Faculté des Sciences & Techniques — Université de Limoges

Master 1ère année

Systèmes Embarqués

Corrections TD 1

*Système embarqué & Programmation RTOS*

## Programmation RTOS

**1 –** a. Pourquoi disposer de « ***timer hardware*** » est intéressant ?
*Un « timer hardware » est un circuit indépendant du processeur qui réalise un compteur et qui est lié à une interruption matérielle.*
*Il permet d'automatiser un traitement indépendamment du processeur en fonction du temps.*

b. Quel intérêt par rapport à une **tâche freeRTOS** ?
*Par rapport à une tâche RTOS, il n'est pas traité par le CPU :*
- ◇ *il ne ralentit pas l'OS : pas d'instruction du CPU pour faire évoluer son compteur ;*
- ◇ *il est plus précis : il dépend directement de l'horloge du SoC, «* System on Chip *» ;*

**2 –** Est-ce que la présence d'un **port série** est toujours nécessaire sur un IoT ?
*Un port série permet une communication entre le «firmware» du Système Embarqué et l'utilisateur/développeur :*
- ▷ *faire un suivi du fonctionnement du logiciel : déboguage, log, suivi des différentes étapes de démarrage du SoC et du RTOS,* etc.
- ▷ *intéragir avec l'utilisateur/développeur : dans le cas où l'IoT est suffisamment puissant, il peut embarquer un Linux complet et offrir un "login" et un shell de commande où l'on peut récupérer des données confidentielles comme des identifiants, mot de passe et certificats d'accès à la plateforme Cloud du constructeur. Il peut alors servir de vecteur d'attaque vers ce cloud ou vers le réseau de l'utilisateur où est connecté cet IoT.*
  *On peut également essayer de débloquer des fonctionnalités non autorisées ou soumises au paieent d'une option.*

*En résumé, l'intérêt peut être de lutter contre l'obsolescence d'un IoT lorsqu'il n'est plus géré par le constructeur et le défaut de servir de vecteur d'attaque.*

**3 –** a. Les processeurs **ARM** et **Risc-V** disposent-ils pour toute leur gamme des mêmes instructions ? *Non, ils ont des jeux d'instructions plus ou moins étendus en fonction de la cible du processeur en terme d'usage et de coût :*
- ▷ *par exemple sur ARM, le jeu d'instruction réalisant des calculs flottants ou la disponibilité d'un jeu d'instruction sur 16bits (thumb) nécessitant moins de place peuvent être ou non intégrés ;*
- ▷ *en RISC-V, il existe des extensions similaires.*

b. Comment est-ce géré ?
*Sous ARM, il existe différentes familles de processeurs disposant de fonctionnalités différentes : cortex M0, M0+, M4 etc.*
*En RiscV, on dispose de différents processeurs avec différentes capacités.*
*C'est lors de la fabrication/achat du processeur que l'on choisit.*

**4 –** a. Si une tâche freeRTOS doit **démarrer** plus tard dans la « *vie* » du système, est-il nécessaire ou non de la créer au démarrage/allumage du système et pourquoi ?
*En général, on va installer toutes les tâches au démarrage :*
- ◇ *pour s'assurer de la disponibilité suffisantes de ressources nécessaires : périphériques, mémoire pour la pile, mémoire interne au RTOS pour la gestion des tâches dans le scheduler etc. ;*
- ◇ *l'évolution du RTOS est figée : on ne va pas charger dynamiquement de nouvelles tâches comme dans le cas d'un Linux (sauf dans le cas d'une m-à-j de l'OS ou dans le cas où une tâche ressemble à un script comme dans le cas de micro-python où le fait que ce soit un interprète le permet).*
- ◇ *pour spécialiser/finaliser le RTOS : pas besoin d'une liste chaînée pour la gestion des tâches s'il n'y a que 3 tâches en tout par exemple. Cela permet une économie de place et une accélération des traitements.*

b. Comment **l'activer** au moment où on en a besoin ?
*En :*
- ◇ *utilisant les priorités : la tâche peut être activé en lui donnant une priorité supérieure à celle de la tâche courante ;*
- ◇ *activant dans le RTOS :* `vTaskResume()` *;*
- ◇ *la libérant la sémaphore qu'elle attend pour démarrer.*

**5 –** a. Comment **entrelacer** l'exécution de plusieurs tâches $T_1$, $T_2$ et $T_3$ qui s'exécutent indéfiniment ?
*En démarrant les différentes tâches avec la* ***même priorité***

b. Comment peut-on faire si on veut, en plus, **contrôler** dans quel ordre l'exécution passe d'une tâche à l'autre ?
*Par exemple,* $T_1 \longrightarrow T_2 \longrightarrow T_3 \longrightarrow T_1 \longrightarrow T_2 \longrightarrow \ldots$
*En utilisant une sémaphore entre chaque couple de tâches : sur l'exemple, une sémaphore* $S_{12}$ *entre les tâches* $T_1$ *et* $T_2$*,* $S_{23}$ *entre les tâches* $T_2$ *et* $T_3$*,* $S_{31}$ *entre les tâches* $T_3$ *et* $T_1$*.*
- ◇ *la tâche* $T_2$ *attend la sémaphore* $S_{12}$ *;*
- ◇ *la tâche* $T_1$ *libère la sémaphore* $S_{12}$ *;*
- ◇ *la tâche* $T_2$ *peut alors démarrer ;*
- ◇ *la tâche* $T_1$ *attend la sémaphore* $S_{31}$ *;*

*etc*

**6 –** Un **ESP32** est installé dans un boitier plastique avec les éléments suivants :
- □ un écran LCD de contrôle capable d'afficher deux lignes de texte :
  - ◇ l'écran affiche de manière permanente le contenu d'un tampon mémoire de 32 caractéres réparti sur deux lignes de 16 caractères ;
  - ◇ la fonction `void write_screen(char *buffer);` mets à jour le contenu du tampon ;
- □ une LED rouge : reliée à une broche GPIO en sortie ;
- □ un bouton poussoir : relié à une broche GPIO en entrée.

Le travail de l'ESP32 est le suivant :
- ▷ l'heure `HH:MM` doit être affiché sur la seconde ligne de l'écran avec une mise à jour toute les minutes ;
- ▷ l'appui sur le bouton poussoir doit allumer la LED pendant 3s, afficher un texte sur la première ligne de l'écran « `START WORK` », et déclenche la tâche `task_work()` ;
- ▷ l'écran doit être capable d'afficher des messages en provenance de la tâche `task_work()` sur la ligne 1 tout en continuant à afficher l'heure sur la ligne 2.
- ▷ le travail de `task_work()` peut prendre plusieurs minutes.

*Vous allez définir comment, en freeRTOS, vous allez mettre en place le travail de cet ESP32.*

**Questions**

a. Est-ce que la mise à jour de l'écran peut créer des conflits ? Lesquels ?
*Oui, il va y avoir un conflit d'accès.*
*L'écran est décomposé en deux zones d'affichage mise à jour par deux tâches différentes :* `task_work()` *et une tâche qui va gérer la mise à jour de l'heure toutes les minutes : .*

b. Comment gérer le lien entre le bouton et la LED ? Est-ce qu'il y a des risques ?
*Il faut synchroniser l'illumination de la LED à l'appui sur le bouton et maintenir cet illumination pendant 3s.*
*Pour « découvrir » l'appui sur le bouton, on va utiliser une interruption.*
*On ne peut pas attendre 3s pour terminer le traitement de l'interruption car ce traitement bloque le déroulement du RTOS qui ne peut alors pas traiter les autres tâches et fonctions dont il a la responsabilité (le traitement d'une interruption bloque la gestion des autres interruptions et en particulier, celle qui permet de changer de contexte vers le RTOS).*
*Il faut donc gérer le délai de l'illumination de la LED en dehors de l'interruption.*

c. Comment déclencher le travail de la tâche `task_work()` avec l'appui sur le bouton ?
*En utilisant une sémaphore pour :*
- ◇ *bloquer la tâche* `task_work()` *en attente ;*
- ◇ *libérer cette sémaphore depuis l'interruption liée à l'appui du bouton.*

d. Comment « *bloquer* » la prise en compte du bouton tant que la tâche `task_work()` s'exécute ?
*On peut utiliser une variable booléenne :*
   - ◇ *écrite par la tâche `task_work()` à vrai lorsqu'elle commence et à faux lorsqu'elle finit ;*
   - ◇ *lue par la fonction gérant l'interruption liée au bouton pour savoir si elle doit ou non libérer la sémaphore.*

e. Comment allez vous gérer la mise à jour du temps ?
*On peut utiliser un timer hardware ou un timer système avec `xTimerCreate()`.*

f. Avec quels éléments de freeRTOS allez vous gérez les différents composants matériels et leur(s) interaction(s) compte tenu de votre analyse précédente ?
   - ◇ *le bouton va être géré par une interruption chargée de libérer une sémaphore réalisant la synchronisation avec la tâche `task_work()` ;*
   - ◇ *l'illumination de la LED peut être réalisée par une tâche synchronisée par une sémaphore qui au démarrage allume la LED, puis attend 3s avant de l'éteindre.*
     *Cette sémaphore est libérée dans le traitement de l'interruption ;*
   - ◇ *l'écran peut être associé à un buffer pour contenir le texte à afficher, et une file de messages avec `xQueueCreate()` pour protéger l'accès concurrent des tâches de mise à jour de ce buffer.*

**7 –** Soit le programme suivant :

```
 1 #define APPCPU 1
 2
 3 static SemaphoreHandle_t mutex_1;
 4 static SemaphoreHandle_t mutex_2;
 5
 6 void doTaskA(void *parameters) {
 7   while (1) {
 8     xSemaphoreTake(mutex_1, portMAX_DELAY);
 9     vTaskDelay(1 / portTICK_PERIOD_MS);
10
11     xSemaphoreTake(mutex_2, portMAX_DELAY);
12     vTaskDelay(500 / portTICK_PERIOD_MS);
13
14     xSemaphoreGive(mutex_2);
15     xSemaphoreGive(mutex_1);
16
17     vTaskDelay(500 / portTICK_PERIOD_MS);
18   }
19 }
20
21 void doTaskB(void *parameters) {
22   while (1) {
23     xSemaphoreTake(mutex_2, portMAX_DELAY);
24     vTaskDelay(1 / portTICK_PERIOD_MS);
25
26     xSemaphoreTake(mutex_1, portMAX_DELAY);
27     vTaskDelay(500 / portTICK_PERIOD_MS);
28
29     xSemaphoreGive(mutex_1);
30     xSemaphoreGive(mutex_2);
31
32     vTaskDelay(500 / portTICK_PERIOD_MS);
33   }
34 }
35
36 void setup() {
37
38   mutex_1 = xSemaphoreCreateMutex();
39   mutex_2 = xSemaphoreCreateMutex();
40
41   xTaskCreatePinnedToCore(doTaskA, "Task A", 1024, NULL, 2, NULL, APPCPU);
42   xTaskCreatePinnedToCore(doTaskB, "Task B", 1024, NULL, 1, NULL, APPCPU);
43   vTaskDelete(NULL);
44 }
45
46 void loop() {
47 }
```

Décrivez le fonctionnement des tâches à l'aide d'un chronogramme.
Est-ce que tout fonctionne bien ?