**TP2 - Freertos**

**Jouer avec le Shell**



**21/11/2022**

**DANDACHE Ali PHILIPPE Quentin**

# Présentation

On utilise la carte STM32F746G Discovery :

A close-up of a circuit board

Description automatically generated with medium confidence

On récupère le projet qu’on a commencé en TD de shell. Voici le lien GitHub : <https://github.com/lfiack/rtos_td_shell.git>

On pense à bien vérifier que le bout de code pour le printf est bien présent :

**int** **\_\_io\_putchar**(**int** ch)

{

HAL\_UART\_Transmit(&huart1, (uint8\_t\*)&ch, 1, HAL\_MAX\_DELAY);

**return** ch;

}

# Terminer le shell du TD

## Compilation du projet présent sur GiHub

On reprend ce qu’on a fait en TD. On peut rajouter des fonctions à la listes des fonctions du shell. On y avait rajouté la fonction add. On utilise la fonction shell\_add afin de créer des nouvelles fonctionnalités dans le shell. Il faut faire attention au BufferSize puisque le UART en a besoin pour la communication. S’il est trop petit, les lignes seront tronquées. De plus, il faut utiliser la fonction atoi afin de s’assurer que les caractères ascii sont bien converti à des integer. Ajouter la librairie pour sous les balises /\* USER CODE BEGIN Includes \*/.

**#include** <stdlib.h>

Nous avons réglé **#define** BUFFER\_SIZE 256.

## Faire apparaitre les arguments

argc donne le nombre d'éléments de la ligne de commande, et argv contient ces éléments sous la forme d'un tableau de chaînes de caractères. argv[0] contient le nom de la commande, argv[i], pour i allant de 1 à argc-1, le ième argument.

## Mécanisme d’exécution

On a la fonction shell\_init qui initialise le shell avec la liste des fonctions. Ensuite cette fonction envoie les caractères de début du shell pour avoir un visuel du lancement.



Le shell\_init rajoute la fonction help à la liste des fonctions du shell avec la fonction shell\_add.

La fonction shell\_add rajoute à la liste de fonctions du shell des fonctions en précisant le caractère d’appel de la fonction (ex : h pour help), la fonction et sa description.

Ensuite, shell\_run lance le shell et commence ainsi la boucle d’écoute des caractères entrés. Lorsque le Carriage return est lancé (la touche entrée du clavier, le caractèere \r), le shell execute la commande qu’il a écouté et gardé dans un buffer à l’aide de la fonction shell\_exec.

Shell\_exec comme sont nom l’indique, exécute les commandes transmises.

## Problème du shell

Le problème avec tout ce qui a été expliqué juste avant est qu’on reste bloqué dans le shell, à écouter. Ainsi, les tache moins prioritaire que le shell ne s’exécuterons point.

## Solution

Comme vu en cours, la solution à ce problème est d’utiliser un sémaphore binaire et gérer cela à l’aide de xSemaphoreGiveFromISR et HAL\_UART\_Receive\_IT. L’idée est que lorsqu’on écrit dans la console du shell, une interruption de l’UART arrive, on libère le sémaphore pour que l’information soit traitée.

Graphical user interface, text, application

Description automatically generated  
*Cours rtos\_chapitre3\_slide12*

La fonction portYIELD\_FROM\_ISR appelle manuellement le scheduler.

# Respect des priorités

Graphical user interface, text, application, email

Description automatically generated

Si on ne respecte pas les priorités décrites dans la partie présentation, la communication via le printf, qui est relié sur le USART1, risque d’être interrompu.

Nous fixons donc LIBRARY\_MAX\_SYSCALL\_INTERRUPT\_PRIORITY à 7.  


La valeur de Preemption du USART1 reste identique à celle de la LIBRARY\_MAX\_SYSCALL\_INTERRUPT\_PRIORITY malgré le changement.

Graphical user interface, application, Teams

Description automatically generated

Pour dissocier les valeurs, il faut désactiver le bouton « Uses FreeRTOS functions » depuis le NVIC. Ensuite on arrive à éditer la valeur du niveau de preemption de l’usart1 . En pratique, ceci n’a rien changé..

# Fonction led() appelable depuis le shell

Pour allumer une LED, on a besoin de configurer la brocher PI1 de la carte.

Code lié au shell :

**int** **setLedRate**( **int** argc, **char** \*\* argv)

{

**if**(argc!=2)

{

**int** size = **snprintf** (print\_buffer, BUFFER\_SIZE, "Error y a pas le bon nombre d'arguments pour la fonction setLedRate\r\n");

uart\_write(print\_buffer, size);

}

**else**

{

LED\_DELAY = **atoi**(argv[1]);

vTaskResume(xHandleLED); //reveille la tache si elle est suspended else pass

}

**printf**("%d\n\r", LED\_DELAY);

**return** 0;

}

**Code de la tache :**

**int** **led**()

{

**int** LED\_Delay\_local\_var;

**while**(1)

{

LED\_Delay\_local\_var = LED\_DELAY;

**switch**(LED\_Delay\_local\_var)

{

**case** 0 :

HAL\_GPIO\_WritePin(LED\_GPIO\_Port, LED\_Pin,*RESET*);

vTaskSuspend(xHandleLED); // communiquer PROPREMENT cad suspendre si l 0 == led off

**break**;

**default** :

HAL\_GPIO\_TogglePin(LED\_GPIO\_Port, LED\_Pin);

vTaskDelay(LED\_Delay\_local\_var);

**break**;

}

vTaskDelay(LED\_Delay\_local\_var);

}

**return** 0;

}

Nous avons commencé par vérifier le bon nombre de caractèeres. Si ce n’est pas le cas, on renvoie une erreur. Suite, si c’est le bon, on traite la commande. Dans le cas où l’utilisateur veut mettre la LED à 0, on endort la tache (vTaskSuspend(xHandleLED);) jusqu’à nouvel ordre, sinon, on fait clignoter la LED avec la valeur demandée.

# Fonction spam

Prenons un exemple de cette fonction : spam Message 3

Message

Message

Message

**int** **spammer**( **int** argc, **char** \*\* argv)

{

**if**(argc!=3)

{

**int** size = **snprintf** (print\_buffer, BUFFER\_SIZE, "Error y a pas le bon nombre d'arguments pour la fonction Spam \r\n");

uart\_write(print\_buffer, size);

**return** -1;

}

**else**

{

**int** size = **snprintf** (print\_buffer, BUFFER\_SIZE, "%s\r\n", argv[1]);

**for** (**int** i=0; i<**atoi**(argv[2]);i++) uart\_write(print\_buffer, size);

}

**return** 0;

}

On rajoute la fonction spam au shell avec

shell\_add('s', spammer, "spam un mot le nombre que vous donnez en argument");

**On fait le test :**

A picture containing text

Description automatically generated

**TP3 - Freertos**

**Debug, gestion d’erreur et statistiques**

Logo

Description automatically generated

**21/11/2022**

**DANDACHE Ali PHILIPPE Quentin**

# Gestion du tas

## Zone réservée à l’allocation dynamique

En anglais, on appelle cette zone Heap. En francais on appelle ça le **tas**. Tout ce qui est en lien avec de l’allocation dynamique de mémoire se trouve dans le tas.

Dans CubeIDE, on peut voir le type de Heap qu’on veut utiliser :

Table

Description automatically generated

## Gestion du tas

La gestion du tas se fait par FreeRTOS .

FreeRTOS offers several heap management schemes that range in complexity and features. It is also possible to provide your own heap implementation, and even to use two heap implementations simultaneously.

## Error Handler

On ajoute la fonction gestion d’erreur sur toutes les taches qui sont susceptible de générer des erreurs c’est-à-dire toute les taches finissant par Create.

On rajoute alors l’Error\_Handler pour la creation de nos taches et semaphores.

semaphori = xSemaphoreCreateBinary();

semaphore\_led = xSemaphoreCreateBinary();

**if** (semaphori == NULL) Error\_Handler();

**if** (semaphore\_led == NULL) Error\_Handler();

**if**(xReturned == pdTRUE)

{

**printf**("LED task created \r\n");

}

**else**

{

**printf**("Error LED task not created \r\n");

Error\_Handler();

}

## RAM et Flash

Pour afficher la fenetre demandée, on va dans CubeIDE et on clique sur le bouton Build Analyser (prêt du bouton de la console).

Table

Description automatically generated

## Création de taches bidons

On va chercher à saturer la mémoire en créant des taches bidons.

A picture containing table

Description automatically generated

Notons que la taille du tas fait 15360 Bytes et qu’on avait set la taille de chaque pile (Stack) de chaque tache à 1024 via la variable **#define** STACK\_SIZE 1024.

Notons pour simplifier cette procédure, on peut fixerun stack size plus grand pour nos fonction bidons **#define** STACK\_SIZE\_BIDON 4096.

Text

Description automatically generated

## Nouvelle utilisation de la mémoire

Table

Description automatically generated

On remarque que cette valeur n’a pas changé. C’est normal puisqu’on a saturé le tas et non pas toute la mémoire. Le tas étant fixé à 15360 Bytes. On retrouve cette valeur en regardant la RAM utilisé. 19 KB étant nos 15KB de Heap plus des KB pour le code, variables globales …

## Augmenter la taille du tas

On augmente la taille du tas. Dans CubeIDE, on va dans l’IOC, middleware, FreeRTOS, Config Parameters, on scroll down jusqu’à voir la partie Memory Management. On modifie la valeur :

Graphical user interface, application, Word

Description automatically generated

On reconnais nos 320 KB de mémoire RAM.

Table

Description automatically generated

On compile et on teste :

Text

Description automatically generated

Text

Description automatically generated

Nos taches ont bien été créées et fonctionnelles. Ici on a repris le code donné en exemple dans le cours. On voit bien que les taches bidons 1 2 et 3 sont dans leur état Running :

Diagram

Description automatically generated

## Memory use

Comme on a augmenté la valeur de la taille du tas à 65536 Bytes, on a l’usage de la mémoire RAM qui a augmenté aussi :

Table

Description automatically generated

On voit bien que nos 65KB de Heap sont bien représenté dans les 68KB de RAM utilisée.

Par la suite, on active et désactive la creation de fonction bidons via le BIDON\_FLAG.

/\* Private variables ---------------------------------------------------------\*/

/\* USER CODE BEGIN PV \*/

**int** LED\_DELAY = 500;

**int** BIDON\_FLAG = 0;

/\* USER CODE END PV \*/

# Stickers rtfm manual read à acheter en ligne | SpreadshirtGestion des piles

## Lire la doc (RTFM)

Each task maintains its own stack. If a task is created using [xTaskCreate()](https://www.freertos.org/a00125.html) then the memory used as the task's stack is allocated automatically from the [FreeRTOS heap](https://www.freertos.org/a00111.html), and dimensioned by a parameter passed to the xTaskCreate() API function. If a task is created using [xTaskCreateStatic()](https://www.freertos.org/xTaskCreateStatic.html) then the memory used as the task's stack is pre-allocated by the application writer. Stack overflow is a very common cause of application instability. FreeRTOS therefore provides two optional mechanisms that can be used to assist in the detection and correction of just such an occurrence. The option used is configured using the [configCHECK\_FOR\_STACK\_OVERFLOW configuration constant](https://www.freertos.org/a00110.html).

### Stack Overflow Detection - Method 1

It is likely that the stack will reach its greatest (deepest) value after the RTOS kernel has swapped the task out of the Running state because this is when the stack will contain the task context. At this point the RTOS kernel can check that the processor stack pointer remains within the valid stack space. The stack overflow hook function is called if the stack pointer contain a value that is outside of the valid stack range.

This method is quick but not guaranteed to catch all stack overflows. Set configCHECK\_FOR\_STACK\_OVERFLOW to 1 to use this method.

### Stack Overflow Detection - Method 2

When a task is first created its stack is filled with a known value. When swapping a task out of the Running state the RTOS kernel can check the last 16 bytes within the valid stack range to ensure that these known values have not been overwritten by the task or interrupt activity. The stack overflow hook function is called should any of these 16 bytes not remain at their initial value.

This method is less efficient than method one, but still fairly fast. It is very likely to catch stack overflows but is still not guaranteed to catch all overflows.

Set configCHECK\_FOR\_STACK\_OVERFLOW to 2 to use this method

On a deux configurations pour vérifier le dépassement de pile.

Methode 1 : Vérifier que les pointeurs pointent dans une zone valide de la pile. Si ce n’est pas le cas, le stack overflow hook est appelé.

Methode 2 : Vérifie que les 16 dernier octets de la pile correspondent aux valeurs connues et n’ont pas été écrasé par autre chose.

## Configurer CubeIDE pour vérifier les dépassement de pile

Dans l’IOC, Middleware, FreeRTOS, Config Parameters on scroll à la partie Hook function related definitions

Text

Description automatically generated

On choisit l’option 1.

## Ecrire la fonction

**void vApplicationStackOverflowHook( TaskHandle\_t xTask, signed char \*pcTaskName );**

The xTask and pcTaskName parameters pass to the hook function the handle and name of the offending task respectively. Note however, depending on the severity of the overflow, these parameters could themselves be corrupted, in which case the pxCurrentTCB variable can be inspected directly.

Stack overflow checking introduces a context switch overhead so its use is only recommended during the development or testing phases.

On a fait une fonction simple :

**void** **vApplicationStackOverflowHook**( TaskHandle\_t xTask, **signed** **char** \*pcTaskName ){

**int** size = **snprintf** (print\_buffer, BUFFER\_SIZE, "STACK OVER FLOWN \r\nDepassement de la pile de %s\r\n",pcTaskName );

uart\_write(print\_buffer, size);

Error\_Handler();

}

On dit quelle tache a fait un dépassement de pile. Et on entre dans la boucle infinie du Error Handler. Le error handler désactive toutes les interruptions avec \_\_disable\_irq();.

## Stack Over Flow

Pour faciliter un stack over flow, on prend une taille de stack égale à 10 pour notre fonction bidon.

**Text

Description automatically generated with medium confidence**

On vous garde ce test avec le flag **int** SOV\_FLAG = 0;.

## Autres hooks

Text

Description automatically generated

Il existe d’autre hooks. RTFM ici <https://www.freertos.org/a00016.html>

On cherche dans la documentation de FreeRTOS, on trouve l’utilité des autres hooks :

IDLE HOOK :

The idle task can optionally call an application defined hook (or callback) function - the idle hook. The idle task runs at the very lowest priority, so such an idle hook function will only get executed when there are no tasks of higher priority that are able to run. This makes the idle hook function an ideal place to put the processor into a low power state - **providing an automatic power saving** whenever there is no processing to be performed.

TICK HOOK :

The tick interrupt can optionally call an application defined hook (or callback) function - the tick hook. **The tick hook provides a convenient place to implement timer functionality**.

MALLOC FAILED HOOK :

A malloc() failure hook (or callback) function that can be configured to **get called if pvPortMalloc() ever returns NULL**.

Defining the malloc() failure hook will **help identify problems caused by lack of heap memory** - especially when a call to pvPortMalloc() fails within an API function.

DAEMON TASK STARTUP HOOK :

The RTOS daemon task is the same as the [Timer Service Task](https://www.freertos.org/RTOS-software-timer-service-daemon-task.html). Sometimes it is referred to as the daemon task because the **task is now used for more than just servicing timers**.

# Statistiques dans l’IDE

## Activation des paramètres

Text

Description automatically generated

## Generer le code et debug mode

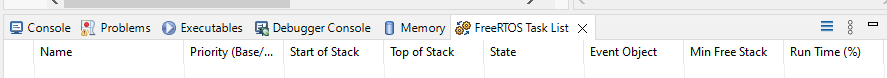
Générer le code et switch vers la vue debuggueur.

## Ajouter la fenêtre des statistiques

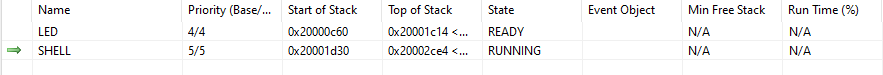
Graphical user interface, text, application, Word

Description automatically generated

Une nouvelle fenêtre près de la console s’affiche :

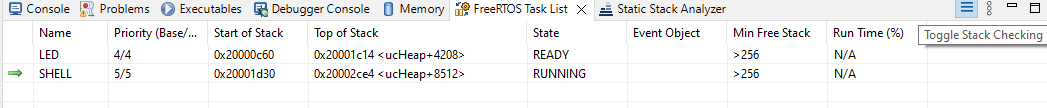


## Lancer et regarder les taches



On a bien deux taches, avec le shell qui tourne (RUNNING) et la LED qui est prête (READY).

## Toggle Stack Checking

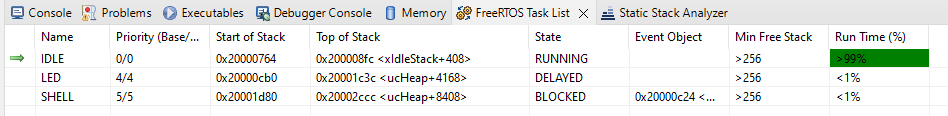


## Utilisation du CPU

**void** configureTimerForRunTimeStats(**void**);

**unsigned long** getRunTimeCounterValue(**void**);

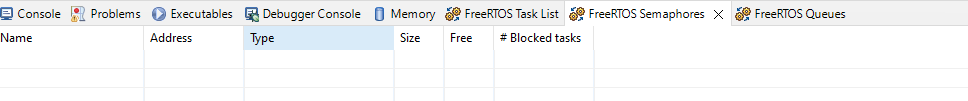
On écrit ces deux fonctions en suivant le tutoriel <https://the78mole.de/freertos-debugging-on-stm32-cpu-usage/>



On a l’usage de CPU par nos taches.

## Afficher les sémaphores et queues

On suit le même principe qu’avant : Windows>Show view> FreeRTOS > Selectionner semaphare et puis refaire pour avoir les Queues.



## Créer des sémaphores et queues pour les visualiser

Il faut un sémaphore partagé par deux fonctions. On utilise les codes d’exemple du cours.