low Layer drivers »[11] :

22.2.3 Initialization and de-initialization functions

This section contains the following APIs:

- *HAL_GPIO_Init()*
- *HAL_GPIO_DeInit()*

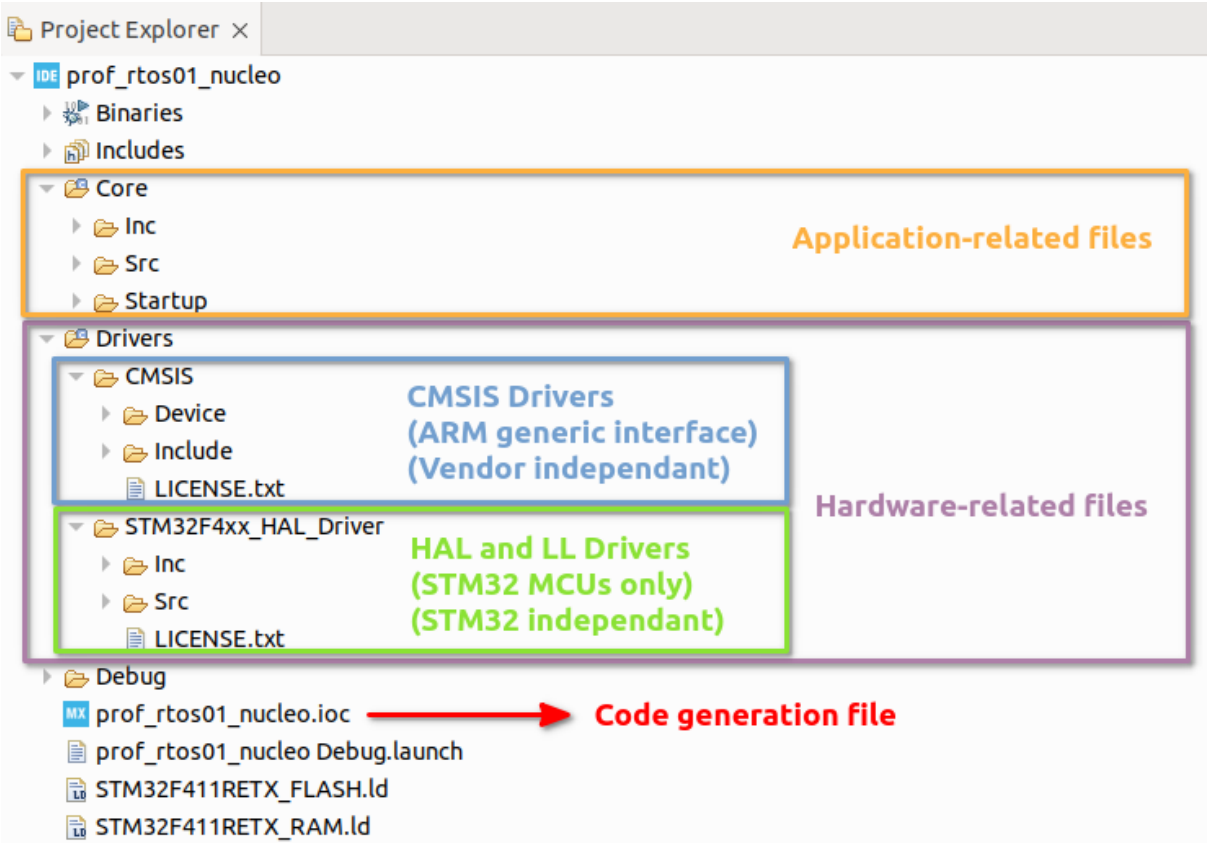
22.2.4 IO operation functions

This section contains the following APIs:

- *HAL_GPIO_ReadPin()*
- *HAL_GPIO_WritePin()*
- *HAL_GPIO_TogglePin()*
- *HAL_GPIO_LockPin()*
- *HAL_GPIO_EXTI_IRQHandler()*
- *HAL_GPIO_EXTI_Callback()*

À partir du moment où au moins une GPIO est configurée avec STM32CubeMX, alors toutes ces fonctions sont définies et fournies dans les fichiers du projet. Ainsi, des fonctions similaires sont définies pour l'UART, timer, … et tout autre périphérique configuré.

Dans l'arborescence du projet, toutes les fonctions de la HAL se trouvent déclarées et définies dans le répertoire `<Project>/Drivers/STM32xxxx_HAL_Driver`. En parcourant les fichiers, il reste à regarder les fonctions disponibles et comment s'en servir pour les appeler depuis les fichiers sources de l'application.

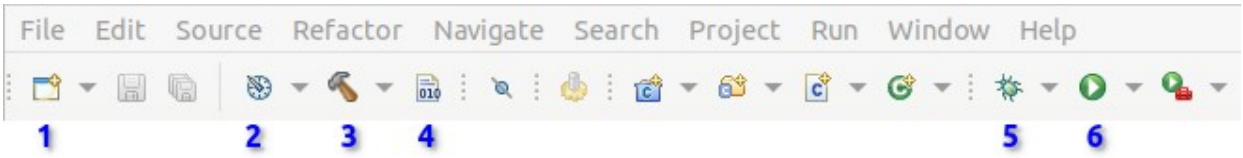


11 https://www.st.com/en/embedded-software/stm32cubel0.html#documentation

I.4. Programmer et debugger avec STM32Cube

I.4.a) En bref

Faisons simple : ces menus et boutons sont les plus importants de STM32CubeIDE. Les deux que vous utiliserez le plus sont la compilation (Build All) et le mode debug.



1. Nouveau projet (File → New → STM32 Project) [Alt+Shift+N]
2. Configuration (mode Debug / mode Release)
3. Build current configuration of current project
4. **Build all (Project → Build All) [Ctrl+B]**
5. **Debug current project (Run → Debug) [F11]**
6. Run current project (Run → Run) (programmation et exécution de la cible)

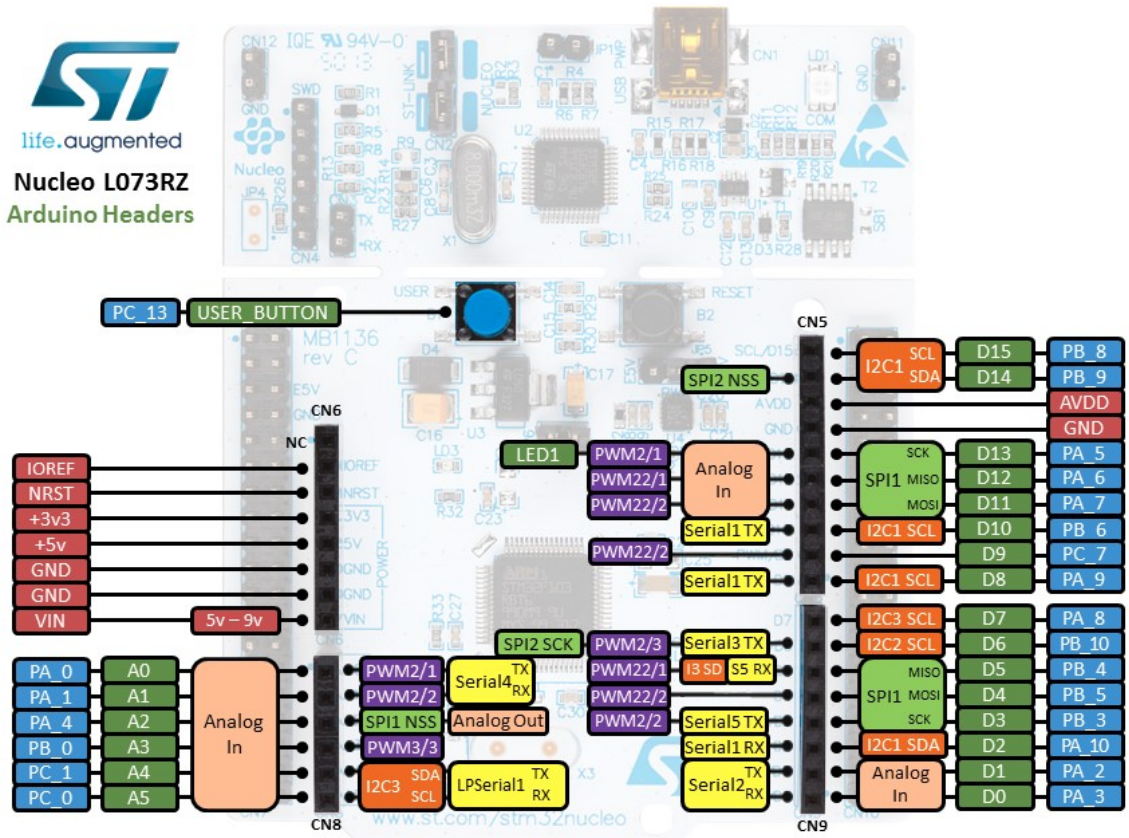
I.4.b) Debugger

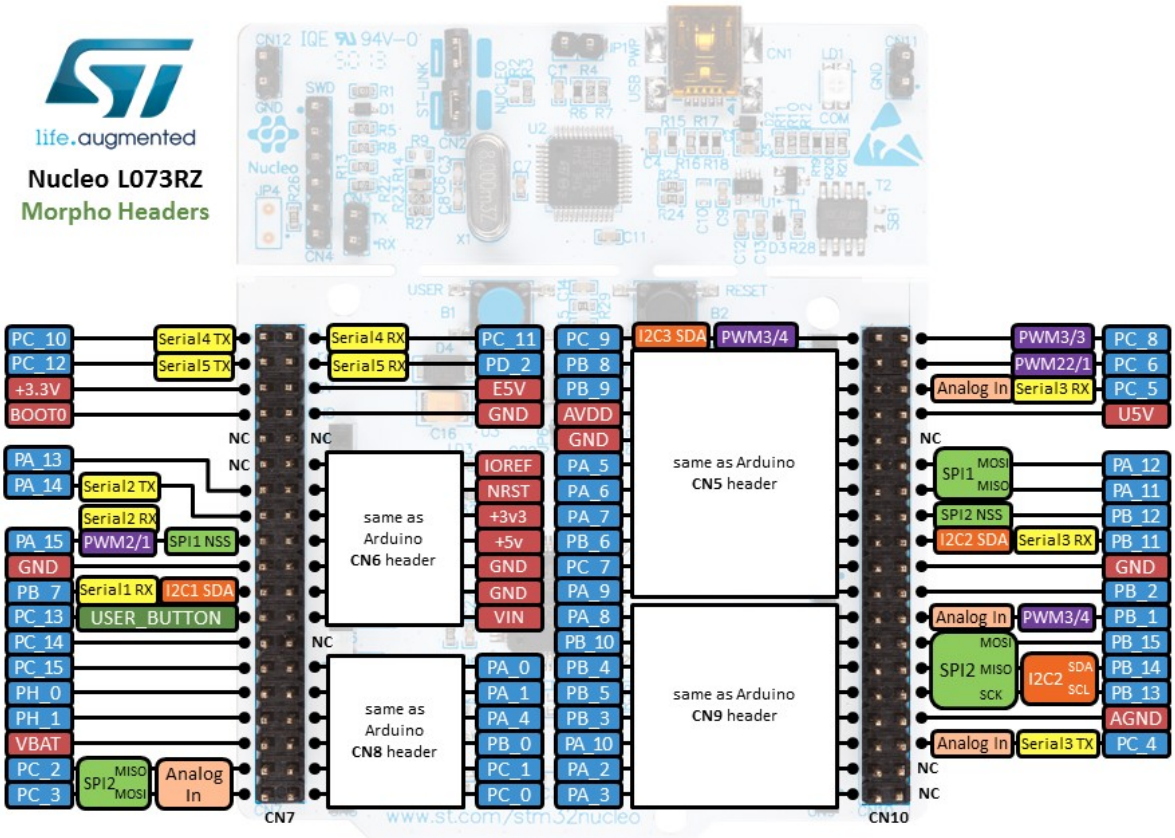
La première fois que vous lancez le debugger, la fenêtre « Edit Configuration » s'affiche. Cliquez sur OK. En mode debug, l'IDE change de perspective (la configuration des fenêtres et menus change). Il est évidemment possible de revenir à la perspective d'édition en arrêtant le mode debug.

1. Reset du composant et de la session de debug
2. Le debugger s'arrête / ne s'arrête pas aux breakpoints
3. Arrêt et redémarrage
4. Démarre / continue [F8]
5. Pause
6. Stop (arrête la session de debug, revient au mode édition)
7. Step into : passe à l'instruction suivante
8. Step over : passe à la ligne suivante
9. Step return : continue jusqu'à quitter la fonction en cours

II. Carte Nucleo-L073RZ

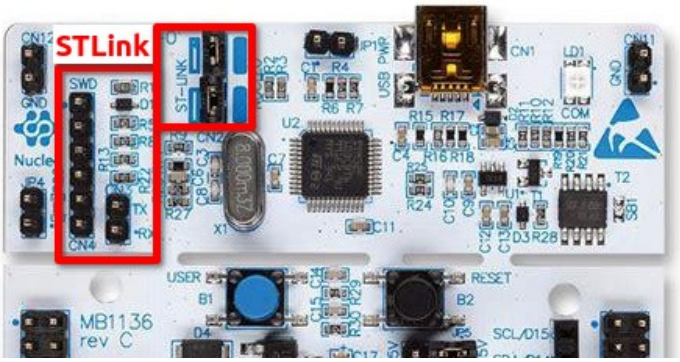
II.1. Pinout diagram





II.2. UART et STLink Virtual Com Port

Par défaut sur les cartes Nucleo-64, les broches du périphérique UART2 du MCU ne sont pas physiquement reliées aux connecteurs présents sur la carte. En effet, elles sont à la place reliées au **STLink *Virtual Com Port* (VCP)**, encadré en rouge sur la figure ci-dessous (connecteurs CN2, CN3 et CN4). Le VCP une est liaison série émulée à travers le port USB de la Nucleo-64, c'est la raison pour laquelle une communication en UART est possible entre un ordinateur et la carte sans aucun matériel supplémentaire.



Il est possible d'utiliser n'importe quelle autre liaison série avec les connecteurs, tant que les broches associées ne sont pas déjà utilisées par d'autres périphériques. Par exemple, l'UART1 utilise par défaut les broches PA10 et PA9 pour Rx et Tx.

Toutefois il peut arriver (selon le cahier des charges et les broches disponibles) que le choix du périphérique de liaison série se porte obligatoirement sur l'UART2. Auquel cas nous avons toujours la possibilité de modifier la Nucleo-64 :

- Dessouder les *solder bridges* (résistance 0 Ω) SB13 et SB14 pour déconnecter les broches du MCU de celles du STLink
- Souder les *solder bridges* SB62 et SB63 pour relier les broches du MCU aux connecteurs Morpho et Arduino.

Cette manipulation est évidemment réversible et sans danger (sauf pour vos doigts).

Les documents utiles sont donnés dans le répertoire `doc/` de l'archive de TP :

- `en.MB1136-DEFAULT-<C03/C04/C05>_Schematic` (le numéro est indiqué sur la carte).
- UM1724 - STM32 Nucleo-64 boards (MB1136).

III. FreeRTOS

III.1. Introduction

FreeRTOS est un système d'exploitation temps-réel sous licence MIT et ayant été racheté par Amazon en 2017.

Le site de FreeRTOS est très complet et propose plusieurs documentations différentes (https://www.freertos.org/RTOS.html) :

- **Developper Docs :** de la documentation plutôt généraliste (applicable à d'autres OS) sur les concepts et mécanismes liés aux RTOS. Considérez cette partie comme un cours rapide sur les RTOS.
- **Secondary Docs :** de la documentation un peu plus poussée, et en même temps un peu plus centrée sur FreeRTOS.
- **API Reference :** la documentation sur l'API (les fonctions et services) proposés par FreeRTOS.

III.2. Porter FreeRTOS dans un projet

III.2.a) Sources et arborescence

Les sources sont accessibles directement par le site de FreeRTOS :

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

Le site propose généralement deux versions en téléchargement : la dernière ***release*** en date et la version **LTS (*Long Term Support*)**. La dernière *release* a pour avantage de contenir les nouveautés de l'OS mais n'est maintenue que jusqu'à la sortie de la version suivante. La version LTS est potentiellement plus ancienne (de quelques mois) mais le support (corrections de bugs critiques et sécurité) est garanti pendant deux ans.

Dans les deux cas, la version téléchargée contient non seulement les sources du *kernel*, mais également des sources AWS pour l'IoT (suite au rachat par Amazon) et d'autres librairies (HTTP, TCP, …).

La même page Web propose aussi une redirection vers GitHub, qui offre la possibilité de ne télécharger que les sources du *kernel* (https://github.com/FreeRTOS/FreeRTOS-LTS). C'est ceci qui a été téléchargé et fourni dans l'archive de TP.

Si on se concentre uniquement sur les sources du *kernel*, l'arborescence est donnée sur la page suivante. On y distingue deux ensembles de fichiers.

Les **fichiers architecture-agnostiques** (situés à la racine et dans `include/`) sont génériques et utilisés quelle que soit l'architecture du MCU cible. Certains fichiers peuvent ne pas être intégrés au projet tant que les fonctionnalités ne sont pas utilisées (par exemple `queue.c`, …) .

Les **fichiers architecture-spécifiques** (situés dans `portable/`) sont eux propres à l'architecture cible. Certains fichiers sont mêmes en assembleur. Ces fichiers doivent être intégrés au projet car ils contiennent les définissions de fonctions appelées par les sources génériques (comme par exemple la configuration du *timer* pour l'ordonnanceur). Ce répertoire contient des sources pour plus de 40 architectures différentes[12] (la représentation page suivante a été allégée pour des raisons de lisibilité). Les sources sont classées par *toolchain* (`GCC` dans notre cas), puis par architecture cible (`ARM_CM0` pour Cortex-M0).

Toujours dans `portable/`, le répertoire `MemMang/` **(*Memory Management*)** ne correspond pas à une *toolchain* mais contient 5 fichiers correspondant chacun à une stratégie de gestion du tas (allocation, désallocation, ...). Un de ces fichiers doit être choisi et intégré au projet.

12 https://freertos.org/RTOS_ports.html

```
└── FreeRTOS-kernel/
    ├── include/
    │   ├── deprecated_definitions.h
    │   ├── queue.h
    │   ├── event_groups.h
    │   ├── timers.h
    │   ├── StackMacros.h
    │   ├── portable.h
    │   ├── mpu_prototypes.h
    │   ├── stream_buffer.h
    │   ├── stack_macros.h
    │   ├── atomic.h
    │   ├── task.h
    │   ├── stdint.readme
    │   ├── projdefs.h
    │   ├── mpu_wrappers.h
    │   ├── list.h
    │   ├── message_buffer.h
    │   ├── croutine.h
    │   ├── semphr.h
    │   └── FreeRTOS.h
    ├── croutine.c
    ├── tasks.c
    ├── timers.c
    ├── README.md
    ├── queue.c
    ├── event_groups.c
    ├── CMakeLists.txt
    ├── LICENSE.md
    ├── list.c
    ├── stream_buffer.c
    └── portable/
        ├── GCC/
        │   ├── ARM_CM4F/
        │   │   ├── portmacro.h
        │   │   └── port.c
        │   └── ARM_CM0/
        │       ├── portmacro.h
        │       └── port.c
        ├── CCS/
        ├── MemMang/
        │   ├── heap_3.c
        │   ├── ReadMe.url
        │   ├── heap_1.c
        │   ├── heap_4.c
        │   ├── heap_5.c
        │   └── heap_2.c
        └── MPLAB/
            ├── PIC32MZ/
            ├── PIC18F/
            └── PIC32MX/
```

Headers (fichiers d'en-tête) associés aux sources ci-dessous

Sources du kernel FreeRTOS

Sources liées au hardware, classées par *toolchain* puis par cœur *(bien élaguées pour faciliter la lecture)*

Fichiers définissant la stratégie de gestion du tas (choisir 1 fichier pour un projet)

Sources liées au hardware, classées par *toolchain* puis par cœur *(bien élaguées pour faciliter la lecture)*

Partie architecture-agnostique du noyau (indépendante de la cible)

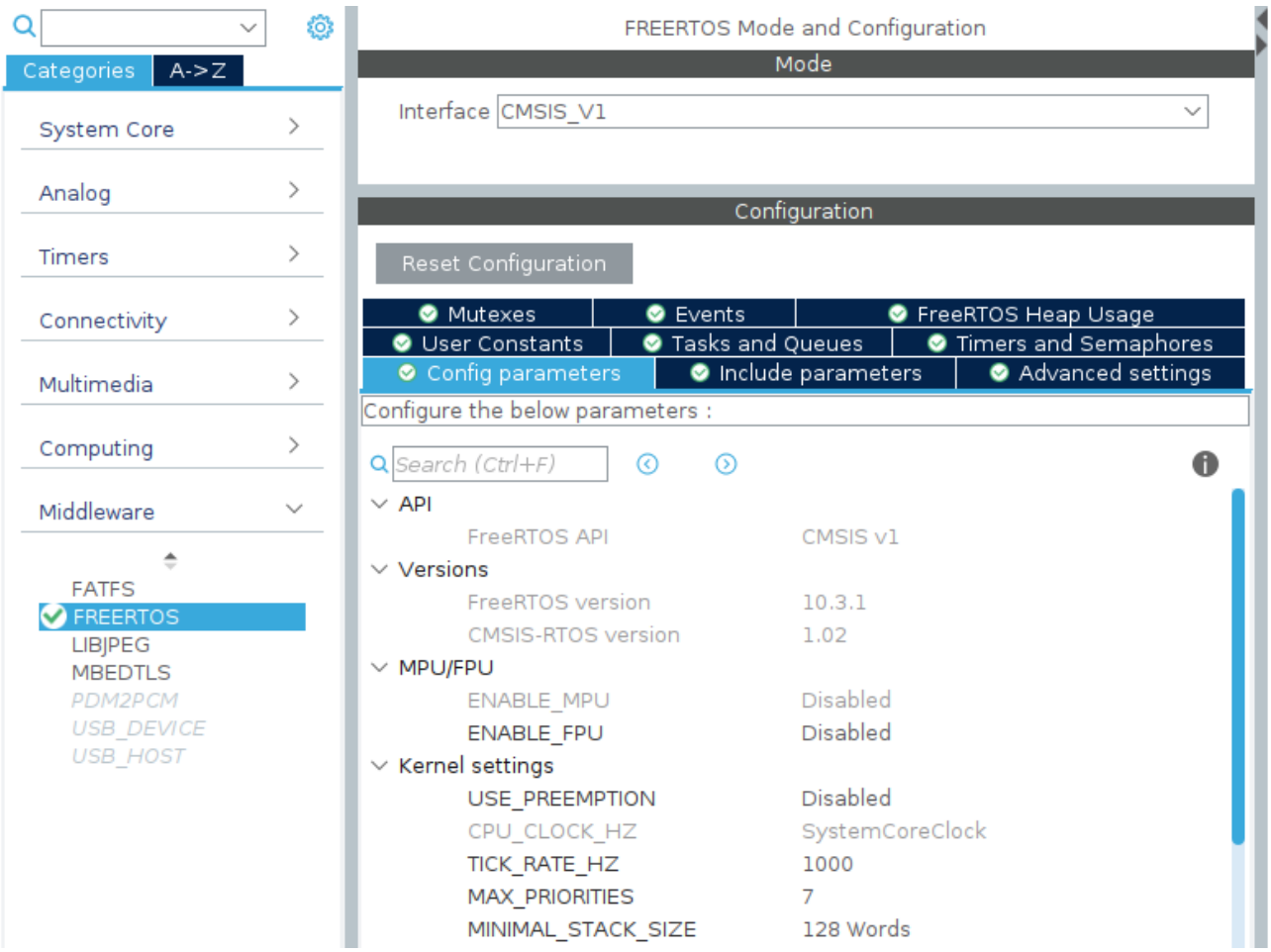
Partie architecture-spécifique du noyau (fichiers en fonction de la cible)

III.2.b) Ajouter FreeRTOS depuis STM32CubeMX

FreeRTOS peut directement être intégré à un projet STM32Cube grâce à l'outil de configuration SMT32CubeMX, soit lors de la création du projet, soit en ouvrant le fichier `*.ioc` associé et en re-générant le code.

Les champs situés dans les onglets « *Config parameters* » et « *Include parameters* » correspondent aux définitions des macro-constantes du fichier `FreeRTOSConfig.h`. Ces macros restent directement éditables dans le fichier, mais elles seront redéfinies en cas de re-génération du code par CubeMX.

De la même manière les tâches, sémaphores, mutex et autres peuvent directement être créés depuis cette fenêtre de configuration. Toutefois, il faut se rappeler que toute configuration avec CubeMX sera réalisée par un *wrapping* de l'API FreeRTOS par l'API CMSIS-RTOS (v1 ou v2).



Bref : *clic & select* → *Generate code* → Ding, c'est prêt !

Vous remarquerez la création d'un répertoire `Middlewares/Third_Party/FreeRTOS/` qui contient les sources du kernel de FreeRTOS, désormais inclus au projet.

III.2.c) Ajouter manuellement les fichiers à un projet

Il est également possible d'intégrer à un projet les sources de FreeRTOS sans passer par CubeMX. Pour savoir quels fichiers de `FreeRTOS-Kernel/` doivent être intégrés au projet, poursuivez la lecture de cette section. Pour savoir comment les intégrer au projet, reprenez la section « I.2 Intégrer des sources existantes à un projet » de l'annexe.

Parmi les fichiers des sources (à la racine de `FreeRTOS-Kernel/`), seuls les fichiers C utilisés peuvent être intégrés au projet. Les fichiers dans `include/` doivent être importés (avec les chemins d'inclusion précisés à la *toolchain*).

Parmi les fichiers du répertoire `portable/`, il ne faut sélectionner que le répertoire correspondant à la fois à la toolchain et au processeur cible. Dans notre cas, il s'agit respectivement de `GCC` (GNU's Compiler Collection) et `ARM_CM0` (ARM Cortex-M0). Les autres répertoires ne doivent pas être intégrés au projet (sauf point suivant).

Enfin, un fichier de `portable/MemMang/` doit obligatoirement être intégré au projet, suivant la stratégie que l'on souhaite employer pour la gestion du tas.

En bref :

- `FreeRTOS-Kernel/`
  - fichiers C : prendre uniquement les fichiers nécessaires
  - `include/` : tout inclure
  - `portable/MemMang/` : prendre uniquement le `heapX.c` que l'on souhaite
  - `portable/GCC/ARM_CM0/` : tout inclure
  - `portable/` : ne prendre aucun des autres répertoires

III.2.d) FreeRTOSConfig.h

Le fichier `FreeRTOSConfig.h` est un fichier ne comportant que des macro (pas de déclaration de fonction). Il s'agit du seul fichier physiquement sorti du système de fichiers de FreeRTOS, et est par défaut intégré avec les fichiers liés à l'application. Ce fichier est essentiel et assure la configuration du noyau avant compilation.

Par exemple, de nombreuses macros sont préfixées par `config`. Il s'agit donc des constantes permettant de configurer FreeRTOS dans un mode de fonctionnement propre à l'application. On peut par exemple choisir le mode de l'ordonnanceur (coopératif ou préemptif), la fréquence d'interruption de l'ordonnanceur, les modes d'allocations, la taille du tas ou des piles, … Autant de paramètres obligatoires mais ajustables.

Un autre ensemble de macros, celles préfixées par `INCLUDE_`, permet de sélectionner les fonctionnalités intégrées (ou non) à FreeRTOS. Mettre les macros à '0' permet donc de désactiver certaines fonctionnalités (fonctions de l'API), même si les fonctionnalités élémentaires (création de tâches, démarrage du *scheduler*, …) sont toujours présentes. L'idée est ne compiler, linker et écrire dans la mémoire programme du processeur que les fonctions utilisées dans l'optique de réduire à la fois le temps de compilation et la taille du programme binaire.

```c
/*
 * FreeRTOS Kernel V10.3.1
 * [...]
 * See http://www.freertos.org/a00110.html
 *----------------------------------------------------------*/

#ifndef FREERTOS_CONFIG_H
#define FREERTOS_CONFIG_H

#define configENABLE_FPU                        0
#define configENABLE_MPU                        0

#define configUSE_PREEMPTION                    0
#define configSUPPORT_STATIC_ALLOCATION         0
#define configSUPPORT_DYNAMIC_ALLOCATION        1
#define configUSE_IDLE_HOOK                     1
#define configUSE_TICK_HOOK                     0
#define configCPU_CLOCK_HZ                      ( SystemCoreClock )
#define configTICK_RATE_HZ                      ((TickType_t)1000)
#define configMAX_PRIORITIES                    ( 7 )
#define configMINIMAL_STACK_SIZE                ((uint16_t)128)
#define configTOTAL_HEAP_SIZE                   ((size_t)15360)
#define configMAX_TASK_NAME_LEN                 ( 16 )
#define configUSE_16_BIT_TICKS                  0
#define configIDLE_SHOULD_YIELD                 0
#define configQUEUE_REGISTRY_SIZE               8
#define configENABLE_BACKWARD_COMPATIBILITY     0
#define configUSE_PORT_OPTIMISED_TASK_SELECTION 1

/* Co-routine definitions. */
#define configUSE_CO_ROUTINES                   0
#define configMAX_CO_ROUTINE_PRIORITIES         ( 2 )

/* Set the following definitions to 1 to include the API function, or zero
to exclude the API function. */
#define INCLUDE_vTaskPrioritySet            1
#define INCLUDE_uxTaskPriorityGet           0
#define INCLUDE_vTaskDelete                 1
#define INCLUDE_vTaskCleanUpResources       0
#define INCLUDE_vTaskSuspend                0
#define INCLUDE_vTaskDelayUntil             0
#define INCLUDE_vTaskDelay                  1
#define INCLUDE_xTaskGetSchedulerState      1
#define INCLUDE_xTaskResumeFromISR          0


#define configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY 5
#define configCHECK_FOR_STACK_OVERFLOW               2
#define configUSE_MALLOC_FAILED_HOOK                 1

#endif /* FREERTOS_CONFIG_H */
```

PRISE DE NOTES

PRISE DE NOTES

PRISE DE NOTES

PRISE DE NOTES

PRISE DE NOTES

PRISE DE NOTES

PRISE DE NOTES