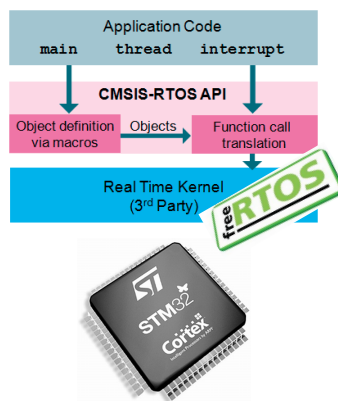




Systèmes Temps Réel

- Travaux pratiques -

Basile Dufay*, Christophe Cordier†



Attention !

Pour chaque TP, des travaux préparatoires sont nécessaires. Ces préparations doivent être effectuées **AVANT** l'arrivée en salle et seront contrôlées par vos enseignants en cours de séance.

Toolchain :

Logiciels	Version
Paquetages	
STM32CubeIDE	1.10.1
CMSIS	5.0.8
GCC arm atolllic eabi	10.6.2
gdbserver for ST-Link	7.0.0
STM32CubeMX	6.6.1
STM32L1 pack	1.10.3
FreeRTOS	10.0.1
TraceAnalyser	4.6.6

Version 3.0 - 2022-2023

*basile.dufay@unicaen.fr - GREYC Électronique - Bureau FB133, ENSICAen Bât. F - 02 31 45 26 91

†christophe.cordier@unicaen.fr - GREYC Électronique - Bureau FB134, ENSICAen Bât. F

Table des matières

1. Mode coopératif et état d'une tâche	3
2. Mode préemptif et gestion mémoire	11
3. Outils de Communication/Synchronisation	18
4. Projet	25
Annexes - un OS temps réel en pratique	31

TP 1.

Mode coopératif et état d'une tâche

La partie centrale d'un OS est l'ordonnanceur (**scheduler**), chargé de partager le temps CPU entre les différentes tâches (ou *processus*, ou *threads*) qui évoluent en parallèle. C'est lui qui établit l'emploi du temps du CPU, garant d'un fonctionnement "multi-tâches". Un paramètre essentiel permettant au scheduler de faire ses choix est l'**état** de chaque tâche. Toutes les tâches n'ayant pas les mêmes besoins au même moment, l'ordonnanceur va leur attribuer un état afin d'en optimiser le traitement.

Dans ce 1^{er} TP, nous allons nous intéresser au mode coopératif (en opposition au mode préemptif), dans lequel les différentes tâches coopèrent de façon explicite afin de partager le temps CPU. Dans la pratique, ce mode est très peu utilisé pour des problèmes de robustesse et d'égalité de temps de partage CPU. Ceci sera illustré dès le premier exercice.

Table des matières

1. Travail préparatoire (à rendre via moodle)	4
2. Exo - Interface de communication	5
2.1. Mise en place de l'environnement de travail	5
2.2. Compréhension de la bibliothèque uart alternative	5
3. Exo - mode Coopératif	6
3.1. Mise en place de l'environnement de travail	6
3.2. Premiers pas dans un environnement multitâche	6
3.3. État bloqué	8
3.4. Tâche Idle	10

1. Travail préparatoire (à rendre via moodle)

En vous aidant de la documentation en ligne proposée sur le site de *FreeRTOS* (<http://www.freertos.org>), répondez aux questions suivantes portant sur les outils et services de base proposés par les systèmes d'exploitation (OS) au sens large et plus particulièrement *FreeRTOS*.

- ✎ Quels sont les états que peut prendre une tâche sous *FreeRTOS* ?
- ✎ Ces états sont-ils les mêmes quelque soit l'OS utilisé ? Donner un exemple.
- ✎ Que se passe-t-il lorsque toutes les tâches préalablement créées sont à l'état bloqué ?
- ✎ En vous aidant de la documentation en ligne de l'API CMSIS-RTOS proposée sur le site <http://www.keil.com/pack/doc/CMSIS/RTOS/html/index.html> ainsi que de celle de *FreeRTOS* (<http://www.freertos.org>), donnez le prototype de la fonction permettant de créer une tâche et précisez le rôle des différents paramètres d'entrée.
- ✎ Donnez celui de la fonction permettant de démarrer le scheduler (OS).

```

1 [...]
2 MESN_UART_Init();
3 uint8_t tempTab[20];
4 MESN_UART_PutString_Poll((uint8_t*)"r\nEntrez du texte puis terminez par ENTREE:");
5 /* Infinite loop */
6 while (1) {
7     MESN_UART_GetString(tempTab);
8     MESN_UART_PutString_Poll(tempTab);
9     MESN_UART_PutString_Poll((uint8_t*)"r\n");
10 }

```

Listing 1 – Lignes de code permettant de tester le fonctionnement de la bibliothèque UART alternative.

2. Exo - Interface de communication

Préalablement à la découverte de *FreeRTOS*, nous allons mettre en place une interface de communication entre la board de développement et l'ordinateur.

Il s'agit d'une communication série asynchrone via UART. Son fonctionnement repose sur une bibliothèque minimaliste, développée par vos enseignants. Il vous est simplement demandé d'en vérifier la bonne compilation et d'en comprendre le fonctionnement.

2.1. Mise en place de l'environnement de travail

L'IDE utilisé pour ces TP est STM32CubeIDE. Pour vous simplifier le travail, votre enseignant a créé un projet intégrant la bibliothèque uart alternative. Suivez les indications ci-dessous :

- Sur l'espace e-campus du cours, récupérez l'archive **.zip du TP1 (UART)** et décompressez-la dans votre workspace.
- A partir de l'IDE STM32CubeIDE, importez ce projet :
Cliquez sur **File→Import...**
 - Sélectionnez **General→Existing project into workspace**. Cliquez *Next*.
 - *Select root directory* : indiquez le répertoire où a été décompressée l'archive (normalement identique au workspace de l'IDE).
 - *Projects* : sélectionnez le projet à importer.
 - *Options* : **ne rien cocher**.
 - Cliquez sur *Finish*. Le projet doit maintenant apparaître dans l'explorateur de projet de l'IDE.

2.2. Compréhension de la bibliothèque uart alternative

- Vérifiez que le corps du `main()` correspond bien au code donné au listing 1.
- Ouvrez un terminal asynchrone de communication côté ordinateur¹ (*TeraTerm*, *HyperTerminal*, *Putty*, *MiniCom*, ...) et configurez-le en accord avec la configuration de l'UART du MCU².
- 🔧 Dans cette configuration, quelle est le **temps approximatif d'émission** d'un caractère ?
- 🔧 Analysez les données reçues et interprétez le fonctionnement du programme³.
- 🔧 A quoi servent les variables `uartRxCircBuff.buffer[]`, `uartRxCircBuff.elNb`, `uartRxCircBuff.indexR`, et `uartRxCircBuff.indexW` ? Comment appelle-t-on ce type de structure ? Explicitez son fonctionnement.

1. Les broches utilisées par le périphérique UART sont PA.2(Tx) et PA.3(Rx), directement connectées au VCP (Virtual COM Port) de la sonde de programmation.

2. Aidez-vous de la fonction `MESN_UART_Init()`.

3. Ne pas hésiter à modifier le fichier `main.c` afin de bien comprendre le fonctionnement de l'API gestion de l'UART.

3. Exo - mode Coopératif

Nous allons maintenant découvrir pas à pas les principaux services proposés par un OS temps réel tel que FreeRTOS.

3.1. Mise en place de l'environnement de travail

➔ Sur l'espace moodle du cours, récupérez l'archive **.zip Coop** et décompressez-la dans votre répertoire de travail.

➔ Répétez la procédure de la section 2.1 afin d'importer ce projet dans l'IDE STM32CubeIDE.

Celui-ci doit normalement contenir le fichier source de la librairie UART alternative **mesn_uart.c/.h** dans le répertoire Drivers/MeSN⁴ ainsi que les fichiers applicatifs suivants, situés dans le répertoire **src** :

main.c fichier source destiné à contenir le programme principal de l'application, **main()**. Ce programme est exécuté après chaque reset, il est chargé d'initialiser les ressources matérielles ainsi que l'OS puis il démarre l'OS qui sera alors exécuté indéfiniment.

freertos.c fichier source destiné à contenir le code des tâches de l'application ainsi que le code des fonctions crochets⁵ de FreeRTOS.

Ce projet contient également un répertoire intitulé Middlewares/Third_Party/FreeRTOS contenant tous les fichiers sources de FreeRTOS. Comme détaillé dans les annexes, ceux-ci se résument à :

6 fichiers sources constituant le cœur de l'OS et indépendants de l'architecture matérielle (*tasks.c*, *queue.c*, *croutine.c*, *timers.c*, *event_groups.c* et *list.c*).

1 fichier source dédié au portage de l'OS sur architecture Cortex-M3 (*port.c*) situé dans /portable/GCC/ARM_CM3.

1 fichier source intégrant les mécanismes de gestion mémoire souhaités, spécifique à l'architecture Cortex-M3 (*heap_1.c*) situé dans /portable/MemMang.

1 fichier source facultatif permettant d'utiliser l'API standard de programmation des OS CMSIS-RTOS à la place de l'API spécifique à FreeRTOS (*cmsis-os.c*). On parle de wrapper. Ceci permet d'assurer la portabilité d'une application sur plusieurs OS respectant le standard CMSIS⁶.

➔ Après avoir vérifié les éléments ci-dessus, assurez-vous que la compilation du projet ne génère pas d'erreur.

3.2. Premiers pas dans un environnement multitâche

Tout d'abord, nous allons créer 3 tâches, de même priorité, chargées chacune d'envoyer une chaîne de caractères vers l'ordinateur via la liaison série.

➔ En vous aidant du document de préambule et de la documentation en ligne de l'API CMSIS-RTOS proposée sur le site <http://www.keil.com/pack/doc/CMSIS/RTOS/html/index.html> ainsi que de celle de FreeRTOS (<http://www.freertos.org>), complétez les fichiers sources *main.c* et *freertos.c* afin de créer **3 tâches de priorité basse**⁷. Les fonctions implémentées par les tâches se nomment

4. Si ces fichiers sont absents de l'explorateur de projet de l'IDE mais bien présents sur le disque de la machine, c'est que le répertoire Drivers/MeSN de l'IDE est vu comme un répertoire virtuel. Dans ce cas, supprimez simplement ce répertoire depuis l'IDE puis rafraîchissez le projet. Il devrait apparaître de nouveau, cette fois en contenant les fichiers.

5. Nous verrons ce que cela signifie d'ici peu.

6. Sous réserve de n'utiliser que des appels de fonctions de l'API standard.

7. Les modifications à apporter aux fichiers sources sont repérées par le mot-clé **TODO** dans les fichiers sources concernés.

```

1 void task1Fn(void const * argument){
2     uint32_t i;
3     while(1){
4         // envoi d un message sur 1 UART
5         MESN_UART_PutString_Poll((uint8_t*)"r\nTask1#");
6         // attendre quelques milli-secondes
7         for (i=0; i<100000; i++);
8     }
9 }

```

Listing 2 – Code source de la tâche n°1.

task1Fn(), task2Fn() et task3Fn(). Chaque tâche ne fera qu'envoyer une chaîne de caractères vers l'ordinateur puis attendre quelques centaines de millisecondes via une temporisation soft avant de répéter l'opération. A titre d'exemple, le code de la fonction implémentée par la tâche 1 est donné sur le listing 2.

- ✎ Lors de la création des tâches, expliquez la valeur du dernier paramètre que vous avez indiqué lors de l'appel à la fonction `osThreadDef()`, donnez son unité.
- ✎ Comparez avec la taille de la RAM du MCU et avec celle du tas alloué à l'OS.

➡ Exécutez votre programme.

D'après les trames reçues sur l'ordinateur, nous constatons que seule la tâche 3 s'exécute. Trois questions se posent alors :

- ✎ Pourquoi sommes nous bloqué ?
Tout simplement car en mode coopératif, la tâche en cours d'exécution doit appeler elle-même l'ordonnanceur (ou une fonction système y faisant appel) afin de rendre la main de façon explicite. Il faut que les tâches "coopèrent".
- ✎ Pourquoi dans la tâche 3 ?
Dans FreeRTOS, au démarrage, si plusieurs tâches de même priorité sont susceptibles de prendre la main, c'est la dernière tâche créée qui sera la première à démarrer. Cette politique dépend de l'OS considéré.
- ✎ Dans quel état se trouvent les tâches 1 et 2 ?
Elles sont dans l'état prêt (ready). Elles sont toutes les deux prêtes à prendre la main si la tâche 3 la redonne.

A partir de cette analyse, nous pouvons représenter l'emploi du temps du CPU tel qu'illustré sur la Figure 1.

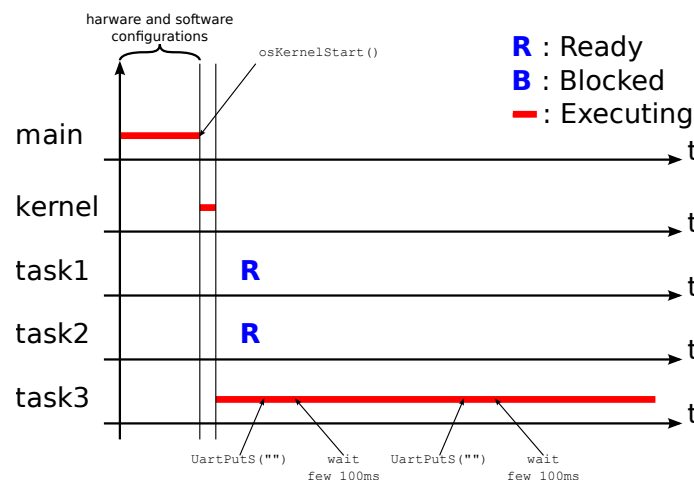


FIGURE 1 – Chronogramme d'exécution du programme, déduit des trames reçues sur l'ordinateur. Les traits pleins représentent le code en cours d'exécution par le CPU.

Nous allons maintenant forcer des commutations de contexte en appelant l'ordonnanceur de façon explicite dans chacune des tâches.

➡ Dans chaque tâche, à la suite de la temporisation logicielle, appelez la fonction `osThreadYield()`.

🔗 En visualisant les données reçues sur l'ordinateur, le fonctionnement vous semble-t-il satisfaisant ?

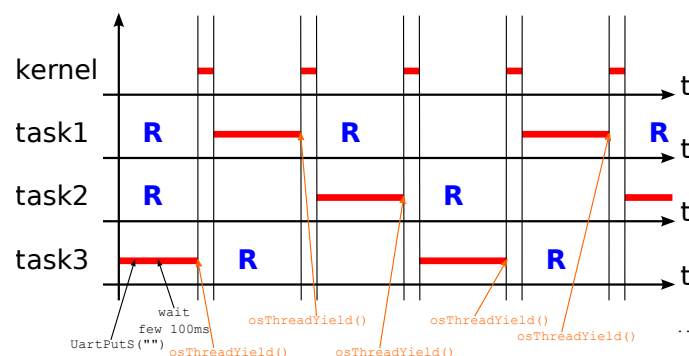


FIGURE 2 – Chronogramme d'exécution du programme, déduit des trames reçues sur l'ordinateur lorsque les tâches coopèrent.

D'après les trames reçues sur l'ordinateur, nous constatons cette fois que l'ordonnanceur donne la main à chacune des tâches à tour de rôle (Figure 2). **Beaucoup d'OS temps réel légers travaillent ainsi pour répartir le temps CPU entre plusieurs tâches de même priorité, il s'agit de la technique dites du round-robin qui suit le principe du tourniquet.** Lorsque plusieurs tâches de même priorité sont toutes à l'état prêt, elle se verront attribuées du temps CPU à tour de rôle.

Nous venons d'illustrer le principe de la coopération entre tâches ainsi que le problème engendré lorsqu'une boucle infinie (bug) intervient dans le code d'une tâche. Dans ce cas, les autres tâches bloquées ou prêtes ne peuvent plus prendre la main, et l'application complète tombe !

3.3. État bloqué

Nous allons maintenant nous intéresser à la fonction `osDelay()`. Comme exposé dans les annexes, FreeRTOS est un OS modulable. C'est à dire que de nombreuses fonctionnalités sont désactivables (non-incluses lors

de la compilation) afin de réduire l'empreinte mémoire lorsque celles-ci ne sont pas nécessaires. La fonction `osDelay()` fait partie de ces fonctions optionnelles.

- ➡ Passez à 1 la macro `INCLUDE_vTaskDelay` dans le fichier d'en-tête *FreeRTOSConfig.h* afin de pouvoir utiliser la fonction `osDelay()`.
- ➡ Dans la tâche 1 supprimez la temporisation `soft` et l'appel à `osThreadYield()` puis remplacez-les par un appel à la fonction `osDelay()` tel qu'illustrer sur le listing 3.

```

1 void task1Fn(void const * argument){
2     while(1){
3         // envoi d un message UART
4         MESN_Uart_PutString_Poll((uint8_t*)"r\nTask1#");
5         //tache retardee pendant 300
6         osDelay( 300 );
7     }
8 }

```

Listing 3 – Utilisation de la fonction `osDelay()`.

- 🔗 Quel est le rôle de la fonction `osDelay()` ? Indiquez à quoi correspond le paramètre 300 et sur quelle base de temps est-il indexé afin d'obtenir une durée équivalente en secondes ?
- 🔗 En analysant les trames reçues sur l'ordinateur, complétez le chronogramme de la figure 3. Précisez à chaque fois les appels aux fonctions `osThreadYield()` et `osDelay()` ainsi que l'état pris par chaque tâche (*R=Ready* et *B=Blocked*).

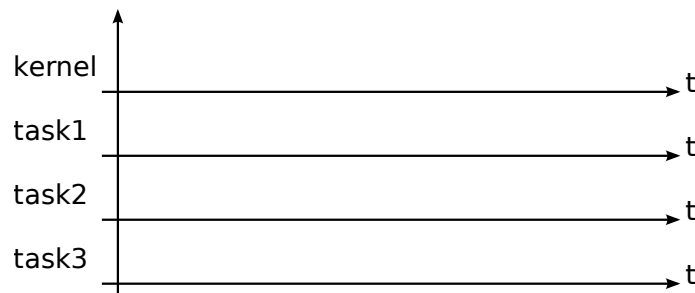


FIGURE 3 – Chronogramme à compléter.

- ➡ Remplacez maintenant les temporisations logicielles et `osThreadYield()` par `osDelay()` dans toutes les tâches.
- 🔗 Complétez le chronogramme de la figure 4 et précisez l'état pris par chaque tâche.

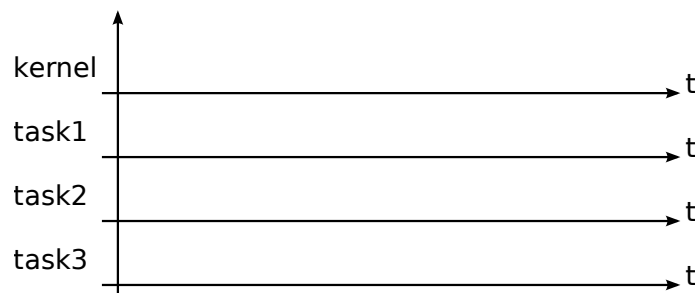


FIGURE 4 – Chronogramme à compléter.

- 🔗 Quel est le code qui s'exécute lorsque toutes les tâches sont à l'état bloqué ?

3.4. Tâche Idle

Nous allons maintenant nous attarder un peu sur la tâche **Idle**.

✎ Indiquer ce que fait, par défaut, la fonction implémentée par la tâche *Idle* lorsque nous sommes en mode coopératif (explorer les sources de FreeRTOS pour répondre) :

Nous allons maintenant détourner cette tâche *Idle* afin d'y insérer du code utilisateur.

➡ Passez à 1 la macro **configUSE_IDLE_HOOK** dans le fichier d'en-tête *FreeRTOSConfig.h*⁸.

✎ En vous aidant de la doc en ligne de FreeRTOS, ou du code source de la tâche idle, expliquez le rôle de cette macro ?

➡ Éditez le fichier *freertos.c* afin d'y créer une fonction (et non une tâche !) nommée **vApplicationIdleHook()**⁹ telle que présenté sur le listing 4. Cette fonction ne fait qu'envoyer le caractère 'I' via l'UART. Compilez puis exécutez votre programme.

```

1  /**
2  * @fn void vApplicationIdleHook(void)
3  * @brief function called by Idle task
4  */
5  void vApplicationIdleHook(void){
6      MESN_Uart_PutString_Poll((uint8_t*)"I");
7  }
```

Listing 4 – Code de la fonction vApplicationIdleHook().

✎ Compléter le chronogramme de la figure 5 traçant l'exécution de votre programme.

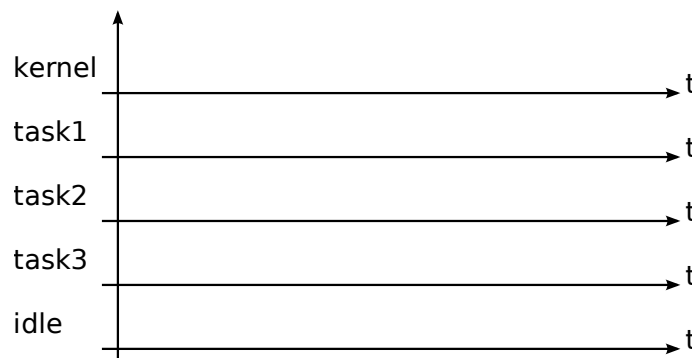


FIGURE 5 – Chronogramme à compléter.

✎ Proposer des cas d'applications et exemples d'utilisations de la tâche Idle. Ne pas hésiter à s'aider du web et d'exemples de détournements de la tâche Idle sur d'autres noyaux temps réel.

8. Pour information "hook" signifie "crochet" ou "détour", ce qui correspond au "détournement" de la tâche Idle.

9. Attention, le prototype de cette fonction est imposé par le noyau

TP 2.

Mode préemptif et gestion mémoire

Dorénavant, et ce 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 la tâche en cours d'exécution, puis force un réordonnancement (Figure 6).** Cette périodicité se nomme **tick** (*timer clock*) et est configurable sous FreeRTOS à travers la macro **configTICK_RATE_HZ** du fichier *FreeRTOSConfig.h*.

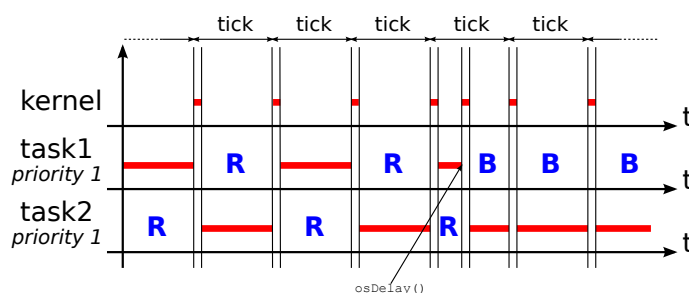


FIGURE 6 – Exemple de chronogramme pour un fonctionnement en mode préemptif.

Ce TP sera également l'occasion de découvrir les mécanismes de gestion mémoire utilisés sous FreeRTOS. Ce point est extrêmement important car source de nombreux bugs et mauvais développements en milieu industriel particulièrement délicats à diagnostiquer.

Table des matières

1. Travail préparatoire (à rendre via moodle)	12
2. Mise en place de l'environnement de travail	14
2.1. Nouveau projet	14
2.2. Intégration des outils de Trace	14
3. Mode préemptif	14
4. Gestion mémoire	16

1. Travail préparatoire (à rendre via moodle)

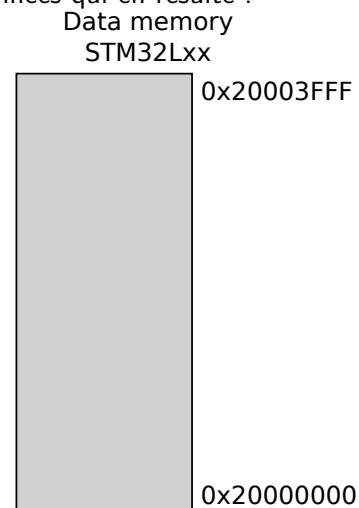
Questions autour du mode préemptif et des mécanismes de gestion de la mémoire par FreeRTOS.

- 🔗 Quels sont les inconvénients et avantages d'un OS **coopératif** vis-à-vis d'un OS **préemptif** ?
- 🔗 Qu'est-ce que le tas (*heap*) d'un OS et que trouve-t-on généralement dedans ?
- 🔗 Les stratégies de gestion du tas par FreeRTOS sont implémentées dans les fichiers **heap_1.c**, **heap_2.c**, **heap_3.c** et **heap_4.c** présents dans le répertoire [...]/*FreeRTOS/Source/portable/MemMang*. En vous aidant de la documentation en ligne, répondez aux questions suivantes :
 - Quelles sont les différences entre les stratégies utilisant heap_1.c ou heap_2.c ?
 - Quelles sont les différences entre les stratégies utilisant heap_2.c ou heap_3.c ?
- 🔗 Qu'est-ce qu'une pile (*stack*) et que trouve-t-on dedans ?

La gestion mémoire est un point important de la programmation des petits exécutifs temps-réel car sujette à de nombreux bugs particulièrement difficile à détecter et diagnostiquer. En effet, sur ce type de noyau, l'utilisation des ressources mémoires est sous la responsabilité du développeur et requiert donc une bonne gestion et maîtrise des allocations mémoire réalisées par la chaîne de compilation et le noyau. Prenons un petit programme d'exemple et observons le mapping mémoire des données qui en résulte :

```

1  uint32_t gbl;
2
3  /**
4  * @fn main(void)
5  */
6  void main(void){
7      uint32_t lclMain;
8      osThreadDef(taskName, taskFn, \
9                  osPriorityNormal, 0, 100);
10     osThreadCreate (osThread(Task1), NULL);
11     osKernelStart(NULL, NULL);
12 }
13
14 /**
15 * @fn void taskFn(void const * argument)
16 */
17 void taskFn(void const * argument){
18     uint32_t lclTask;
19     while(1){
20         /* something to do */
21         // ....
22     }
23 }
```




- 🔗 Représenter sur le schéma ci-dessus le découpage du mapping mémoire en faisant apparaître les zones suivantes : pile du main, tas de l'OS, pile de la tâche.
- 🔗 Que trouve-t-on dans le reste de la mémoire (en dehors des zones spécifiées à la question précédentes) ?
- 🔗 Où se situe la chaîne de caractères **"taskName"** ? Mettre à jour le schéma.
- 🔗 Où se situe la variable globale **gbl** ? Mettre à jour le schéma.
- 🔗 Où se situe la variable locale **lclMain** ? Mettre à jour le schéma.
- 🔗 Où se situe la variable locale **lclTask** ? Mettre à jour le schéma.

Supposons que le programme principal soit modifié comme donné ci-dessous :

```

1  void main(void){
2      uint32_t lclMain;
3      osThreadDef(taskName, taskFn, \
4                  osPriorityNormal, 0, 100);
5      osThreadCreate (osThread(Task1), NULL);
6      osThreadDef(anotherTask, taskFn, \
7                  osPriorityNormal, 0, 100);
8      osThreadCreate (osThread(Task2), NULL);
9      osKernelStart(NULL, NULL);
10 }
```

 Cela vous semble-t-il correct ? Mettez à jour le schéma du mapping mémoire.

2. Mise en place de l'environnement de travail

2.1. Nouveau projet

- Reprenez le projet du TP précédent, concernant le mode coopératif. Si vous le souhaitez, vous pouvez en effectuer une copie depuis l'explorateur de projet de STM32CubeIDE afin de garder une trace du travail précédent.
 - Désactivez l'utilisation du crochet de la tâche *Idle* (macro **configUSE_IDLE_HOOK** à 0 dans le fichier d'en-tête *FreeRTOSConfig.h*).
 - Effacez le contenu des 3 tâches précédemment utilisées.
 - Configurez l'OS en mode **préemptif** (dans le fichier d'en-tête *FreeRTOSConfig.h*).
- ✎ Indiquez la modification effectuée pour réaliser cette étape.

2.2. Intégration des outils de Trace

Jusqu'ici, nous avons étudié le fonctionnement de nos programmes à l'aide de chronogrammes fait à la main. Néanmoins, il existe des outils dédiés permettant d'effectuer ce type d'analyses.

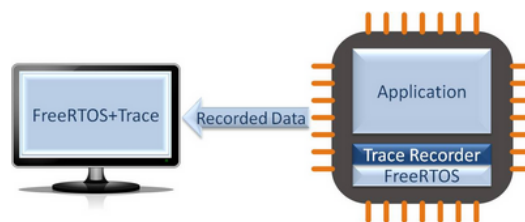


FIGURE 7 – FreeRTOS+Trace.

Afin de faciliter la compréhension du fonctionnement des mécanismes d'ordonnancement, nous allons intégrer ces outils à notre projet.

- Suivez les indications du document d'annexe afin de configurer les outils de traces de votre projet puis vérifiez que la compilation se déroule correctement.

3. Mode préemptif

- Créez 3 tâches dont les fonctions implémentées par les tâches se nommeront respectivement `task1Fn()`, `task2Fn()` et `task3Fn()`, en respectant les niveaux de priorités des tâches ci-dessous :
 - tâche 1 de priorité normale.
 - tâche 2 et 3 de priorité basse.
 - Recopiez le code de chacune des tâches tel que donné au listing de code 5 puis exécutez votre programme.
- ✎ Interprétez le fonctionnement du programme en visualisant les données reçues sur l'ordinateur¹⁰. En vous aidant des outils de trace, complétez le chronogramme de la figure 8.

¹⁰. Le comportement du programme peut sembler étrange dans un premier temps, mais cela s'explique bien entendu très bien !

```
1 void task1Fn(void const * argument){
2     uint32_t cycle = 0;
3
4     while(1){
5         /* send a string to UART, regarding cycle number */
6         if (cycle == 0){
7             MESN_UART_PutString_Poll((uint8_t*)"r\nTask1 ##");
8         }else if(cycle == 1){
9             MESN_UART_PutString_Poll((uint8_t*)"r\nTask1 #####");
10        }else if(cycle == 2){
11            MESN_UART_PutString_Poll((uint8_t*)"r\nTask1 #####");
12        }
13        /* update cycle number */
14        cycle++;
15        if (cycle == 3){
16            cycle = 0;
17        }
18
19        /* task blocked during 300 ms */
20        osDelay(300);
21    }
22 }
23
24 void task2Fn(void const * argument){
25     while(1){
26         /* send a string to UART */
27         MESN_UART_PutString_Poll((uint8_t*)"r\nTask2 ##");
28         /* task blocked during 100 ms */
29         osDelay(100);
30     }
31 }
32
33 void task3Fn(void const * argument){
34     while(1){
35         /* send a string to UART */
36         MESN_UART_PutString_Poll((uint8_t*)"r\nTask3 ###");
37         /* task blocked during 100 ms */
38         osDelay(100);
39     }
40 }
```

Listing 5 – Code des fonctions implémentées par les tâches 1, 2 et 3 respectivement.

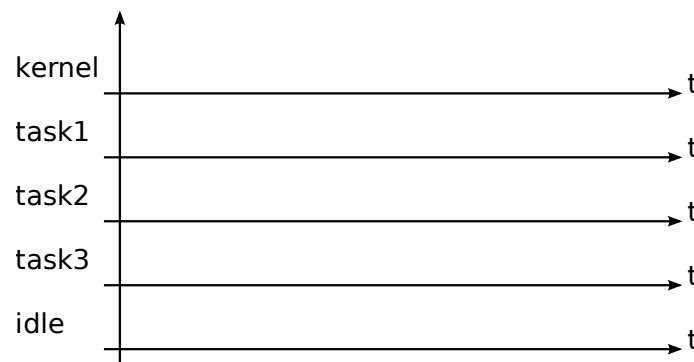


FIGURE 8 – Chronogramme à compléter.

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

L'API CMSIS-RTOS propose la fonction `osKernelSysTick()` permettant de récupérer la valeur courante du tick.

- ➡ En vous inspirant du listing de code ci-dessous, modifiez le code exécuté par la tâche 1 afin de récupérer la valeur courante du tick à chaque réveil de la tâche 1 puis de l'envoyer à l'ordinateur via l'UART ¹¹. Testez votre programme.

```

1 void task1Fn(void const * argument){
2     while(1){
3         /* get current tick value */
4         // ...TODO...
5         /* send it to UART */
6         MESN_UART_PutString_Poll((uint8_t*) "\r\nTask1 - tickVal = ");
7         // ...TODO...
8         /* task blocked during 300ms */
9         osDelay(300);
10    }
11 }
```

- ✎ Qu'en déduisez-vous vis-à-vis de la périodicité de la tâche 1 ?
- ➡ En vous aidant de la documentation en ligne, remplacer la fonction bloquante `osDelay()` par `osDelayUntil()` ¹².
- ✎ En comparant les résultats obtenus avec les deux fonctions, expliquez la différence de fonctionnement. Dans ce cas, la tâche 1 est-elle périodique ?

4. Gestion mémoire

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

Nous allons maintenant mettre en place les mécanisme de détection de débordement de pile.

11. Aidez-vous de la fonction standard `sprintf()`.

12. Attention, il est nécessaire d'activer cette fonctionnalité optionnelle dans la configuration de FreeRTOS.

- ➡ Passez la macro `configCHECK_FOR_STACK_OVERFLOW` à 2 (ou 1) dans le fichier *FreeRTOSConfig.h*. En fonction de la valeur choisie, différentes stratégies de détection seront appliquées par le kernel : <https://www.freertos.org/Stacks-and-stack-overflow-checking.html>.
- ➡ Éditez le fichier **freertos.c** afin d'y créer la fonction crochet `vApplicationStackOverflowHook()` contenant le code suivant :

```

1  /**
2   * @fn void vApplicationStackOverflowHook
3   * @brief This hook function is called if a stack overflow is detected.
4   */
5  void vApplicationStackOverflowHook(xTaskHandle xTask, signed char *pcTaskName){
6      /* Run time stack overflow checking is performed if
7       configCHECK_FOR_STACK_OVERFLOW is defined to 1 or 2.*/
8
9      MESN_UART_PutString_Poll((uint8_t*) "\r\nERROR : stack overflow by ");
10     MESN_UART_PutString_Poll(pcTaskName);
11
12     while(1); // Should replace with a software mcu reset
13 }

```

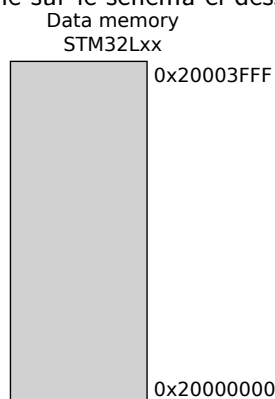
- ➡ Ajoutez le code de la fonction suivante et appelez-la depuis la tâche 1.

```

1  /**
2   * @fn uint32_t growStack(void)
3   * @brief stack allocation until overflow
4   */
5  uint32_t growStack(void){
6      volatile uint32_t lclVar;
7      osDelay(1);
8      lclVar = growStack();
9      return lclVar;
10 }

```

- ➡ Placez un point d'arrêt sur le `while(1);` de la fonction `vApplicationStackOverflowHook()` puis exécutez votre programme en mode debug.
- 🔗 A l'aide des outils de trace et des données reçues sur le terminal UART, expliquez le comportement du programme sur le schéma ci-dessous.



Les mécanismes de détection proposés par FreeRTOS ne sont pas parfaits, et il arrive qu'ils ne soient pas capable de détecter les débordements avant le crash de l'application.

- ➡ Pour illustrer cela, effectuez le même exercice en supprimant l'appel à la fonction `osDelay()` présent dans le fonction `growStack()`.
- 🔗 Que constatez-vous ? Expliquez le comportement du programme sur le schéma précédent.

TP 3.

Outils de Communication/Synchronisation

Nous allons maintenant nous intéresser aux outils de communication, synchronisation et protection proposés par FreeRTOS. Nous verrons notamment les files d'attente (ou *queue de messages*, ou *boîte aux lettres*, ou *mailboxes*) qui permettent d'effectuer des échanges sécurisés entre différentes tâches (ou ISR) de l'application ainsi que les sémaphores.

Table des matières

1. Mise en place de l'environnement de travail	19
2. Synchronisation par sémaphore	19
3. Queue de messages	20
4. Timeout	22
5. Protection de ressource partagée et sections critiques	22
6. Exo - Bibliothèque UART avec appels système	24

1. Mise en place de l'environnement de travail

- Reprenez le projet du TP précédent. Si vous le souhaitez, vous pouvez en effectuer une copie depuis l'explorateur de projet de STM32CubeIDE afin de garder une trace du travail précédent.
- Effacez le contenu des 3 tâches précédemment utilisées.
- Assurez-vous que l'OS est bien configuré en mode **préemptif**.

2. Synchronisation par sémaphore

Dans cet exercice, nous allons tenter de synchroniser la tâche 2 avec la tâche 1 à l'aide de l'outil sémaphore. La synchronisation permet notamment de synchroniser le début de l'exécution d'un traitement par une tâche (par exemple un traitement long) suite à la fin de l'exécution d'un autre traitement par une autre tâche ou d'une ISR (traitement toujours court).

- Créez 3 tâches dont les fonctions implémentées par les tâches se nommeront respectivement `task1Fn()`, `task2Fn()` et `task3Fn()`, en respectant les niveaux de priorités des tâches ci-dessous :
 - tâche 1 de priorité normale.
 - tâche 2 et 3 de priorité basse.
- Éditez le code de ces fonctions en respectant les indications ci-dessous :
 - La tâche 1, devra être périodique avec une périodicité de 500 ms. A chacun de ses réveils, elle libérera un sémaphore afin de débloquer la tâche 2.

```

1 void task1Fn(void const * argument){
2     //TODO...
3
4     while(1){
5         /* 500 msec periodicity */
6         //TODO...
7
8         /* send a string to UART */
9         MESN_Uart_PutString_Poll((uint8_t*) "\r\nTask1 - semaphore given");
10
11        /* Give semaphore */
12        // TODO...
13    }
14 }
```

- La tâche 2 devra être synchronisée avec la tâche 1 et enverra une chaîne de caractères via liaison série. Le reste du temps, cette fonction devra rester bloquée.

```

1 void task2Fn(void const * argument){
2     while(1){
3         /* wait for semaphore */
4         //TODO...
5
6         /* send a string to UART */
7         MESN_Uart_PutString_Poll((uint8_t*) " -> task2 - synchro");
8     }
9 }
```

- La tâche 3 se bloquera pendant 100 ms après avoir envoyé sa chaîne de caractères :

```

1 void task3Fn(void const * argument){
2     while(1){
3         /* send a string to UART */
4         MESN_Uart_PutString_Poll((uint8_t*) "\r\ntask3 ###");
5         /* task blocked during 100 ms */
6         osDelay(100);
7     }
8 }

```

Attention ! Avant qu'une tâche puisse utiliser (*prendre* ou *vendre*) un sémaphore, il faut que celui-ci soit préalablement créé.

➡ Modifier le code de la fonction `MX_FREERTOS_Init()` afin de créer le sémaphore qui sera utilisé pour la synchronisation.

🔗 Interprétez le fonctionnement du programme en visualisant les données reçues sur l'ordinateur puis, en vous aidant des outils de trace, complétez le chronogramme suivant (Figure 9).

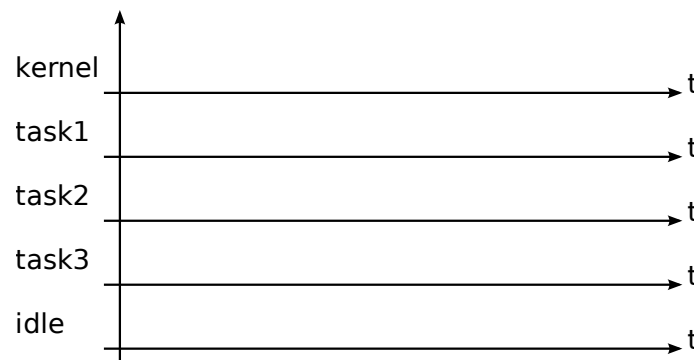


FIGURE 9 – Chronogramme à compléter.

🔗 Quelle est la période d'exécution de la tâche 2 ?

3. Queue de messages

Maintenant que nous venons d'illustrer la synchronisation entre 2 tâches par sémaphore. Nous allons voir comment nous pouvons effectuer des échanges d'informations entre différentes tâches de l'application. Pour cela, nous allons légèrement modifier le code des fonctions implémentées par les tâches précédentes.

➡ Retirer le code faisant référence au sémaphore de l'exercice précédent.

➡ Modifier les fonctions des 3 tâches de la façon suivante :

- La tâche 3 est inchangée.
- La tâche 1, périodique de 500 ms, doit récupérer la valeur courante du tick d'OS (**sans** l'envoyer à l'ordinateur) puis la poster dans une queue de message à destination de la tâche 2. La fonction d'écriture dans la file d'attente ne devra pas être bloquante.

```

1 void task1(void const * argument){
2     //TODO...
3
4     while(1){
5         /* 500 msec periodicity */
6         //TODO...
7
8         /* send a string to UART */
9         MESN_Uart_PutString_Poll((uint8_t*) "\r\ntask1 - MsgSent");
10
11        /* read and post current tick value in mailbox */
12        // TODO...
13    }
14 }

```

- La tâche 2 doit récupérer la valeur du tick qui a été placée dans le queue par la tâche 1 puis la renvoyer à l'ordinateur via la liaison UART. Le reste du temps, cette fonction devra rester bloquée.

```

1 void task2(void const * argument){
2     while(1){
3         /* wait for available message */
4         //TODO...
5         /* send a string to UART */
6         MESN_Uart_PutString_Poll((uint8_t*) "\r\ntask2 - MsgRcvd : ");
7         /* print current tick value on UART */
8         //TODO...
9     }
10 }

```

Attention ! Avant qu'une tâche puisse utiliser (*écrire* ou *lire*) une queue de message, il faut que celle-ci soit préalablement créée.

- ➡ Modifier le code de la fonction `MX_FREERTOS_Init()` afin de créer une queue de message pour échanger de l'information entre les tâches 1 et 2. A vous de fixer la **taille** de la file d'attente ainsi que la **taille** de chaque élément.
- 🔗 Interprétez le fonctionnement du programme en visualisant les données reçues sur l'ordinateur puis, en vous aidant des outils de trace, complétez le chronogramme suivant (Figure 10).



FIGURE 10 – Chronogramme à compléter.

- 🔗 Quelle est la période d'exécution de la tâche 2 ?

4. Timeout

Comme nous venons de le voir, certaines fonctions pour la gestion de queue de messages ou sémaphores sont des fonctions bloquantes¹³. Ces fonctions utilisent alors un **Timeout** : lorsqu'une tâche est bloquée suite à l'appel de l'une de ces fonctions, la tâche se réveillera (passage à l'état prêt) automatiquement après un laps de temps appelé Timeout, même si l'événement attendu n'est pas arrivé.

➔ Reprenez le code de la fonction implémentée par la tâche 2 afin de forcer le réveil de cette tâche toutes les 300 ms en utilisant le timeout associé à la fonction `osMessageGet()`. Après avoir testé la nature du réveil de la tâche (succès ou échec de la lecture du message), envoyer l'une des deux chaînes de caractères suivantes :

- Si réveil par lecture du message présent dans la queue :

```
1 | MESN_Uart_PutString_Poll("\r\ntask2 - MsgRcvd : ");
2 | /* print current tick value on UART */
3 | //...
```

- Si réveil par timeout :

```
1 | MESN_Uart_PutString_Poll("\r\ntask2 - TimeOut");
```

- 🔗 Que se passe-t-il si nous forçons le timeout d'une fonction bloquante à 0 ?
- 🔗 À quelle valeur de timeout (en nombre de ticks d'OS) correspond l'argument **osWaitForever** ? Dans notre cas, quelle est la valeur (en temps) équivalente ? Que vaut cette dernière si la macro **INCLUDE_vTaskSuspend** du fichier *FreeRTOSConfig.h* est à 1 ?
- 🔗 Peut-on trouver un timeout sur une fonction non-bloquante ?

5. Protection de ressource partagée et sections critiques

Une section critique est une portion de code pour laquelle nous devons garantir la bonne exécution et l'intégrité des données à la sortie. Il s'agira le plus souvent de protéger une ressource partagée entre plusieurs tâches et/ou ISR (variable globale, accès à un périphérique, ...). La figure 11 illustre ce phénomène dans le cas où deux tâches sont chargées d'incrémenter une même variable globale `i`¹⁴.

En fonction de la protection requise, une section critique peut-être protégée par différents outils systèmes :

1. Ressource partagée entre plusieurs tâches et ISRs :

Dans ce cas, il est nécessaire de masquer les interruptions lors de l'entrée dans la section critique afin qu'une ISR ne puisse pas préempter la région de code utilisant la ressource partagée. Sous FreeRTOS, ce type de section critique est protégée par les macros **taskENTER_CRITICAL()** et **taskEXIT_CRITICAL()**. Il s'agit de la méthode la plus brutale, à utiliser avec précaution, uniquement pour des section de code très courtes. En effet, les ticks d'OS (générés par timer matériel) ne sont alors plus vu par l'ordonnanceur et le masquage des interruptions peut faire perdre le respect des contraintes temps réel du système.

2. Ressource partagée entre plusieurs tâches uniquement (1/2) :

Dans ce cas, nous pouvons alors garder les interruptions actives et donc la prise en compte des ticks

13. Par exemple : prendre un sémaphore qui n'est pas disponible, écrire dans une queue de message qui est pleine, ...

14. Pour rappel, la ligne de code `i=i+1` signifie généralement plusieurs instructions machine telles que :
lire la valeur de `i` afin de la copier dans un registre interne;
incrémenter ce registre;
écrire la valeur du registre dans `i`;

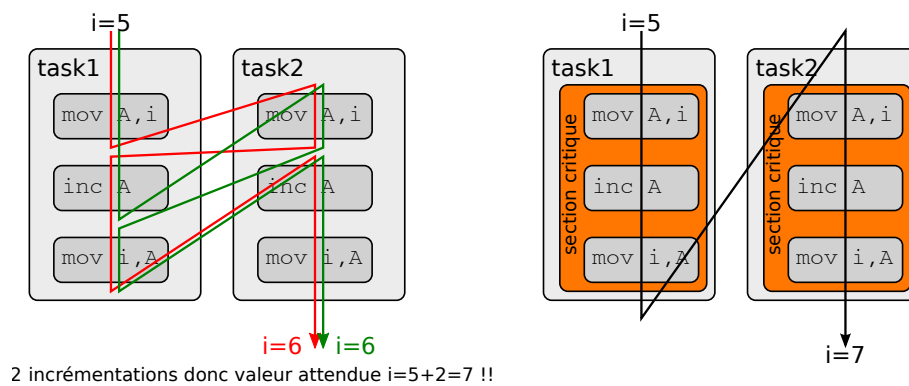


FIGURE 11 – Utilité des sections critiques.

d'OS. Il faut simplement empêcher toutes commutations de contexte afin que la tâche qui entre en section critique ne soit pas préemptée par une autre tâche. Sous FreeRTOS, ce type de section critique est protégée par les fonctions **vTaskSuspendAll()** et **vTaskResumeAll()**. Il apparaît néanmoins que lorsqu'un tâche entre dans ce type de section critique, aucune autre tâche ne pourra prendre la main tant que la section critique ne sera pas terminée, et ce même si il s'agit d'une tâche de priorité supérieure qui ne partage pas la même ressource (et pourrait donc s'exécuter sans risque).

3. Ressource partagée entre plusieurs tâches uniquement (2/2) :

Le principe est le même que précédemment, mais vise à assurer l'intégrité de la section critique uniquement envers les seules tâches susceptibles d'utiliser la ressource partagée. Sous FreeRTOS, ce type de section critique est protégée par **Sémaphore Binaire** ou **Mutex**. Dans ce cas, chaque tâche voulant accéder à la ressource partagée doit préalablement prendre le sémaphore protégeant cette dernière afin de s'assurer que le ressource n'est pas en cours d'utilisation pas une autre tâche. Le cas échéant, elle se bloque en attendant que le sémaphore soit rendu.

Lors des exercices précédents, 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 avec pour conséquence la corruption des chaînes de caractères transmises. 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 cela, vous allez réécrire une partie de la bibliothèque C de gestion de l'UART et donc modifier le fichier source *mesn_uart.c* ainsi que le fichier d'en-tête *mesn_uart.h*.

- Éditez le fichier **mesn_uart.c** de telle sorte qu'un mutex soit créé lors de l'appel à la fonction d'initialisation de l'UART. Choisissez un nom de mutex adapté à sa fonction.
- Modifier le code source de la fonction `MESN_Uart_PutString_Poll()` afin de s'assurer qu'une seule tâche à la fois puisse accéder au périphérique UART en transmission.

Interprétez le fonctionnement du programme et complétez le chronogramme suivant (Figure 12).

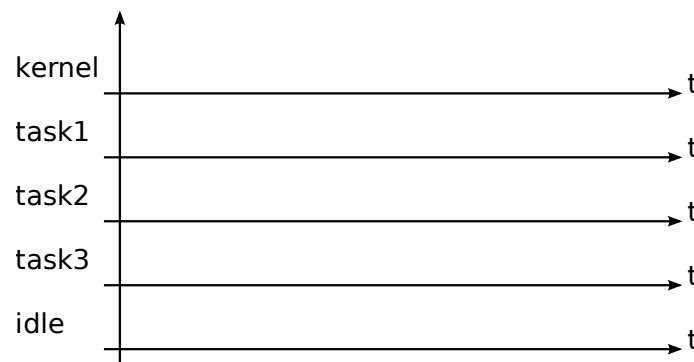


FIGURE 12 – Chronogramme à compléter.

- 🔗 Sous FreeRTOS, qu'elle différence existe-t-il entre un sémaphore binaire et un mutex¹⁵ ?
- 🔗 Dans notre cas, est-il plus intéressant d'utiliser un sémaphore binaire ou un mutex ? Justifiez votre réponse.

6. Exo - Bibliothèque UART avec appels système

ATTENTION : exercice pour les plus avancés uniquement : cet exercice doit être terminé avant la fin de la séance 4. Sinon, passer directement au TP suivant (vous pouvez toujours le faire en dehors des séances de TP) !

Lors du premier TP (section 2), nous avons vu que la bibliothèque d'utilisation de l'UART utilise un buffer circulaire pour l'échange d'information entre la fonction d'interruption (ISR) de réception de l'UART et la fonction `MESN_UART_GetString()`. Lorsque cette dernière est appelée, elle interroge de façon continue le buffer circulaire (*polling*) en attendant la réception d'un caractère. Ce mode de fonctionnement n'est clairement pas optimisé puisqu'il requiert du temps CPU lors du polling, alors qu'il n'y a aucun caractère à traiter. Nous allons utiliser les outils proposés par l'OS pour améliorer cela.

- Dans le fichier **mesn_uart.c**, supprimez toutes les références à l'utilisation du buffer circulaire.
- Modifier l'**ISR** ainsi que la fonction `MESN_UART_GetString()` de façon à synchroniser par queue de messages les réveils de la fonction d'interruption avec l'appel de la fonction `MESN_UART_GetString()`. Pour cela, chaque caractère reçu sera posté dans une queue de messages par l'ISR tandis que la fonction de réception de caractères `MESN_UART_GetString()` implémentera un appel système bloquant interrogeant cette queue de messages.

Une fois ce travail réalisé, testez le fonctionnement en modifiant le code de la fonction de la tâche 3 de façon à réceptionner puis renvoyer les chaînes de caractères envoyées depuis l'ordinateur.

```

1 void task3Fn(void *pvParameters){
2     uint8_t strTmp[50];
3     while(1){
4         /* receive string from UART */
5         MESN_UART_GetString(strTmp);
6         /* echo the received string */
7         MESN_UART_PutString_Poll(&huart3, strTmp);
8         MESN_UART_PutString_Poll(&huart3, "\r\n");
9     }
10 }
```

15. Aidez-vous de la documentation en ligne

TP 4.

Projet

Maintenant que vous connaissez les principaux outils proposés par un OS temps-reel, ainsi que leur utilisation, il convient de mettre en pratique ces connaissances dans l'objectif de réaliser une application multi-tâche possédant des contraintes temps-réel dures. Pour cela, le développement d'un mini-projet vous est proposé, à réaliser durant l'ensemble des séances de TP restantes.

Table des matières

1. Présentation	26
1.1. Périphériques externes au MCU	26
1.2. Spécifications	26
1.3. Ressources mises à disposition	29
2. Travail à effectuer	29
2.1. Construction de l'architecture logiciel	29
2.2. Développement	30
3. Pour les plus avancés	30

1. Présentation

Le but du projet est de développer une application embarquée pilotant un robot auto-balancé (équilibriste), de type *segway*. L'asservissement en position d'équilibre est assuré par un algorithme de commande actionnant la rotation des roues du robot à partir de la connaissance de l'angle que forme l'axe du robot avec l'axe vertical. Cet angle est estimé par le biais d'un observateur à partir des mesures issues d'un accéléromètre (mesure de l'accélération) et d'un gyroscope (mesure de la vitesse angulaire). Ces deux capteurs forment une cellule "inertielle" (IMU : inertial measurement unit).

Le robot dispose également d'une possibilité de communication série permettant l'échange d'informations avec tout ordinateur muni d'un terminal asynchrone de communication.

1.1. Périphériques externes au MCU

1.1.1. Cellule inertielle

La cellule inertielle est composée d'un accéléromètre/gyroscope LSM6DS3 de chez ST, dialoguant via le protocole I2C.

Un pilote de ce périphérique vous est fourni. Il s'appuie sur une couche de communication I2C, intégrant les outils de l'OS, qui est également fournie.

1.1.2. Moteur à courant continu

La rotation des roues est actionnée par un moteur à courant continu. Celui-ci est relié au MCU via un étage de puissance de type pont en H.

Le pilote de ce périphérique vous est fourni, il s'agit du pilote que vous avez développé lors des TP sur MCU 32bits.

1.1.3. Light Emitting Diode (LED)

L'application propose une IHM locale rudimentaire à base de LED. Votre développement utilisera la LED présente sur la maquette de développement.

1.2. Spécifications

1.2.1. Fonctionnalités

L'application doit être multitâche et réaliser les traitements suivants :

1. Une LED sert d'indicateur pour l'utilisateur. Si la valeur courante de l'angle est supérieure à $|25|$ deg, alors la LED doit rester allumée. Si la valeur est inférieure à $|25|$ deg, alors la LED doit clignoter par flash de 100 ms avec une périodicité 0.5 s. Un angle de $|25|$ deg correspond à la limite de stabilité de l'asservissement d'équilibre.
2. Pour fonctionner correctement la loi de commande pilotant l'asservissement d'équilibre doit être exécutée toutes les 10 ms, avec une période d'échantillonnage de mesure identique.
3. L'application doit toujours garder en mémoire les 100 dernières valeurs de l'angle estimé.
4. L'application doit proposer un shell de communication avec tout ordinateur muni d'un terminal asynchrone (*TeraTerm*, *RealTerm*, ...). Ce shell est relativement simple et ne propose à l'intervenant qu'un jeu de 4 commandes décrites au tableau 1. Au démarrage de l'application, le programme doit présenter une invite de commande conformément à l'interface illustrée sur la figure 13. Chaque

commande est validée par l'appuie sur la touche <entrée> côté ordinateur, l'invite de commande est réaffichée à chaque fois qu'une commande est envoyée. La figure 14 montre un aperçu des échanges avec l'application.

5. L'application devra être facilement évolutive notamment afin de permettre une commande du robot via communication Ethernet puis, à terme, en sans-fil via Bluetooth.

Commande	Description
read	Retourne la dernière valeur mesurée de l'angle d'inclinaison du robot (en milli-Degrés).
dump	Retourne les 100 dernières valeurs mesurées (en milli-Degrés).
stream	Retourne en continu, toutes les 10 ms, la dernière valeur mesurée (en milli-Degrés) en effaçant côté terminal la précédente valeur affichée. La sortie du mode stream s'affectue en appuyant sur la touche <enter> côté terminal.
help	Retourne la liste des commandes supportées par l'application et leur description.

TABLE 1 – Liste des commandes supportées par le shell de communication.



FIGURE 13 – Invite de commande proposée par l'application.

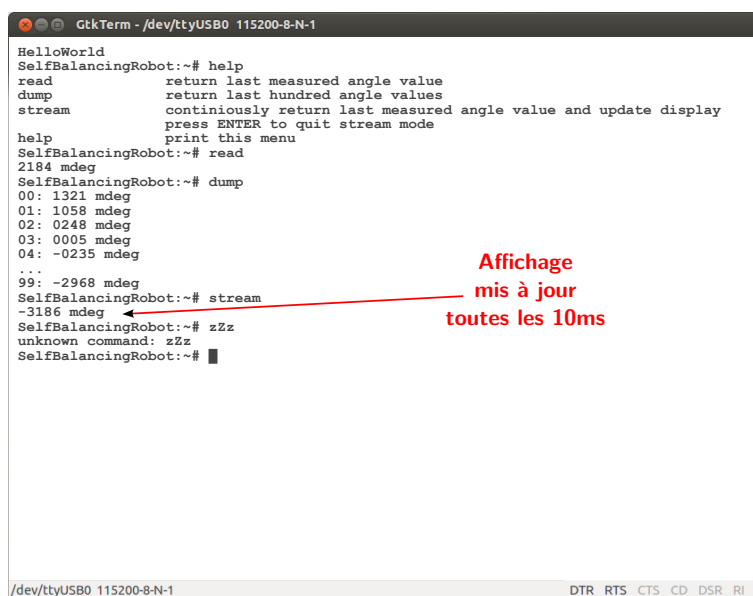


FIGURE 14 – Exemple d'échanges avec l'application.

1.2.2. Environnement logiciel

Une projet pré-configuré vous est fourni, incluant l'ensemble des sources nécessaires au bon fonctionnement (pilotes de périphériques, OS temps-réel, algorithmes d'asservissement).

Organisation du projet : l'organisation des fichiers sources respecte une séparation modulaire (chaque module/périphérique possède ses fichiers sources propres) ainsi qu'un découpage en couches d'abstraction successives. L'organisation résultante est illustrée sur la figure 15.

Les couches inférieures sont fournies sous la forme de fichiers sources C tel que résumé à la section 1.3.

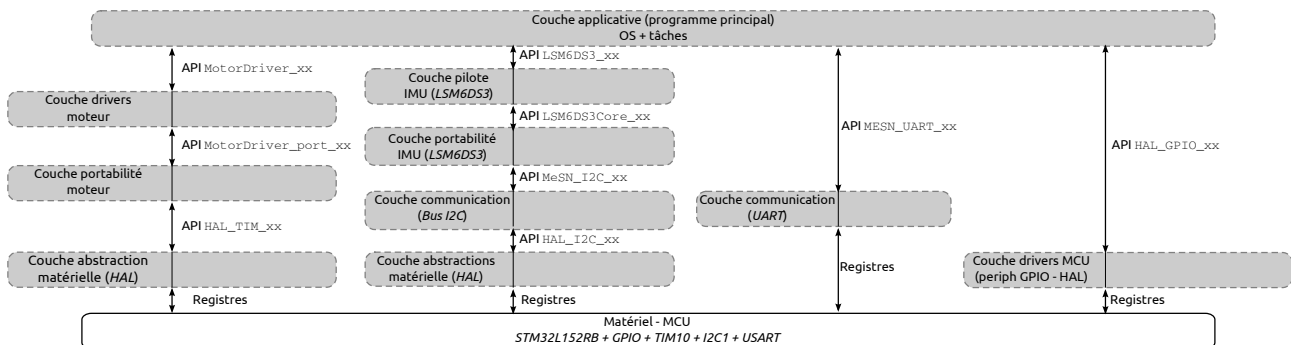


FIGURE 15 – Organisation logicielle de l'application, basée sur un modèle en couches.

Algorithmes d'automatique : les deux algorithmes utilisés fonctionnent en temps discret imposant une période d'échantillonnage fixée à 10 ms. Ces algorithmes sont fournis sous la forme d'une bibliothèque binaire pré-compilée à intégrer à votre projet tel qu'indiqué à la section 1.3.

1. **Observateur :** l'estimation de l'angle à partir des mesures de la cellule inertielle repose sur un algorithme d'observateur comme vous l'avez vu en cours d'automatique. L'API correspondante permettant d'appeler cet observateur est :

```
1  /**
2   * @brief Computes tilt angle of the robot according to observator algorithm.
3   * @param acc_mg : 32-bit integer of the sensed acceleration value in milli-g (1g=9.81
4   *               m per square second).
5   * @param rotAng_mDegSec : 32-bit integer of the sensed rotation value in milli-deg
6   *               per second.
7   * @return 32-bit integer value of the computed tilt angle in milli-degrees.
8   */
9  int32_t autoAlgo_angleObs(int32_t acc_mg, int32_t rotAng_mDegSec);
```

le paramètre de retour est la valeur de l'angle estimé entre l'axe du robot et l'axe vertical, exprimée en milli-degrés. Le paramètre `acc_mg` est la valeur de l'accélération mesurée sur l'axe X en milli-g, et le paramètre `rotAng_mDegSec` est la valeur de la vitesse angulaire mesurée sur l'axe Y exprimée en milli-deg/s.

2. **Loi de commande :** le calcul de la loi de commande repose sur un algorithme d'asservissement comme vous l'avez vu en cours d'automatique. L'API correspondante permettant d'appeler cet observateur est :

```
1  /**
2   * @brief Computes command value to be applied on robot's wheel according to
3   *       regulation algorithm.
4   * @param tiltAngle_mDeg : 32-bit integer of the tilt angle value of the robot in
5   *       milli-degrees.
6   * @param rotAng_mDegSec : 32-bit integer of the sensed rotation value in milli-deg
7   *       per second.
```

```

5  * @return 32-bit integer value of the computed command (expressed in per thousand :
6  *   in the range +/-1000).
7  */
7  int32_t autoAlgo_commandLaw(int32_t tiltAngle_mDeg, int32_t rotAng_mDegSec);

```

le paramètre de retour est la vitesse de rotation à appliquer aux roues du robot exprimée en pourmille de la vitesse maximale. Le paramètre `tiltAngle_mDeg` est la valeur de l'angle entre l'axe du robot et l'axe vertical exprimée en milli-degrés, et le paramètre `rotAng_mDegSec` est la valeur de la vitesse angulaire mesurée sur l'axe Y exprimée en milli-deg/s.

1.2.3. Assignations des broches et configuration matérielle

Le matériel utilisé repose sur la carte d'extension (shield) MeSN utilisée lors des TP MCU-32bits. La configuration matérielle et l'assignation des broches est donc identique.

1.3. Ressources mises à disposition

Le tableau 2 résume l'ensemble des ressources mises à votre disposition pour mener à bien votre projet.

Chemin	Fichier	Description
/Drivers/MeSN/LSM6DS3/	LSM6DS3.c/.h	Pilote de la centrale inertielle LSM6DS3
	LSM6DS3_port.c/.h	Couche portabilité du pilote LSM6DS3
/Drivers/MeSN/MotorDriver/	MotorDriver.c/.h	Pilote du driver de moteur
	MotorDriver_port.c/.h	Couche portabilité du pilote de moteur
/Drivers/MeSN/i2c/	MeSN_i2c.c/.h	Couche communication bus I2C
/Drivers/MeSN/UART/	mesn_uart.c/.h	Couche communication UART
/Drivers/MeSN/commun/	errorStatus.h	Fichier d'en-têtes intégrant une redéfinition de type utilisé par l'ensemble des pilotes
/Lib/Autom/	libSBR_autom_obs-corr.a/.h	Binaire implémentant les algorithmes de régulation
/Datasheets/	*.pdf	Datasheets des composants et boards utilisés
/Middlewares/Third_Party/FreeRTOS/	*.c/.h	Sources de FreeRTOS
/Middlewares/Trace Recorder/	*.c/.h	Outils de trace

TABLE 2 – Liste et arborescence des fichiers mis à votre disposition.

2. Travail à effectuer

2.1. Construction de l'architecture logiciel

L'architecture logicielle (découpage en tâches, synchronisation/communication inter-tâche et tâche-ISR, protection des ressources, ...) a déjà été construite en cours, en utilisant le formalisme présenté en figure 16. Il s'agissait d'un travail en équipe où chaque équipe a pu présenter son résultat. La version de votre enseignant est disponible sur e-campus.

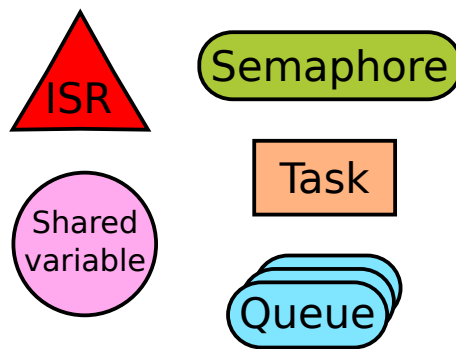


FIGURE 16 – Formalisme de représentation graphique de l'environnement logiciel.

2.2. Développement

- Pour commencer, téléchargez l'archive contenant le projet pré-configuré depuis e-campus. Décompressez-la puis importez le projet dans l'IDE STM32CubeIDE.
- Implémentez l'ensemble des fonctionnalités en respectant l'architecture logicielle de la section 2.1.

3. Pour les plus avancés

ATTENTION : Avant de passer à cet exercice, faites valider votre travail par votre enseignant.

Le cahier des charges évolue afin d'ajouter une nouvelle fonctionnalité à l'application sachant que l'ensemble des fonctionnalités précédentes doivent rester présentes et fonctionnelles. L'application doit maintenant proposer une interface homme-machine (IHM) locale reposant sur un afficheur LCD et un clavier matriciel. Voici les spécifications :

- Affichage sur un écran LCD 16 caractères et 2 lignes (le driver a été développé précédemment lors des TP de MCU32bits).
- Pilotage depuis un clavier matriciel à 12 touches (le driver a été développé précédemment lors des TP de MCU32bits).
- L'ensemble des commandes supportées via UART doivent être disponibles sur l'IHM locale : **Read**, **Dump** et **Stream**.
- Lors du démarrage, l'application doit présenter le message d'accueil "App Started..." sur l'afficheur pendant 1 seconde, puis afficher le menu principal.
- Le menu principal doit afficher le texte :
 - ligne 1 : "Choix Commande:"
 - ligne 2 : "1:Rd 2:Dmp 3:Str"
- Depuis le menu principal, l'appui sur les touches '1', '2' ou '3' du clavier doit exécuter la commande correspondante. Toutes autres touches entraînent l'affichage du message "Erreur!" sur la ligne 1 puis retour au menu principal.
- Pour chaque commande, la ligne 1 doit afficher le nom de la commande en cours et la ligne 2 les informations correspondantes :
 - Read : Affiche, pendant 1 seconde, la dernière valeur de l'angle mesuré (affichage en milli-degrés, sur 6 digits). Puis retour au menu principal.
 - Dump : Affiche les 100 dernières valeurs, puis retour au menu principal.
 - Stream : affichage en continu (toutes les 10 ms) de la valeur de l'angle. L'appui sur n'importe quelle touche permet de quitter le mode Stream et de revenir au menu principal.

Annexes - un OS temps réel en pratique

Table des matières

A. Formulaire de TP	33
B. Description de l'organisation des fichiers sources	33
B.1. Fichiers indépendants de l'application	33
B.2. Fichiers à modifier en fonction de l'application : <i>FreeRTOSConfig.h</i>	34
C. Extraits de l'API FreeRTOS et équivalent CMSIS-RTOS	35
C.1. Gestion des tâches	35
C.2. Gestion du noyau	36
C.3. Outils de synchronisation et communication	37
D. Outils de trace	40
D.1. Principe de fonctionnement	41
D.2. Intégration au projet	42
D.3. Utilisation	42

Avertissement : Ce document d'annexes ne contient aucun exercice de TP. Il constitue simplement un aide mémoire qui vous permettra de mettre en place, à partir de zéro, puis de travailler sur un projet de développement embarqué intégrant un OS temps réel. Il s'agit d'un complément pratique au cours que vous avez reçu, lisez-le attentivement. Les exercices proprement dits commencent au TP1.

Afin de mettre en pratique le cours que vous avez reçu concernant les OS à destination du monde de l'embarqué, et plus particulièrement les petits exécutifs temps-réel, l'OS retenu est FreeRTOS. Celui-ci est un système d'exploitation (ou exécutif, ou noyau) temps réel à faible empreinte mémoire pour microcontrôleurs. Son code est open source et distribué gratuitement sous une licence GPL modifiée.

Il faut garder à l'esprit qu'un OS, ce n'est que du logiciel (i.e. un programme qui peut s'exécuter). Dans le cas d'un OS open-source, il est de surcroît possible de consulter le code source dans le langage de programmation avec lequel il a été écrit (ici du langage C). En conséquence, faire le choix d'utiliser un OS lors d'un projet, consiste à intégrer les fichiers (sources ou binaires) de l'OS à votre projet, puis à construire votre application en utilisant l'API de programmation proposée par cet OS. D'un point de vue du développeur, un OS ajoute une couche supplémentaire d'abstraction entre le matériel et le code applicatif (figure 17).

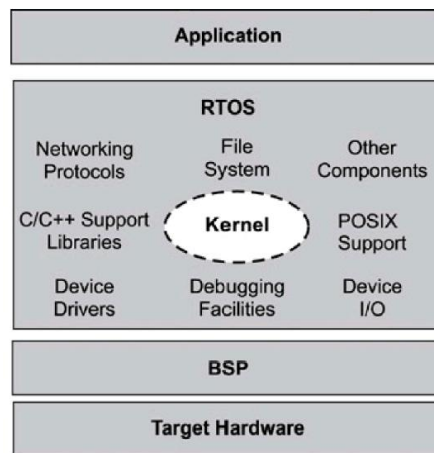


FIGURE 17 – L'OS vue comme une couche d'abstraction.

FreeRTOS est un OS largement portable. C'est à dire que seule une très faible portion du code source est dépendante de l'architecture matérielle sur laquelle il s'exécute tandis qu'une large partie est commune à toutes les architectures supportées.

Dans une première partie de ce préambule, nous exposerons comment sont organisés les fichiers sources de FreeRTOS puis nous verrons comment les intégrer à un projet de développement (illustré dans le cas d'une architecture *ARM-CortexM* et chaîne de compilation). Ensuite, nous détaillerons quelques fonctions les plus couramment utilisées de l'API de FreeRTOS ainsi que leurs équivalent CMSIS-RTOS. Enfin, nous terminerons sur un aperçu du fonctionnement des outils de trace d'exécution.

A. Formulaire de TP

Ce formulaire regroupe plusieurs énoncés de TP. Ceux-ci sont systématiquement constitués des éléments suivants :

1. Mise en contexte et rappel de cours concernant le sujet du TP.
2. Travail préparatoire à effectuer avant l'arrivée en séance, constitué d'une série de questions.
3. Travail à effectuer en séance.

Concernant cette dernière partie, deux pictogrammes sont à retenir, dont voici la signification :



Désigne une question dont la réponse est à fournir au CR de TP.



Désigne un travail pratique à effectuer sur la machine ou la maquette (édition de code source, manipulation de fichier, relevé de mesure, ...)

B. Description de l'organisation des fichiers sources

Les sources de FreeRTOS sont librement téléchargeables depuis le site officiel : <http://sourceforge.net/projects/freertos/files/>. Pour des raisons de simplicité, FreeRTOS est distribué en incluant les sources des portages vers tous les processeurs officiellement supportés. Par conséquent, le nombre de fichiers est relativement élevé mais seule une petite partie est finalement nécessaire lorsque l'on s'intéresse à une seule architecture en particulier. Dans le cadre de ces TP, une version modifiée de l'archive sera utilisée, qui ne contient que les fichiers strictement utiles au fonctionnement sur le matériel de TP.

B.1. Fichiers indépendants de l'application

A la racine, l'archive est séparée en deux sous répertoires :

- **/FreeRTOS** qui contient les fichiers sources du noyau temps-réel ainsi que les projets de démonstration pour chaque architecture officiellement supportée.
- **/FreeRTOS-Plus** qui contient tous les composants additionnels. Ce point ne sera pas traité ici, et ce répertoire peut donc être ignoré (*à supprimer*).

Le répertoire **/FreeRTOS** qui nous intéresse contient lui même quatre sous-répertoires :

- **/Demo** qui contient les projets de démonstration pour chaque architecture officiellement supportée. Ces projets n'ont d'autre but que d'illustrer les possibilités de FreeRTOS et ne sont d'aucune utilité si l'on souhaite démarrer un nouveau projet (*à supprimer*).
- **/License** qui contient les informations relatives aux licences d'utilisation de FreeRTOS.
- **/Source** qui contient les fichiers sources de l'OS.

B.1.1. Fichiers multi-plateformes (indépendants de l'architecture cible) :

Le code du noyau temps réel est contenu dans seulement 3 fichiers appelés **tasks.c**, **queue.c** et **list.c** auxquels s'ajoutent 3 fichiers optionnels¹⁶ **timers.c**, **event_groups.c** et **croutine.c**. Tous les 6 sont situés à la racine du répertoire **/FreeRTOS/Source**. Ces fichiers sont indépendants de l'architecture et sont donc identiques quelque-soit le processeur cible.

Les fichiers d'en-tête (.h) relatifs à ces fichiers sources sont situés dans le répertoire **/FreeRTOS/Source/Include**.

¹⁶. Ces 3 fichiers ne sont utiles que si l'on souhaite utiliser les fonctionnalités de *timers* logiciels, les *co-routines* et les *event-group*. Plus d'info à retrouver sur le site de FreeRTOS.org.

B.1.2. Couche portable (fichiers dépendants de la cible) :

Chaque processeur supporté nécessite cependant une petite partie de code spécifique à l'architecture concernée¹⁷. Il s'agit de la couche portable de FreeRTOS constituée de deux sous-répertoire :

- `/FreeRTOS/Source/Portable/[Compiler]/[Architecture]` où *[Compiler]* désigne la chaîne de compilation utilisée et *[Architecture]* désigne le processeur sur lequel doit s'exécuter le portage. Dans notre cas, il s'agit donc du répertoire qui contient un fichier source C, `port.c`. Il contient également 1 fichier d'en-têtes (.h) relatif à la couche portable de FreeRTOS.
- `/FreeRTOS/Source/Portable/MemMang` contenant les fichiers relatifs à la gestion de la mémoire par FreeRTOS (tas). Les différents profils de gestion de la mémoire disponibles sont contenus dans les fichiers `heap_x.c`. Dans notre cas nous n'utiliserons que le modèle de mémoire `heap_1.c`.

Les autres répertoires contenus dans `/FreeRTOS/Source/Portable/` concernent les autres architectures supportées.

B.1.3. Résumé :

Finalement, il ne reste que l'arborescence suivante :

- `/FreeRTOS/Source/` contenant les fichiers source du noyau.
- `/FreeRTOS/Source/include/` contenant les fichiers d'en-têtes du noyau.
- `/FreeRTOS/Source/portable/MemMang/` contenant l'implantation du modèle de mémoire.
- contenant les fichiers spécifiques à l'architecture.

Les fichiers présentés jusqu'ici sont indépendants du projet développé. Ils permettent uniquement d'avoir accès aux fonctionnalités proposées par FreeRTOS, charge à vous de développer votre propre application (constituée de différentes tâches) à l'aide de ces outils. Cela signifie que vous n'avez aucune raison de les modifier, à moins de vouloir apporter votre contribution au développement de FreeRTOS ou pour effectuer son portage vers une nouvelle architecture non-supportée officiellement. **Attention, ne pas les modifier ne signifie pas qu'il ne faut pas les ouvrir. Au contraire, il vous sera très instructif de lire le code source de FreeRTOS pour mieux en appréhender son fonctionnement !**

Si ces fichiers n'ont pas à être modifiés, nous pouvons donc utiliser les mêmes pour différents projets.

B.2. Fichiers à modifier en fonction de l'application : *FreeRTOSConfig.h*

A l'inverse des fichiers précédents, il existe un fichier (unique) qui est spécifique à chaque application développée. Ce fichier est le seul à être physiquement sorti de l'arborescence de fichiers de FreeRTOS et vous devez en créer une nouvelle copie pour chaque nouveau projet développé. Il s'agit du fichier **FreeRTOSConfig.h**.

Ce fichier ne contient que des macros permettant de configurer le noyau avant compilation en fonction des besoins de l'application. Par exemple, si vous souhaitez utiliser le mode préemptif, il faut que la macro `configUSE_PREEMPTION` soit à 1, sinon c'est le mode coopératif qui sera utilisé. Il apparaît donc clairement que ce type de configuration est laissée au choix du développeur. Prenons deux cas de figure pour illustrer le rôle de ce fichier :

- toutes (ou presque) les macros sont à "1" : dans ce cas, vous pouvez utiliser toutes l'API et fonctionnalités proposées par FreeRTOS. Cependant, la taille du code généré aura fortement augmentée car toutes les fonctions auront été générées, linkées et seront embarquées dans la mémoire programme du processeur.

17. Notamment la partie concernant la sauvegarde de contexte, la génération du Tick d'OS et la gestion du tas.

- toutes les macros sont à "0" : dans ce cas, vous n'aurez accès qu'aux fonctionnalités minimales de FreeRTOS et les fonctions (donc les macros) devront être ajoutées en fonction des besoins afin de ne pas gaspiller inutilement les ressources mémoires du processeurs.

L'archive de TP contient un exemplaire de ce fichier de configuration. Il est ensuite de votre ressort de modifier les macros utiles¹⁸.

C. Extraits de l'API FreeRTOS et équivalent CMSIS-RTOS

Une fois que vous avez réussi à créer un projet incluant les sources (ou binaires) de votre OS, vous pouvez alors commencer à construire votre application en utilisant les outils que l'OS met à votre disposition. Cela se fait par l'interface de programmation de l'OS, c'est à dire une bibliothèque de fonction appelée API.

Chaque OS propose une API qui lui est propre. Cependant, afin de faciliter le travail des développeurs et de rendre le code d'une application exécutable sur plusieurs OS différents (portabilité), il existe des API standardisées. La plus connue d'entre-elle est l'API POSIX, que respectent notamment les systèmes UNIX et partiellement Windows.

Dans le monde de l'embarqué, la situation est beaucoup plus hétérogène et il existe une multitude d'OS aux caractéristiques différentes et largement incompatibles entre-eux. Néanmoins, dans un soucis de proposer un environnement de développement cohérent pour ses architectures Cortex, la société ARM a standardisé une API de programmation des OS appelée **CMSIS-RTOS**. Les différentes sociétés proposant des OS fonctionnant sur architecture Cortex peuvent alors opter pour une compatibilité CMSIS-RTOS. C'est cette API de programmation que nous allons découvrir en TP, l'OS en lui-même étant FreeRTOS développé par le société *Real Time Engineers Ltd*.

La documentation complète des API FreeRTOS et CMSIS-RTOS est disponible en ligne aux adresses : <http://www.freertos.org/a00106.html> et <http://www.keil.com/pack/doc/CMSIS/RTOS/html/modules.html>. En voici un petit extrait pour les fonctions que vous aurez le plus à utiliser en séance.

C.1. Gestion des tâches

C.1.1. Création de taches

Crée une tâche et l'ajoute à la liste des tâches candidates à l'exécution (état ready).

- API FreeRTOS :

```
1 BaseType_t xTaskCreate(TaskFunction_t pvTaskCode, const char * const pcName, unsigned short
  usStackDepth, void *pvParameters, UBaseType_t uxPriority, TaskHandle_t *pvCreatedTask);
```

- API CMSIS :

```
1 osThreadId osThreadCreate(const osThreadDef_t *thread_def, void *argument);
```

- Exemple d'utilisation :

```
1 #include "cmsis_os.h"
2
3 void Thread1_function (void);           // function prototype for Thread1
4 osThreadId Thread1_identifier;         // Declare handler for Thread1
5
6 void ThreadCreate_example (void) {
7     :
```

¹⁸. La description des différentes options proposés par le fichier *FreeRTOSConfig.h* peut être obtenue à l'adresse <http://www.freertos.org/a00110.html>. Un exemplaire en est fournie avec les projets de démonstration pour chaque architecture officiellement supportée.

```

8  /* Thread1 definition */
9  osThreadDef (Thread1_name, Thread1_function, osPriorityNormal, 0, ...);
10 /* Thread1 creation */
11 Thread1_identfier = osThreadCreate (osThread (Thread1_name), NULL);
12 :
13 }

```

C.1.2. Retarder une tâche

Permet de retarder l'exécution d'une tâche en la passant à l'état bloqué pendant un temps déterminé.

- API FreeRTOS :

```
1 void vTaskDelay( const TickType_t xTicksToDelay );
```

- API CMSIS :

```
1 osStatus osDelay (uint32_t millisec);
```

- Exemple d'utilisation :

```

1 #include "cmsis_os.h"
2
3 void Thread_1 (void const *arg) {    // Thread function
4     osStatus status;                // capture the return status
5
6     :
7     status = osDelay (1000);        // suspend thread execution for 1 second
8     :
9 }

```

C.1.3. Récupérer la valeur des ticks d'OS

Permet de connaître la valeur actuel du compteur de ticks, c'est à dire le nombre de ticks écoulé depuis le démarrage de l'OS.

- API FreeRTOS :

```
1 volatile TickType_t xTaskGetTickCount( void );
```

- API CMSIS :

```
1 uint32_t osKernelSysTick (void)
```

C.2. Gestion du noyau

C.2.1. Réordonnancement

Permet de forcer un réordonnancement.

- API FreeRTOS :

```
1 taskYIELD();
```

- API CMSIS :

```
1 osStatus osThreadYield ( void );
```

C.2.2. Démarrage de l'ordonnanceur

Démarre l'ordonnanceur temp réel. Après son démarrage, c'est le noyau qui contrôle quelle tâche s'exécute et quand.

- API FreeRTOS :

```
1 | void vTaskStartScheduler( void );
```

- API CMSIS :

```
1 | osStatus osKernelStart ( void );
```

C.3. Outils de synchronisation et communication

C.3.1. Semaphore

Outil dédié à la synchronisation de tâches ou entre tâche et ISR.

Création d'un semaphore binaire :

- API FreeRTOS :

```
1 | SemaphoreHandle_t xSemaphoreCreateBinary( void );
```

- API CMSIS :

```
1 | osSemaphoreId osSemaphoreCreate( const osSemaphoreDef_t * semaphore_def, int32_t count);
```

Test d'un semaphore :

- API FreeRTOS :

```
1 | xSemaphoreTake( SemaphoreHandle_t xSemaphore, TickType_t xTicksToWait);
```

- API CMSIS :

```
1 | int32_t osSemaphoreWait ( osSemaphoreId semaphore_id, uint32_t millisec);
```

Vente d'un semaphore :

- API FreeRTOS :

```
1 | xSemaphoreGive( SemaphoreHandle_t xSemaphore );
```

- API CMSIS :

```
1 | osStatus osSemaphoreRelease ( osSemaphoreId semaphore_id);
```

Exemple d'utilisation :

```

1  #include "cmsis_os.h"
2  /* Thread declarations */
3  osThreadId T1_handler; // ID for thread 1
4  osThreadId T2_handler; // ID for thread 2
5  /* Semaphore declaration */
6  osSemaphoreId sem_handler; // Semaphore ID
7
8  /*
9  Thread 1 - High Priority - Active every 3ms
10 */
11 void T1_function (void) {
12     int32_t val;
13
14     while (1) {
15         osDelay(3); // Pass control to other tasks for 3ms
16         val = osSemaphoreWait (sem_handler, 1); // Wait 1ms for the free semaphore
17         if (val > 0) {
18             : // OK, the interface is free now, use it.
19             osSemaphoreRelease (sem_handler); // Return a token back to a semaphore
20         }
21     }
22 }
23
24 /*
25 Thread 2 - Normal Priority - looks for a free semaphore and uses
26 the resource whenever it is available
27 */
28 void T2_function (void) {
29     while (1) {
30         osSemaphoreWait (sem_handler, osWaitForever); // Wait indefinitely for a free
31         semaphore
32         // OK, the interface is free now, use it.
33         :
34         osSemaphoreRelease (sem_handler); // Return a token back to a semaphore
35     }
36 }
37
38 /* Init */
39 void StartApplication (void) {
40     /* Thread definitions */
41     osThreadDef(T1NAME, T1_function, osPriorityHigh, 0, ...);
42     osThreadDef(T2NAME, T2_function, osPriorityNormal, 0, ...);
43     /* Thread creations */
44     T1_handler = osThreadCreate(osThread(T1NAME), NULL);
45     T2_handler = osThreadCreate(osThread(T2NAME), NULL);
46
47     /* Semaphore definition */
48     osSemaphoreDef(sem_handler);
49     /* Semaphore creation */
50     sem_handler = osSemaphoreCreate(osSemaphore(sem_handler), 1);
51     :
52 }

```

C.3.2. Mutex

Outil dédié à la gestion de l'exclusion mutuelle vis-à-vis d'une ressource partagée.

Création d'un mutex :

- API FreeRTOS :

```

1 | SemaphoreHandle_t xSemaphoreCreateMutex( void );

```

- API CMSIS :

```
1 | osMutexId osMutexCreate ( const osMutexDef_t * mutex_def);
```

Prise d'un mutex :

- API FreeRTOS :

```
1 | xSemaphoreTake( SemaphoreHandle_t xSemaphore, TickType_t xTicksToWait);
```

- API CMSIS :

```
1 | osStatus osMutexWait ( osMutexId mutex_id, uint32_t millisec);
```

Libération d'un mutex :

- API FreeRTOS :

```
1 | xSemaphoreGive( SemaphoreHandle_t xSemaphore );
```

- API CMSIS :

```
1 | osStatus osMutexRelease ( osMutexId mutex_id);
```

C.3.3. Queue de message

Outil dédié à la communication entre tâches ou entre tâche et ISR.

Création d'une queue de message :

- API FreeRTOS :

```
1 | QueueHandle_t xQueueCreate( UBaseType_t uxQueueLength, UBaseType_t uxItemSize);
```

- API CMSIS :

```
1 | osMessageQId osMessageCreate( const osMessageQDef_t * queue_def, osThreadId thread_id);
```

Écriture d'une donnée dans une queue de message :

- API FreeRTOS :

```
1 | BaseType_t xQueueSend( QueueHandle_t xQueue, const void * pvItemToQueue, TickType_t xTicksToWait);
```

- API CMSIS :

```
1 | osStatus osMessagePut( osMessageQId queue_id, uint32_t info, uint32_t millisec);
```

Lecture d'une donnée dans une queue de message :

- API FreeRTOS :

```
1 | BaseType_t xQueueReceive( QueueHandle_t xQueue, void *pvBuffer, TickType_t xTicksToWait);
```

- API CMSIS :

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

Exemple d'utilisation :

```

1  #include "cmsis_os.h"
2
3  osThreadId tid_thread1;           // ID for thread 1
4  osThreadId tid_thread2;           // ID for thread 2
5
6  osMessageQDef(MsgBox, 5, uint32_t); // Define message queue
7  osMessageQId MsgBox;               // ID for message queue
8
9
10 /* Thread 1: Send thread */
11 void send_thread (void) {
12     int32_t value;
13
14     while(1){
15         value = value + 2;           // Set the message content
16         osMessagePut(MsgBox, value, 0); // Send Message with no timeout
17         :
18         osDelay(100);               // Wait few millisecs
19     }
20 }
21
22 /* Thread 2: Receive thread */
23 void recv_thread (void) {
24     osEvent evt;
25     uint32_t mess;
26
27     while (1) {
28         evt = osMessageGet(MsgBox, osWaitForever); // wait for message
29         if (evt.status == osEventMessage) {
30             mess = evt.value.v;
31             printf ("\nValue: %d", mess);
32         }
33     }
34 }
35
36 /* Thread definitions */
37 osThreadDef(send_thread, osPriorityNormal, 1, 0);
38 osThreadDef(recv_thread, osPriorityNormal, 1, 2000);
39
40 /* Init */
41 void StartApplication (void) {
42     MsgBox = osMessageCreate(osMessageQ(MsgBox), NULL); // create msg queue
43     tid_thread1 = osThreadCreate(osThread(send_thread), NULL);
44     tid_thread2 = osThreadCreate(osThread(recv_thread), NULL);
45     :
46 }

```

D. Outils de trace

Sur un système multitâches, une fois que le scheduler est démarré, c'est lui qui contrôle quelle tâche s'exécute et à quel moment. En cas de dysfonctionnement, il peut s'avérer difficile pour le développeur de déterminer quel en est la cause et les techniques de débogage classiques (point d'arrêt, watch-windows, ...) sont relativement limitées dans ce cas. Les outils de trace peuvent alors être d'un grand secours. En effet, ils permettent de récupérer une trace de l'évolution du système au cours de l'exécution. Autrement dit, il devient possible de savoir quelle tâche s'exécute, à quel moment et quels sont les liens entre les différentes tâches et événements extérieurs. La figure 18 illustre un exemple de trace d'exécution dans le cas de FreeRTOS.

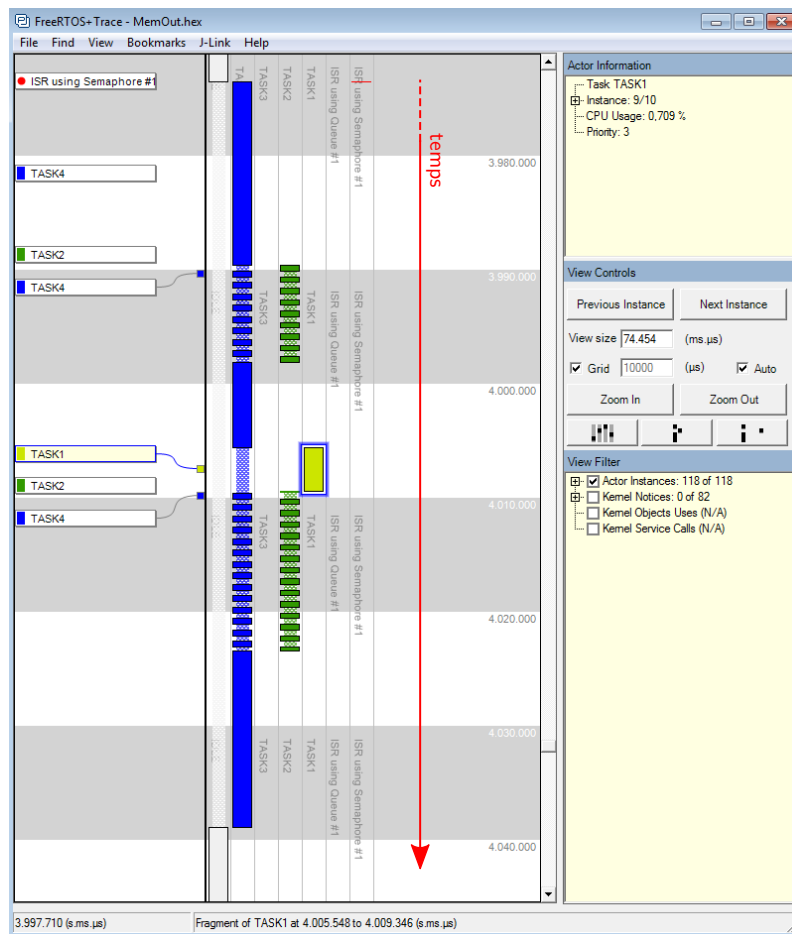


FIGURE 18 – Exemple de trace d'exécution sous FreeRTOS pour un système constitué de 4 tâches, dont deux (TASK3 et TASK4) sont synchronisées sur des ISR, tandis que les deux autres (TASK1 et TASK2) échangent de l'information via queue de message. Toutes sont en concurrence vis-à-vis d'une ressource partagée, protégée par sémaphore.

D.1. Principe de fonctionnement

Nous utiliserons les outils de trace développés par la société *Percepio*. Ils sont évidemment compatibles avec FreeRTOS, mais également avec d'autres OS.

Le principe de fonctionnement est le suivant, comme illustré sur la figure 19 :

1. Un espace en mémoire donnée (RAM) va être réservé afin de collecter les différents événements traçables au cours du temps.
2. Au cours de l'exécution, si les outils de trace sont activés, chaque appel de fonctions systèmes (réordonnancement, synchronisation, délai, ...) va alors lui-même appeler une fonction de trace chargée d'inscrire l'événement dans l'espace mémoire dédié.
3. Lors du débogage, il est alors possible d'arrêter l'exécution du programme afin de venir lire le contenu de la mémoire RAM (dump de la mémoire).
4. L'analyse du contenu de l'espace réservé à la collecte des traces d'exécution permet alors de reconstruire l'emploi du temps du CPU.

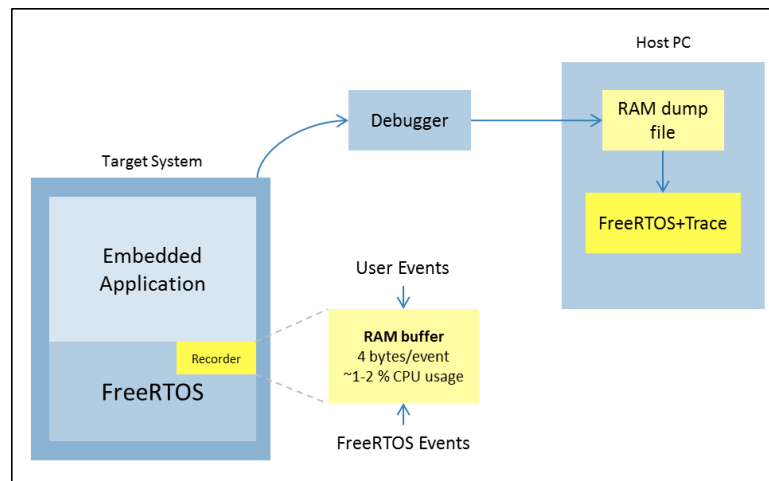


FIGURE 19 – Principe de fonctionnement des outils de trace de la société *Percepio*.

D.2. Intégration au projet

Les outils de traces de *Percepio* reposent sur un ensemble de fichiers sources à ajouter au projet. Si l'OS que vous utilisez ainsi que l'architecture sur laquelle s'exécute votre programme sont supportés par *Percepio*, alors l'intégration est relativement aisée. Il suffit de suivre les indications de la documentation en ligne : http://percepio.com/docs/FreeRTOS/manual/Recorder.html#Recorder_Library_FreeRTOS_Integration.

En pratique, il faut récupérer les fichiers sources de *Percepio* (fichiers **TrcSnapshotRecorder.c**, **TrcStreamingRecorder.c** et **TrcKernelPort.c**) puis les ajouter au projet. Ensuite, il convient de configurer la chaîne de compilation afin d'indiquer les fichiers d'en-têtes des outils de trace contenus dans le sous-répertoire **/include** et **/config**.

Dans le cadre de ces TP, ce travail a déjà été fait puisque le projet qui vous a été fourni contient déjà ces fichiers et qu'il est configuré pour travailler avec.

D.3. Utilisation

D.3.1. Mise en place

Une fois le projet fonctionnel, il convient d'activer les outils de trace dans le code de votre application.

- Éditez le fichier d'en-tête **FreeRTOSConfig.h** afin d'autoriser les traces de FreeRTOS en passant à 1 la macro `configUSE_TRACE_FACILITY`.
- Vérifiez la configuration des outils de trace contenue dans le fichier **config/trcConfig.h**, notamment :

- La version de FreeRTOS utilisée via la macro

```
1 | #define TRC_CFG_FREERTOS_VERSION TRC_FREERTOS_VERSION_10_0_0
```

- L'architecture matérielle utilisée via la macro

```
1 | #define TRC_CFG_HARDWARE_PORT TRC_HARDWARE_PORT_ARM_Cortex_M
```

- Appeler la fonction `vTraceEnable(TRC_START)` depuis le `main()` afin d'initialiser et de démarrer l'enregistrement de trace. Cette fonction doit être appelée après l'initialisation du matériel, mais préalablement à toute création d'un objet lié à l'OS (tâches etc.). Dans le cadre de nos TP, placez cet appel au début du `main()`, juste après l'initialisation des périphériques par HAL.

D.3.2. Exploitation

Une fois l'application en fonctionnement (code en cours d'exécution) les traces seront enregistrées dans l'espace mémoire réservé. Il suffit alors d'arrêter le programme à l'aide du debugger pour ensuite récupérer le contenu de la mémoire RAM.

Les outils de Trace de Perceprio étant directement intégrables à l'IDE STM32CubeIDE, cette manipulation est relativement aisée. Suivez l'une des 2 méthodes ci-dessous :

1. Méthode depuis l'IDE STM32CubeIDE :

- Arrêtez l'exécution du programme à l'aide du debugger (bouton "pause", ou point d'arrêt).
- Cliquez sur **Perceprio** > **Save Snapshot Trace** présent dans la barre de menu¹⁹.

Le logiciel TraceAnalyzer doit s'ouvrir et afficher la trace qui vient d'être capturée.

2. Méthode depuis le logiciel TraceAnalyzer :

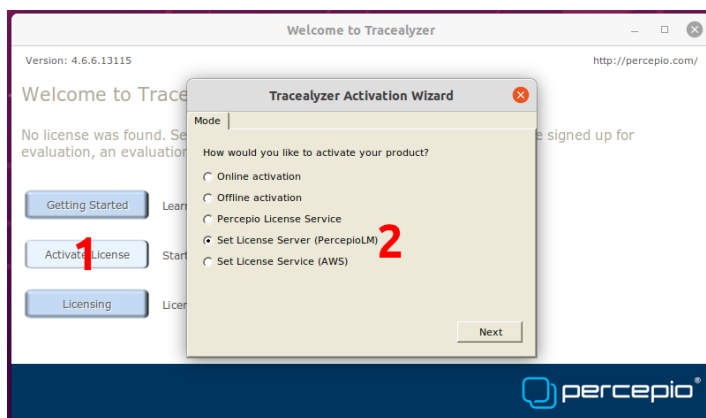
- Ouvrez le logiciel TraceAnalyzer puis cliquez sur **Read Snapshot Trace**. Sélectionnez **Snapshot engine = API connection** et **interface = STLink** puis cliquez sur **Read Snapshot**.

Le logiciel TraceAnalyzer affiche maintenant la trace qui vient d'être capturée.

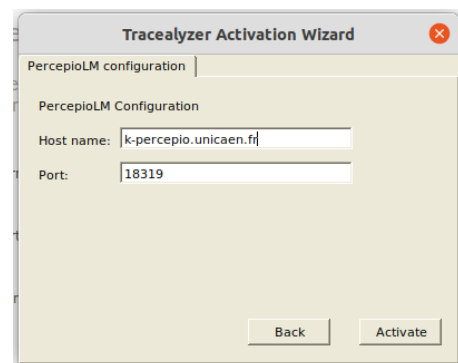
Quelque soit la méthode, répétez simplement les mêmes étapes à chaque fois que vous arrêtez votre programme afin de mettre à jour la trace.

Le logiciel TraceAnalyzer nécessite une licence. L'activation de la licence se fait via un serveur de licence (accessible depuis le réseau *Unicaen* ou *WifiCampus*), en suivant les étapes ci-dessous :

1. Choisissez l'option d'activation "serveur de licence" : figure 20a.
2. Renseignez les paramètres de configuration comme sur la figure 20b.
3. Et voilà.



(a) Étape 1.



(b) Étape 2.

FIGURE 20 – Activation de la licence du logiciel TraceAnalyzer via le serveur de licence.

Un plugin TraceAnalyzer pour l'IDE STM32CubeIDE est disponible depuis le Marketplace de l'IDE : **Help** > **Eclipse Marketplace**. Onglet **Search** > **Find**, taper **Perceprio** puis **Go**. Sélectionner le plugin **Perceprio Trace Exporter** > **Install**. Cela facilite l'utilisation.

¹⁹. Si le menu Perceprio n'apparaît pas dans la barre de menu, il est nécessaire d'ajouter le plugin Perceprio à l'IDE : **Help** > **Eclipse Marketplace**. Onglet **Search** > **Find**, taper **Perceprio** puis **Go**. Sélectionner le plugin **Perceprio Trace Exporter** > **Install**. Suivre la procédure.

