



Électronique et Technologies Numériques
Systèmes Embarqués Temps-Réel

Rapport de Travaux Pratiques

Utilisation de l'Exécutif Temps-Réel FreeRTOS sur carte
SAMD21

Janvier 2023 – ETN5 Promo 2023

Auteurs:
Téo BITON
Louison GOUY

Encadrant:
Olivier Pasquier

Contents

Introduction	1
1 Réponse aux questions	2
1.1 Politique d'ordonnancement	2
1.2 Gestion de la mémoire	2
1.3 Fréquence de clignotement	3
2 Ressource FreeRTOS	5
2.1 Les tâches	5
2.2 Sémaaphores	6
2.3 Files de message	7
3 Développement de l'application	8
3.1 Cahier des charges	8
3.2 Éléments de l'application	8
3.3 Résultat	9
Conclusion	12
Appendix	14
main.c	14
FreeRTOSConfig.h	19

List of Figures

1 Structure fonctionnelle initiale de l'application à développer	8
2 Structure fonctionnelle de l'application à développer	9
3 Affichage de l'écran au lancement de l'application	10
4 Affichage de l'écran après une session de mesure du temps de réaction	11

Introduction

Dans le cadre de l'option Systèmes-Embarqués Temps-Réels en cinquième année d'Électronique et Technologies Numériques, le module ME3 "Système Temps-Réels" est proposé aux étudiants. Ce module a pour objectif final de développer une application temps-réel simple de mesure de temps de réaction. Pour y parvenir, nous découvrons au fil des séances les informations permettant de conduire au développement de l'application et à l'implantation des éléments requis.

Pour mener à bien ce projet, nous utiliserons l'environnement de développement *Microchip Studio* déjà utilisé en quatrième année. La plus-value est l'importation d'un exécutif Temps-Réel au projet, FreeRTOS, dont nous découvrirons la configuration progressivement. Enfin, la cible matérielle est un système embarqué composé d'une carte SAMD21XPLAINEDPRO et d'une carte d'extension OLED1XPLAINEDPRO. Ainsi, le présent rapport présentera dans un premier temps les différents

Les codes sources demandés de l'application (fichiers *main.c* et *FreeRTOSConfig.h*) sont disponibles en annexe du présent document. De plus, pour en prouver le fonctionnement, des photos prises lors du fonctionnement sont ajoutées en rapport (figures 3 et 4). Bien conscients que ces photos ne permettent pas d'attester du fonctionnement de l'application, nous avons également joint une vidéo dans l'archive.

1 Réponse aux questions

1.1 Politique d'ordonnancement

Le rôle d'un ordonnanceur est de résoudre le problème de ressources partagées. Dans le cas d'un RTOS la ressource centrale est le(s) cœur(s) de processeur. La question étant, sur quels éléments se baser pour déterminer quelle tâche peut y accéder. Il existe un grand nombre de choix bien documenté dans la littérature. La politique implémentée dans FreeRTOS est l'ordonnancement préemptif à priorités fixes (Fixed priority pre-emptive scheduling). Dans le cas de priorité identique un round-robin est appliqué. Afin de s'approprier le code du noyau, il est demandé d'*identifier la procédure en charge de la politique d'ordonnancement en faisant le lien avec les constantes définies dans le fichier FreeRTOSConfig.h. Quelle(s) procédure(s) faudrait-il modifier pour changer la politique ?*

Les procédures liées à l'ordonnancement sont définies dans le fichier `task.c`. Parmi elles, on retrouve `vTaskSwitchContext()`. Qui, comme son nom l'indique, assure le changement de contexte des tâches. Mais, avant de faire le changement effectif, elle doit identifier la tâche la plus prioritaire. Le bloc de code suivant est extrait de la fonction.

```
1 /* Select a new task to run using either the generic C or port optimised asm code. */
2 taskSELECT_HIGHEST_PRIORITY_TASK();
```

Ainsi, pour être plus précis, c'est la macro `taskSELECT_HIGHEST_PRIORITY_TASK` qui est en charge d'appliquer la politique de préemption à priorité fixe. Si on souhaite la changer il faut la remplacer. Dans le cas d'un système temps réel cela n'est pas simple car il faut probablement, pour atteindre de bonnes performances, écrire le code en assembleur. Il est toutefois, possible d'ajuster la politique avec les paramètres de configuration mis à disposition. En effet, il est possible de désactiver la préemption dans `config.h` avec `configUSE_PREEMPTION`. Aussi, les priorités dites fixes, car non modifiée par le RTOS, sont en fait modifiable durant l'exécution. Cela aura pour résultat d'influencer la politique.

1.2 Gestion de la mémoire

Le noyau du RTOS a besoin de RAM chaque fois qu'une tâche, une file de message, un mutex, un délai logiciel, un sémaphore ou un groupe d'événements est créé. La RAM peut être automatiquement allouée dynamiquement à partir du tas, ou elle peut être fournie par le développeur. La question posée est la suivante : *Qu'impliquent les 5 politiques de gestion de la mémoire par FreeRTOS ?*

FreeRTOS offre plusieurs schémas de gestion du tas qui varient en complexité et en fonctionnalités. Il est possible d'utiliser deux politiques de gestion mémoire simultanément. Cela permet par exemple, aux piles des tâches et aux autres objets du RTOS d'être placés dans la RAM interne rapide, et aux données de l'application d'être placées dans la RAM externe plus lente [3].

heap 1

C'est le fonctionnement le plus simple, il ne permet pas de libérer de la mémoire une fois allouée. Malgré cela, il convient à la majorité des applications temps réel. En effet, quand les contraintes mémoire sont rudes, il est usuel de déclarer les tâches et objets RTOS statiquement au démarrage. Le programme de heap 1 divise une zone mémoire en petits blocs. La taille globale étant donnée par `configTOTAL_HEAP_SIZE`.

heap 2

Le second mode propose une fonctionnalité supplémentaire, il permet de libérer de la mémoire, mais ne permet pas la fusion de blocs libres adjacents. Il peut être utilisé dans le cas où l'application affecte de la mémoire dynamiquement en connaissance des risques. En effet, la taille des blocs alloués doit être constante sans quoi la mémoire risque de se fragmenter au cours de l'exécution et finir par causer une erreur d'allocation.

heap 3

Ce mode est bien différent des autres puisqu'il encapsule des fonctions standard `malloc()` et `free()`.

L'environnement de compilation doit contenir le code de ces deux fonctions. Les inconvénients à cela sont le non-déterminisme de l'exécution et l'occupation mémoire. En effet, d'un appel à l'autre le temps d'exécution de `malloc` et `free` diffère.

heap 4

Ce schéma s'apparente au heap 2 avec en plus la fusion des blocs libres adjacents pour éviter la fragmentation. Cela le rend lui aussi non déterministe. Il reste cependant bien plus performant que la plupart des implantations de `malloc`.

heap 5

Le cinquième est dernier mode de gestion est le plus avancées mis à disposition par FreeRTOS. Il propose de la possibilité d'étendre le tas (heap) à travers plusieurs zones de mémoire non adjacentes.

1.3 Fréquence de clignotement

Comment faire varier la fréquence de clignotement de la LED ? (plusieurs possibilités)

Il existe plusieurs façons de faire varier la fréquence de clignotement de la LED. Certaines solutions ont un impact uniquement sur le clignotement de LED dans la tâche "LEDControl" du programme main-LED_BP.c donné en annexe. D'autres solutions ont un impact sur le fonctionnement générale du système. Elles seront donc classées de la plus simple (ou la plus "propre") à mettre en place à la moins judicieuse pour faire varier la fréquence de clignotement de la LED.

La première solution consiste à ajouter un compteur dans la tâche "Clock" et de tester sa valeur, diminuant ainsi la fréquence d'envoi de la séaphore à la tâche "LEDControl". Une implémentation possible est celle-ci:

```

1 void vCodeClock(void * pvParameters)
2 {
3     uint8_t cpt = 0;
4
5     for(;;)
6     {
7         vTaskDelay(HORLOGE_TASK_DELAY);
8
9         // Fréquence divisée par 2
10        if (cpt == 1) {
11            xSemaphoreGive( xSem_H1 );
12            cpt = 0;
13        } else {
14            cpt++;
15        }
16    }
17 }
```

Une deuxième solution consiste à faire varier la constante `HORLOGE_TASK_DELAY`. Cette constante définit la fréquence de fonctionnement globale de l'ordonnanceur. Dans une application avec plus de tâches ou si plus de sémaphores étaient envoyées par la tâche "Clock", modifier cette constante aurait un impact sur les fréquences d'appel des tâches.

```

1 // Déclaration de la constante de temps d'attente dans la tâche "Clock"
2 #define HORLOGE_TASK_DELAY 1
```

Dans la même idée, mais d'une façon moins appropriée, il est possible de faire varier la fréquence de clignotement de la LED en rajoutant des appels à la fonction `vTaskDelay`.

```

1 void vCodeClock(void * pvParameters)
2 {
3     for(;;)
4     {
5         // Deux appels de vTaskDelay divisent par deux la fréquence
6         vTaskDelay(HORLOGE_TASK_DELAY);
7         vTaskDelay(HORLOGE_TASK_DELAY);
```

```

8     xSemaphoreGive( xSem_H1 );
9 }
10 }
```

Les solutions présentées ci-dessus interviennent directement dans l'application. Une autre possibilité cette fois réside dans la modification de l'OS via le fichier FreeRTOSConfig.h. Dans ce fichier il est possible de modifier la fréquence de fonctionnement de l'OS.

```
1 #define configTICK_RATE_HZ          ( ( TickType_t ) 1000 )
```

Cette ligne définit le nombre de Ticks d'horloge par secondes envoyés par l'OS. Ici, la valeur associée est 1000, soit une fréquence de 1 kHz et 1000 Ticks/secondes. Modifier cette valeur permet de faire varier l'horloge envoyée par l'OS aux applications et donc de toute application qui tourne sur l'OS. C'est une possibilité pour faire varier la fréquence de clignotement de la LED, mais peu adaptée.

Enfin, la dernière solution proposée est de modifier directement l'horloge source en provenance du microcontrôleur. L'horloge de référence du micro peut également être modifiée dans le fichier FreeRTOSConfig.h, à la ligne suivante:

```
1 #define configCPU_CLOCK_HZ          ( system_clock_source_get_hz( SYSTEM_CLOCK_SOURCE_0SC8M ) )
```

Par défaut, c'est un oscillateur à 8 MHz qui est l'horloge de référence du CPU. Dans le fichier conf_clocks.h, il est possible d'appliquer des facteurs de division d'horloge prédéfinis.

```

1 # define CONF_CLOCK_CPU_DIVIDER           SYSTEM_MAIN_CLOCK_DIV_1
2 ...
3 /* SYSTEM_CLOCK_SOURCE_0SC8M configuration - Internal 8MHz oscillator */
4 # define CONF_CLOCK_0SC8M_PRESCALER        SYSTEM_0SC8M_DIV_1
```

Ces facteurs de division (de 1 à 1024, A VÉRIFIER) divisent l'horloge avant qu'elle ne soit envoyé à l'OS par le CPU. Il est également possible de changer de source d'horloge en choisissant par exemple une horloge à 32 kHz.

```
1 #define configCPU_CLOCK_HZ          ( system_clock_source_get_hz( SYSTEM_XOSC32K_STARTUP_65536 ) )
```

Peu importe la méthode choisie pour modifier la fréquence du CPU, c'est une modification très risquée. En effet, il faut s'assurer au préalable que le taux d'occupation permet une telle modification; diminuer la fréquence du CPU, c'est augmenter le taux d'occupation et ralentir fortement les tâches, notamment dans le cas d'une division par un facteur 1048. Et même si cela est possible, il faut également s'assurer que l'horloge choisit permet d'assurer le nombre de Ticks/seconde défini pour l'OS. C'est donc une possibilité pour faire varier la fréquence de clignotement de la LED mais avec des conséquences non négligeables sur l'application.

2 Ressource FreeRTOS

Le noyau FreeRTOS est un système d'exploitation temps réel (RTOS) ciblé pour les microcontrôleurs et petits processeurs. Il est distribué sous licence libre MIT. Le projet est composé d'un noyau et d'un panel de bibliothèques mises à jour régulièrement. Ces dernières réduisent considérablement le "time to market" en fournissant un panel de programmes destiné aux applications industrielles [1].

2.1 Les tâches

Une application temps réel basée sur RTOS peut être structurée comme un ensemble de tâches indépendantes. Chaque tâche s'exécute dans son propre contexte sans dépendance avec d'autres tâches du système [2]. Une seule tâche de l'application peut être en cours d'exécution à la fois. C'est l'ordonnanceur qui décide quelle tâche sélectionner en fonction de nombreux paramètres comme : leur priorité, leur état, les signaux asynchrones d'entrée, etc. Dans le programme, une tâche est une fonction à exécuter qui contient généralement une boucle infinie. Cette fonction a cependant une particularité, elle ne doit jamais atteindre le retour. Une tâche s'arrête en appelant la procédure `vTaskDelete(NULL)`; alors qu'atteindre la fin de la fonction engendre une exception. Le programme ci-dessous est donné par FreeRTOS [2] a titre d'exemple et illustre la structure de la fonction d'une tâche.

```
1 void vATaskFunction( void *pvParameters ) {
2     for( ; ; )
3     {
4         -- Task application code here. --
5     }
6     /* Tasks must not attempt to return from their implementing
7      function or otherwise exit. */
8
9     vTaskDelete( NULL );
10 }
```

Elle respecte un prototype imposé permettant de récupérer des paramètres lors de la création. La boucle infinie (facultative) vient ensuite. Cet exemple s'attache à mettre en avant la nécessité de supprimer la tâche et de ne jamais sortir de la fonction.

Une fois la fonction d'exécution définie, la tâche doit être créée. Chaque tâche nécessite de la RAM pour stocker son état et des données comme sa pile. Si une tâche est créée à l'aide de `xTaskCreate()`, la RAM requise est automatiquement allouée. Si une tâche est créée en utilisant `xTaskCreateStatic()` alors la RAM est fournie par le développeur de l'application, elle peut donc être allouée statiquement au moment de la compilation. La tâche est ensuite ajoutée à la liste des tâches prêtes à être exécutées. Le programme ci-dessous est donné par FreeRTOS [?] a titre d'exemple et illustre la création d'une tâche.

```
1 void Init_function( void ){
2     BaseType_t xReturned;
3     TaskHandle_t xHandle = NULL;
4
5     /* Create the task, storing the handle. */
6     xReturned = xTaskCreate(
7         vATaskFunction,    /* Function that implements the task. */
8         "NAME",           /* Text name for the task. */
9         STACK_SIZE,        /* Stack size in words, not bytes. */
10        ( void * ) 1,       /* Parameter passed into the task. */
11        tskIDLE_PRIORITY, /* Priority at which the task is created. */
12        &xHandle );        /* Used to pass out the created task's handle. */
13
14     if( xReturned == pdPASS ){
15         /* The task was created. Use the task's handle to delete the task. */
16         vTaskDelete( xHandle );
17     }
}
```

L'intérêt principal de cet exemple est l'utilisation de la fonction `xTaskCreate` et des paramètres associés. On retrouve premièrement un pointeur vers la fonction précédente `vATaskFunction`, c'est l'adresse de

branchement initiale de la tâche. Vient ensuite le nom, sous forme d'une chaîne de caractère, la taille de la pile à allouer, la liste des paramètres à transmettre à la fonction d'exécution, sa priorité et finalement son "identifiant". Ce dernier permettra par la suite de supprimer une tâche donnée par exemple. La condition suivante est parfois négligée par les développeurs, mais permet d'assurer que tout s'est bien déroulé dans la création. Elle est essentielle pour assurer une application robuste. Une fois la tâche créée est complètement indépendante sera appelée par l'ordonnanceur.

2.2 SémaPhores

Un sémaPhore est une variable ou plus généralement un type de données abstrait utilisé pour synchroniser des tâches ou éviter des problèmes de sections critiques. Son utilisation passe par les deux fonctions atomiques `wait()` et `signal()` respectivement remplacées `xSemaphoreTake` et `xSemaphoreGive` dans FreeRTOS. Ainsi, pour franchir un "wait" il faut nécessairement qu'il soit précédé d'un "signal". Le sémaPhore a été inventé par Edsger Dijkstra. Il se modélise très bien par un réseau de pétri ce qui facilite sa preuve formelle. Il existe deux types de sémaPhore binaire et compteur [7].

Dans un sémaPhore binaire l'entier le composant ne peut prendre que deux valeurs 0 ou 1. Son fonctionnement est assimilable à un loquet où 0 signifie occupé et 1 disponible. Il garantit une exclusion mutuelle entre deux fonctions voulant accéder à une ressource critique. Un sémaPhore binaire peut être utilisée pour une **synchronisation**, la tâche endormie aura alors pour seule information si oui ou non des événements ont eu lieu pendant son sommeil. Elle ne saura pas combien. Ce n'est pas un défaut, il faut se poser les bonnes questions en fonction du contexte. Pour créer un sémaPhore binaire, il faut faire appel à la fonction `xSemaphoreCreateBinary`. Son prototype est le suivant :

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

Aucun paramètre n'est nécessaire. Il conviendra simplement de vérifier que la fonction ne retourne pas `NULL`.

Dans l'application temps réelle implémentée par la suite nous utiliserons des sémaPhores pour fixer la période d'exécution des tâches. Il n'est pas nécessaire de vérifier la disponibilité du sémaPhore avant de le prendre lors d'une synchronisation. Le programme ci-dessous donne un exemple d'utilisation pour `xSemaphoregive`.

```
1 if( xSemaphore != NULL ) {
2     /* Give a semaphore and control if an error occurred */
3     if( xSemaphoreGive( xSemaphore ) != pdTRUE ) {
4         /* pdFALSE if an error occurred */
5     }
6 }
```

Un simple appel à la fonction `xSemaphoreGive` suffit à donner le sémaPhore. Les conditions ajoutées contribuent à rendre le programme plus robuste en vérifiant si le sémaPhore a bien été créé, et si aucune erreur n'est apparue.

L'attente du sémaPhore se fait avec la fonction `xSemaphoreTake`. Elle est bloquante et ne peut être franchie que si un jeton (pétri) est disponible. Sans cela, la tâche passera dans l'état d'attente (wait). Le programme ci-dessous illustre la prise d'un sémaPhore dans le cas d'une utilisation pour synchronisation.

```
1 if( xSemaphore != NULL ) {
2     /* Wait forever a semaphore */
3     xSemaphoreTake( xSemaphore, portMAX_DELAY );
4     /* Do something */
5 }
```

La fonction est bloquante les lignes suivant le commentaire *Do something* ne pourront être atteint que si une autre tâche donne un jeton. Le second paramètre permet de spécifier un temps maximum d'attente. Le recours à cette fonctionnalité n'est pas nécessaire dans notre application.

2.3 Files de message

Une file de message est un outil d'ingénierie logiciel communément utilisée pour les communications inter-processus ou inter-processeur. Elle se comporte comme une mémoire FIFO où l'entrée est connecté au processus émetteur et la sortie au récepteur. Pour établir une communication bidirectionnelle entre deux partis il faudra nécessairement deux files. Une file de message peut être vue comme une mémoire tampon entre deux fonctions asynchrone. Dans le cas où la fonction effectuant la lecture s'exécute plus rapidement que celle en lecture la file de message se retrouve généralement vide. L'API FreeRTOS met à disposition des fonctions pour implémenter des files de messages. La première à utiliser est `xQueueCreate` pour la création. Le programme ci-dessous est donné par FreeRTOS [6] à titre d'exemple.

```
1 struct AMessage {
2     char ucMessageID;
3     char ucData[ 20 ];
4 };
5
6 QueueHandle_t xQueue;
7 /* Create a queue capable of containing 10 pointers to AMessage
8 structures. These are to be queued by pointers as they are
9 relatively large structures. */
10 xQueue = xQueueCreate( 10, sizeof( struct AMessage * ) );
11
12 if( xQueue == NULL ) {
13     /* Queue was not created and must not be used. */
14 }
```

Ce programme met en évidence la possibilité d'utiliser des messages taille variable. Dans ce cas, une structure composée d'un identifiant et d'un tableau d'octet est définie. La taille de la file en nombre de messages est fixée à 10. Une fois créée l'instance `xQueue` peut être utilisée. La fonction `xQueueSend` permet de poster un message. Le programme ci-dessous est donné par FreeRTOS [5] à titre d'exemple.

```
1 if( xQueue != 0 ) {
2     /* Send a pointer to a struct AMessage. Don't block if already full. */
3     pxMessage = &xMessage;
4     xQueueSend( xQueue, ( void * ) &pxMessage, ( TickType_t ) 0 );
5 }
```

On considère un message `xMessage` affecté au préalable. Son adresse est passée en paramètre. Le contenu du message est copié à l'appel de `xQueueSend`, la variable n'a pas besoin d'être statiquement défini. Le dernier paramètre offre la possibilité de stipuler un temps maximum d'attente dans le cas où la file est pleine. Dans notre exemple ce cas n'est pas traité, il est toutefois intéressant de savoir que cela est possible. La fonction `xQueueReceive` permet ensuite de récupérer les messages depuis une autre tâche. Le programme ci-dessous est donné par FreeRTOS [4] à titre d'exemple.

```
1 void vADifferentTask( void *pvParameters ) {
2     struct AMessage xRxedStructure;
3
4     if( xStructQueue != NULL ) {
5         /* Receive a message from the created queue to hold complex struct AMessage
6          structure. The value is read into a struct AMessage variable, so after calling
7          xQueueReceive() xRxedStructure will hold a copy of xMessage. */
8         if( xQueueReceive( xStructQueue, &( xRxedStructure ),
9                           ( TickType_t ) portMAX_DELAY ) == pdPASS ) {
10             /* xRxedStructure now contains a copy of xMessage. */
11         }
12     }
13 }
```

Après l'exécution de la fonction, la variable `xRxedStructure` contient le message le plus ancien de la file. Si la file est vide la fonction est bloquée. Il est possible de borner l'attente en spécifiant une valeur en troisième paramètre. Dans cet exemple le temps d'attente est maximum.

3 Développement de l'application

L'application finale demandée reprend les éléments de configuration vus en section 2. Cette section comporte un récapitulatif des éléments de l'application avec éventuellement des portions de code montrant leur utilisation. Un exemple de résultats obtenus après lancement de l'application sont montrés.

3.1 Cahier des charges

Il s'agit d'implanter une application simple permettant de mesurer le temps de réaction d'une personne (Utilisateur) et mettant en œuvre différentes entrées/sorties disponibles sur la carte SAMD21 en utilisant SW0 et LED0 et sur la carte OLED1 (extension de la carte SAMD21) en utilisant BP1, BP2, BP3, LED1, LED2, LED3 et l'écran OLED. La structure fonctionnelle de l'application à développer peut être présentée dans un premier temps (légèrement modifiée par la suite) par la figure suivante :

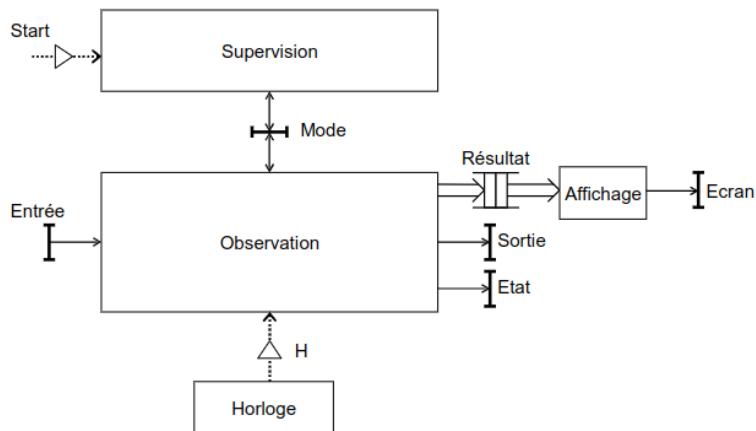


Figure 1: Structure fonctionnelle initiale de l'application à développer

La mesure de vitesse de réaction est démarrée lors de l'apparition de l'événement Start. Elle consiste alors à émettre un code aléatoire en allumant un ensemble de LED (variable Sortie matérialisée par LED1, LED2 et LED3) et à mesurer le temps nécessaire à l'utilisateur pour reproduire la combinaison sur des boutons poussoirs (variable Entrée, matérialisée par BP1, BP2 et BP3) en comptant le nombre d'événements H entre la production d'une valeur sur Sortie et la reproduction de cette valeur sur Entrée. Plusieurs mesures de vitesse de réaction peuvent être effectuées afin d'obtenir des valeurs minimale, moyenne et maximale (constante NbMesToDo dans l'automate présenté par la suite, constante dont la valeur est strictement supérieure à 1, peut avoir la valeur 5 par exemple). Lorsque toutes les mesures ont été effectuées, des statistiques sont transmises à la fonction Affichage qui se charge de les afficher sur l'Ecran de la carte OLED1. La précision de la mesure du temps de réaction doit être de 1 ms. L'état de la fonction d'observation (Observe ou Non_Observe) est en permanence affiché (variable Etat, matérialisée par LED0 de la carte SAMD21XPLAINEDPRO, la LED0 doit être allumée lorsqu'Etat a la valeur Observe). Le comportement de la fonction Observation est défini par l'automate de la figure 3.

3.2 Éléments de l'application

L'application se base sur la structure fonctionnelle suivante :

Par rapport à la structure fonctionnelle présentée en figure 1, *Start* n'est plus un évènement mais une variable partagée. Cette solution est plus simple à implanter car elle ne nécessite pas la gestion des interruptions engendrée par l'attente d'un évènement en entrée du système. De ce schéma, différentes informations peuvent être extraites. Quatre tâches distinctes peuvent être imaginées pour cette application temps-réel :

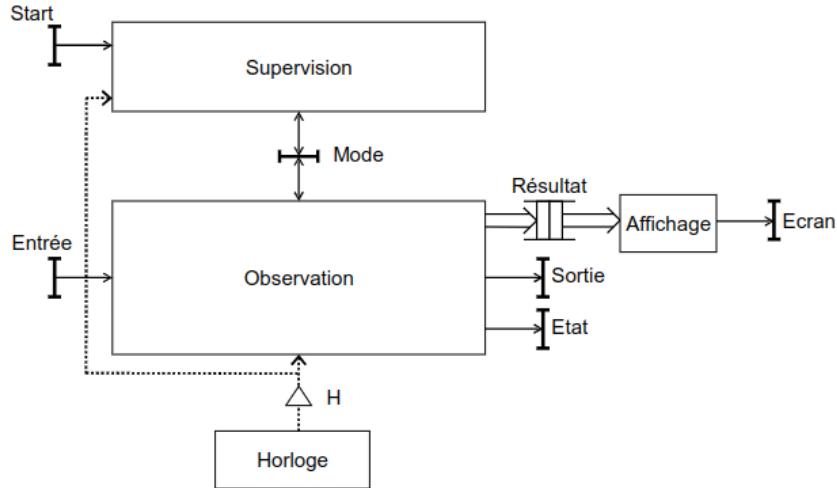


Figure 2: Structure fonctionnelle de l'application à développer

- Horloge
- Supervision
- Observation
- Affichage

Nous pouvons faire l'hypothèse que deux sémaphores, matérialisées dans la structure par les relations de Start et H, peuvent être implantées au sein de l'application. L'application contient une variable partagée *Mode* et une file de message *Résultats*. Les différentes entrées/sorties présentées dans cette structure fonctionnelle correspondent à des éléments physiques présents sur la carte tels que des boutons poussoirs, des LEDs ou l'écran.

3.3 Résultat

L'application peut être testée en programmant la carte. L'utilisateur doit appuyer sur le bouton SW0 pour lancer une série de mesures du temps de réaction. Dans le cadre des résultats montrés ci-dessous, cinq mesures successives sont effectuées avant l'affichage des résultats sur l'écran. Les figures suivantes montrent l'affichage de l'écran au lancement de l'application (figure 3) et à la fin d'une série de mesure (figure 4). Les résultats montrés sont, dans l'ordre:

- Le temps de réaction minimal
- Le temps de réaction moyen
- Le temps de réaction maximal
- La période d'observation entre deux appels de la tâche Observation

L'unité de chacune de ces valeurs est le *tick d'horloge*. Le projet utilise une horloge à la fréquence de 8 MHz, divisée par un facteur AAA. Ainsi, un *tick d'horloge* correspond à approximativement BBB secondes.

Au lancement de l'application, l'écran affiche du texte en préparation de l'arrivée des résultats. C'est lors de l'initialisation du matériel que cet affichage a lieu.

Après une série de mesures, la file de message contenant les résultats est envoyée de la tâche *Observation* à la tâche *Affichage*. Cette file de message contient la variable *Résultat*, instance de la structure

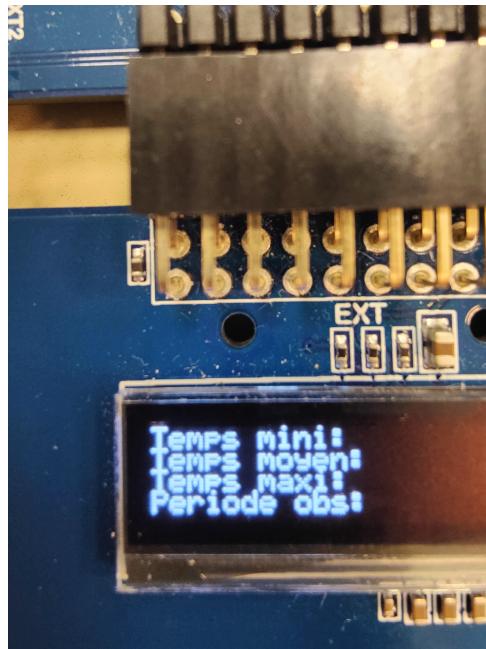


Figure 3: Affichage de l'écran au lancement de l'application

Resultat.t contenant les différentes mesures. Après réception de la file de message, la tâche *Affichage* fait apparaître sur l'écran les résultats de la mesure.

Pour prouver le bon fonctionnement de l'application, une vidéo est proposée dans le même dossier que ce rapport. Cette vidéo couvre le démarrage de l'application après un reset. Une série de mesures a lieu, menant à l'affichage des résultats. Enfin, l'utilisateur relance une série de mesures après appui sur le bouton SW0.

Outre l'aspet fonctionnel de l'application, il est nécessaire de prouver son bon fonctionnement d'un point de vue temps-réel. La mesure de la période d'obsevation permet de justifier que les tâches s'exécutent dans une fenêtre de temps qui respecte les contraintes temporelles. La tâche "Observation" est réveillée par une sémaphore en provenance de la tâche "Clock". Selon la configuration du projet, l'horloge définie par l'OS pour les applications est de 1 khHz. La tâche "Clock" envoie une sémaphore d'éveil après une attente de 1 Tick d'horloge. La tâche "Observation" devrait donc être réveillée environ tout les 1000 Ticks d'horloge. Lors des tests effectués, c'est ce qui a pu être observé, avec une valeur de période d'observation toujours comprise entre 990 et 994. Dans le cas des résultats présentés en figure 4, la période d'observation est de 991 Ticks d'horloge.



Figure 4: Affichage de l'écran après une session de mesure du temps de réaction

Conclusion

Pour conclure, les séances de Travaux Pratiques du module Systèmes Temps-Réel nous ont amené à obtenir une application fonctionnelle.

Les différentes étapes réalisées au préalable du développement de l'application nous ont permis de comprendre les différents éléments à manipuler. Il était intéressant d'étudier les étapes de configuration de l'exécutif FreeRTOS, puis la configuration de la carte SAMD21.

Enfin, ce mini-projet de développement d'une application Temps-Réel a été l'opportunité d'aller plus loin dans l'utilisation du logiciel Microchip Studio par rapport aux autres projets réalisés en quatrième ou conquième année.

References

- [1] FreeRTOS web site. About freertos kernel : The freertos™ kernel, January 2023.
- [2] FreeRTOS web site. Developer docs : Tasks and co-routines, January 2023.
- [3] FreeRTOS web site. Memory management, January 2023.
- [4] FreeRTOS web site. Queues : xqueuereceive, January 2023.
- [5] FreeRTOS web site. Queues : xqueuesend, January 2023.
- [6] FreeRTOS web site. Task creation : xtaskcreate(), January 2023.
- [7] Wikipédia. Sémaphore (informatique). *Wikipedia.org*, September 2022.

Appendix

main.c

```
1  /*-----*
2   *-----* Inclusion des bibliothèques -----*/
3  *-----*/
4  #include <asf.h>
5  #include <stdio.h>
6  #include <string.h>
7  #include "FreeRTOS.h"
8  #include "task.h"
9  #include "semphr.h"
10 #include "timers.h"
11 #include "MesTemps.h"
12 #include "oled1.h"
13
14 /* Définition de prototypes généraux */
15
16 void vSetupHardware(void);
17 int8_t get_Entree(void);
18 void set_Sortie(int8_t);
19
20 /*-----*/
21 //! Extension header for the OLED1 Xplained Pro
22 #define OLED1_EXT_HEADER EXT1 // définition de la connection de la carte OLED1 sur l'extension 3 de la carte
23
24 static OLED1_CREATE_INSTANCE(oled1, OLED1_EXT_HEADER);
25
26 /*-----*/
27
28 /*-----*
29  *-----* Déclaration des priorités des tâches -----*/
30 *-----*/
31
32 #define CLOCK_TASK_PRIORITY ( tskIDLE_PRIORITY +5 )
33 #define OBSERVATION_TASK_PRIORITY ( tskIDLE_PRIORITY +2 )
34 #define SUPERVISION_TASK_PRIORITY ( tskIDLE_PRIORITY +3 )
35 #define AFFICHAGE_TASK_PRIORITY ( tskIDLE_PRIORITY +1 )
36
37 // Définition de la structure Résultat
38 typedef struct {
39     uint16_t periodeObservation;
40     uint16_t MesMin;
41     uint16_t Moyenne;
42     uint16_t MesMax;
43 } Resultat_t;
44
45 /*-----*
46  *-----* Déclaration des sémaphores -----*/
47 *-----*/
48
49 xSemaphoreHandle xSem_H1=NULL; // Pour réveiller LEDControl
50 xSemaphoreHandle xSem_H2=NULL; // Pour réveiller Supervision
51
52 /*-----*
53  *-----* Déclaration de files de message -----*/
54 *-----*/
55
56 xQueueHandle ResultQueue;
57
58 /*-----*
59  *-----* Déclaration de constantes et de variables globales -----*/
60 *-----*/
61
62 #define HORLOGE_TASK_DELAY 1
```

```

63 enum mode_t {OBSERVATION_MODE, NORMAL_MODE};
64 enum mode_t g_mode = NORMAL_MODE;
65
66 /*-----*
67     Déclaration des tâches
68 -----*/
69 //exemple de déclaration de fonction de tache: void MaFonction( void * pvParameters );
70
71 void vCodeClock( void * pvParameters );
72 void vCodeObservation( void * pvParameters );
73 void vCodeSupervision( void * pvParameters );
74 void vCodeAffichage( void * pvParameters );
75
76 /***** Fonctions primitives freeRTOS hook/callback *****/
77
78 /***** Déclaration de la fonction d'initialisation matérielle *****/
79
80 /***** Initialisation matérielle *****/
81
82 void vSetupHardware(void)
83 {
84     system_init();          // Initialisation matérielle
85     gfx_mono_init();        // Initialisation écran
86     oled1_init(&oled1);    // initialisation de la carte OLED1
87
88     // Reset total de l'écran
89     gfx_mono_draw_filled_rect(0, 0, GFX_MONO_LCD_WIDTH, GFX_MONO_LCD_HEIGHT, GFX_PIXEL_CLR);
90
91     // Affichage initiale de l'écran au lancement de l'application
92     gfx_mono_draw_string("Temps mini:", 0, 0, &sysfont);
93     gfx_mono_draw_string("Temps moyen:", 0, 7, &sysfont);
94     gfx_mono_draw_string("Temps maxi:", 0, 14, &sysfont);
95     gfx_mono_draw_string("Periode obs:", 0, 21, &sysfont);
96
97 }
98
99 /***** Entrées/sorties *****/
100
101 void get_Entree(void) {
102     bool BP1, BP2, BP3;
103     BP1 = oled1_get_button_state(&oled1, OLED1_BUTTON1_ID);
104     BP2 = oled1_get_button_state(&oled1, OLED1_BUTTON2_ID);
105     BP3 = oled1_get_button_state(&oled1, OLED1_BUTTON3_ID);
106
107     // Entrée est comprise entre 0b000 et 0b111
108     return (BP1 | (BP2 << 1) | (BP3 << 2));
109 }
110
111 void set_Sortie(int8_t Sortie) {
112
113     // Sortie est comprise entre 0b000 et 0b111
114     oled1_set_led_state(&oled1, OLED1_LED1_ID, Sortie & 0x1);
115     oled1_set_led_state(&oled1, OLED1_LED2_ID, Sortie & (1 << 1));
116     oled1_set_led_state(&oled1, OLED1_LED3_ID, Sortie & (1 << 2));
117 }
118
119 /***** Programme principal *****/
120
121
122 }
123
124
125
126
127
128

```

```

129 int main (void)
130 {
131     //Initialisation du matériel
132     vSetupHardware();
133     ConfigureMeasure();
134
135     ResultQueue = xQueueCreate( 3, sizeof( Resultat_t ) );
136
137     if( ResultQueue == NULL )
138     {
139         /* Queue was not created and must not be used. */
140         while(1);
141     }
142
143     //Création des sémaphores
144
145     xSem_H1 = xSemaphoreCreateBinary();
146     xSem_H2 = xSemaphoreCreateBinary();
147
148     //Création des tâches
149
150     xTaskCreate( vCodeClock, ( const char * ) "Clock",
151 configMINIMAL_STACK_SIZE, NULL, CLOCK_TASK_PRIORITY, NULL );
152     xTaskCreate( vCodeObservation, ( const char * ) "Observation",
153 configMINIMAL_STACK_SIZE, NULL, OBSERVATION_TASK_PRIORITY, NULL );
154     xTaskCreate( vCodeSupervision, ( const char * ) "Supervision",
155 configMINIMAL_STACK_SIZE, NULL, SUPERVISION_TASK_PRIORITY, NULL );
156     xTaskCreate( vCodeAffichage, ( const char * ) "Affichage",
157 configMINIMAL_STACK_SIZE, NULL, AFFICHAGE_TASK_PRIORITY, NULL );
158
159     //Lancement de l'ordonnateur
160     vTaskStartScheduler();
161 }
162
163
164 /*****
165     Déclaration du code des tâches
166 *****/
167
168 /*****
169     tache : Clock
170 *****/
171 void vCodeClock(void * pvParameters)
172 {
173     for( ; ; )
174     {
175         vTaskDelay(HORLOGE_TASK_DELAY);
176         xSemaphoreGive( xSem_H1 );
177         xSemaphoreGive( xSem_H2 );
178     }
179 }
180
181 /*****
182     code tache : Supervision
183 *****/
184
185 void vCodeSupervision (void * pvParameters){
186     bool start_bp_state;
187     static bool prev_start_bp_state;
188
189     for(;;){
190
191         xSemaphoreTake(xSem_H2, portMAX_DELAY);
192
193         // Récupération de la valeur (true ou false) du bouton poussoir 0
194         start_bp_state = port_pin_get_input_level(BUTTON_0_PIN);

```

```

195     // Les boutons sont en logique négative (bouton appuyé -> false)
196     // Pour démarrer l'application, il faut appuyer sur le bouton poussoir 0
197     if ((prev_start_bp_state == true) && (start_bp_state == false) && (g_mode == NORMAL_MODE)){
198         g_mode = OBSERVATION_MODE;
199     }
200
201     prev_start_bp_state = start_bp_state;
202 }
203
204 /**
205  * code tache : Affichage
206 *****/
207
208 void vCodeAffichage (void * pvParameters){
209     Resultat_t resultat;
210     char str_val[10];
211
212     for(;;){
213
214         if( xQueueReceive( ResultQueue ,&( resultat ),( TickType_t ) portMAX_DELAY ) == pdPASS )
215         {
216             // Reset partie droite de l'écran
217             gfx_mono_draw_filled_rect(75, 0, GFX_MONO_LCD_WIDTH, GFX_MONO_LCD_HEIGHT, GFX_PIXEL_CLR);
218
219             // Affichage des résultats sur l'écran
220             itoa(resultat.MesMin, str_val, 10);
221             gfx_mono_draw_string(str_val, 75, 0, &sysfont);
222             itoa(resultat.Moyenne, str_val, 10);
223             gfx_mono_draw_string(str_val, 75, 7, &sysfont);
224             itoa(resultat.MesMax, str_val, 10);
225             gfx_mono_draw_string(str_val, 75, 14, &sysfont);
226             itoa(resultat.periodeObservation, str_val, 10);
227             gfx_mono_draw_string(str_val, 75, 21, &sysfont);
228
229         }
230     }
231 }
232
233 /**
234  * code tache : Observation
235 *****/
236
237
238 void vCodeObservation(void * pvParameters) {
239
240     /* private type */
241     enum Etat_t {NON_OBSERVATION, OBSERVE};
242     enum fsm_state_t {REPOS, ATTENTE, MESURE, ATTENTE_MESURE, FIN_MESURE};
243
244     /* private variable */
245     enum Etat_t Etat = NON_OBSERVATION;
246     enum fsm_state_t fsm_state = REPOS;
247     Resultat_t resultat;
248     uint16_t T;
249     uint8_t Sortie = 0;
250     uint8_t Entree = 0;
251     uint8_t NbMes = 0;
252     uint16_t Somme;
253     uint8_t NbMesToDo = 5;
254     uint16_t Duree_Att = 0;
255
256     for (;;) /* infinite loop */
257     {
258         xSemaphoreTake(xSem_H1, portMAX_DELAY);
259
260         // Récupération de l'entrée (appui sur les boutons par l'utilisateur)

```

```

261 Entrée = get_Entree();
262
263 // Production d'une valeur sur sortie à recopier par l'utilisateur
264 set_Sortie(Sortie);
265
266 // La période d'observation est mesurée à chaque appel de la tâche
267 resultat.periodeObservation = EndMeasure();
268 StartMeasure();
269
270 switch (fsm_state){
271     case REPOS:
272         Etat = NON_OBSERVATION;
273         Sortie = 0;
274
275         if (g_mode == OBSERVATION_MODE) {
276             T = 0;
277             NbMes = 0;
278             Somme = 0;
279             Duree_Att = rand () % 1001 + 2000;
280             fsm_state = ATTENTE;
281         }
282         break;
283     case ATTENTE:
284         Etat = OBSERVE;
285         Sortie = 0;
286
287         if (T < Duree_Att) {
288             T = T + 1;
289         } else {
290             T = 0;
291             Sortie = rand() % 7 + 1;
292             fsm_state = MESURE;
293         }
294         break;
295     case MESURE:
296         Etat = OBSERVE;
297
298         if (Entree != Sortie) {
299             T = T + 1;
300         } else if ((Entree == Sortie) && (NbMes == 0)) {
301
302             // Première mesure
303             resultat.MesMin = T;
304             resultat.MesMax = T;
305             Somme = T;
306
307             NbMes = NbMes + 1;
308             fsm_state = ATTENTE_MESURE;
309         } else if ((Entree == Sortie) && (NbMes != 0) && (NbMes != NbMesToDo - 1)) {
310             if (T < resultat.MesMin) resultat.MesMin = T;
311             if (T > resultat.MesMax) resultat.MesMax = T;
312             Somme = Somme + T;
313
314             NbMes = NbMes + 1;
315             fsm_state = ATTENTE_MESURE;
316         } else if ((Entree == Sortie) && (NbMes == NbMesToDo - 1)) {
317
318             // Dernière mesure
319             if (T < resultat.MesMin) resultat.MesMin = T;
320             if (T > resultat.MesMax) resultat.MesMax = T;
321             Somme = Somme + T;
322
323             NbMes = NbMes + 1;
324             resultat.Moyenne = Somme/NbMesToDo;
325
326             // Envoie de la file de message à la tâche Affichage

```

```

327             if( xQueueSend( ResultQueue,( void * ) &resultat, ( TickType_t ) 0 ) != pdPASS )
328             {/* Failed to post the message */}
329             fsm_state = FIN_MESURE;
330         }
331         break;
332     case ATTENTE_MESURE:
333         Etab = OBSERVE;
334         Sortie = 0;
335
336         if (Entree == 0) {
337             T = 0;
338             Duree_Att = rand () % 1001 + 2000;
339             fsm_state = ATTENTE;
340         }
341         break;
342     case FIN_MESURE:
343         Sortie = 0;
344         Etab = NON_OBSERVATION;
345
346         if (Entree == 0) {
347             g_mode = NORMAL_MODE;
348             fsm_state = REPOS;
349         }
350         break;
351     default:
352         fsm_state = REPOS;
353         break;
354     }
355
356     // Allumage de la LED0 en fonction de l'etat
357     if (Etab == OBSERVE) {
358         port_pin_set_output_level(LED_0_PIN, LED_0_ACTIVE);
359     } else {
360         port_pin_set_output_level(LED_0_PIN, !LED_0_ACTIVE);
361     }
362 }
363 }
```

FreeRTOSConfig.h

```

1  /*
2   * FreeRTOS V202012.00
3   * Copyright (C) 2020 Amazon.com, Inc. or its affiliates. All Rights Reserved.
4   *
5   * Permission is hereby granted, free of charge, to any person obtaining a copy of
6   * this software and associated documentation files (the "Software"), to deal in
7   * the Software without restriction, including without limitation the rights to
8   * use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of
9   * the Software, and to permit persons to whom the Software is furnished to do so,
10  * subject to the following conditions:
11  *
12  * The above copyright notice and this permission notice shall be included in all
13  * copies or substantial portions of the Software.
14  *
15  * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
16  * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS
17  * FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR
18  * COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER
19  * IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN
20  * CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
21  *
22  * http://www.FreeRTOS.org
23  * http://aws.amazon.com/freertos
24  *
```

```

25 * 1 tab == 4 spaces!
26 */
27
28
29 #ifndef FREERTOS_CONFIG_H
30 #define FREERTOS_CONFIG_H
31
32 /*-----
33 * Application specific definitions.
34 *
35 * These definitions should be adjusted for your particular hardware and
36 * application requirements.
37 *
38 * THESE PARAMETERS ARE DESCRIBED WITHIN THE 'CONFIGURATION' SECTION OF THE
39 * FreeRTOS API DOCUMENTATION AVAILABLE ON THE FreeRTOS.org WEB SITE.
40 *
41 * See http://www.freertos.org/a00110.html
42 -----*/
43
44 #include <asf.h>
45
46 #define configUSE_PREEMPTION 1
47 #define configUSE_IDLE_HOOK 0
48 #define configUSE_TICK_HOOK 0
49 #define configCPU_CLOCK_HZ (system_clock_source_get_hz( SYSTEM_CLOCK_SOURCE_OSC8M ))
50 #define configTICK_RATE_HZ ( ( TickType_t ) 1000 )
51 #define configMAX_PRIORITIES ( 7 )
52 #define configMINIMAL_STACK_SIZE ( ( unsigned short ) 100 )
53 #define configTOTAL_HEAP_SIZE ( ( size_t ) ( 16000 ) )
54 #define configMAX_TASK_NAME_LEN ( 15 )
55 #define configUSE_TRACE_FACILITY 0
56 #define configUSE_16_BIT TICKS 0
57 #define configIDLE_SHOULD_YIELD 1
58 #define configUSE_MUTEXES 1
59 #define configQUEUE_REGISTRY_SIZE 8
60 #define configCHECK_FOR_STACK_OVERFLOW 0
61 #define configUSE_RECURSIVE_MUTEXES 1
62 #define configUSE_MALLOC_FAILED_HOOK 0
63 #define configUSE_APPLICATION_TASK_TAG 0
64 #define configUSE_COUNTING_SEMAPHORES 1
65 #define configUSE_QUEUE_SETS 1
66
67 // #define configSUPPORT_STATIC_ALLOCATION 1
68
69 /* Run time stats related definitions. */
70 void vMainConfigureTimerForRunTimeStats( void );
71 unsigned long ulMainGetRunTimeCounterValue( void );
72 #define configGENERATE_RUN_TIME_STATS 0
73 // #define portCONFIGURE_TIMER_FOR_RUN_TIME_STATS() vMainConfigureTimerForRunTimeStats()
74 #define portGET_RUN_TIME_COUNTER_VALUE() ulMainGetRunTimeCounterValue()
75
76 /* Co-routine definitions. */
77 #define configUSE_CO_ROUTINES 0
78 #define configMAX_CO_ROUTINE_PRIORITIES ( 2 )
79
80 /* Software timer definitions. */
81 #define configUSE_TIMERS 1
82 #define configTIMER_TASK_PRIORITY ( 2 )
83 #define configTIMER_QUEUE_LENGTH 5
84 #define configTIMER_TASK_STACK_DEPTH ( 80 )
85
86 /* Set the following definitions to 1 to include the API function, or zero
87 to exclude the API function. */
88 #define INCLUDE_vTaskPrioritySet 1
89 #define INCLUDE_uxTaskPriorityGet 1
90 #define INCLUDE_vTaskDelete 1

```

```

91 #define INCLUDE_vTaskCleanUpResources    1
92 #define INCLUDE_vTaskSuspend           1
93 #define INCLUDE_vTaskDelayUntil       1
94 #define INCLUDE_vTaskDelay            1
95 #define INCLUDE_eTaskGetState         1
96
97 /* This demo makes use of one or more example stats formatting functions. These
98 format the raw data provided by the uxTaskGetSystemState() function in to human
99 readable ASCII form. See the notes in the implementation of vTaskList() within
100 FreeRTOS/Source/tasks.c for limitations. */
101 #define configUSE_STATS_FORMATTING_FUNCTIONS    0
102
103 /* Normal assert() semantics without relying on the provision of an assert.h
104 header file. */
105 #define configASSERT( x ) if( ( x ) == 0 ) { taskDISABLE_INTERRUPTS(); for( ; ; ); }
106
107 /* Definitions that map the FreeRTOS port interrupt handlers to their CMSIS
108 standard names - or at least those used in the unmodified vector table. */
109 #define xPortPendSVHandler PendSV_Handler
110 #define xPortSysTickHandler SysTick_Handler
111
112 /* The size of the global output buffer that is available for use when there
113 are multiple command interpreters running at once (for example, one on a UART
114 and one on TCP/IP). This is done to prevent an output buffer being defined by
115 each implementation - which would waste RAM. In this case, there is only one
116 command interpreter running. */
117 #define configCOMMAND_INT_MAX_OUTPUT_SIZE 2048
118
119 #endif /* FREERTOS_CONFIG_H */

```