TD 7 – ROVIN
Synchronisation & Communication sur un RTOS

Le module ROVIN est doté d'un « pseudo » système d'exploitation multi-tâches (« Virtual Operating System », développé par Comfile Technology) chargé d'attribuer le processeur (selon une politique Round Robin) aux tâches de l'application. L'objectif de ce TD est d'ajouter **la couche communication et de synchronisation** entre les tâches, que n'offre pas le VOS natif. Cette couche s'appuie sur la mémoire partagée (XHeap, supposée initialisée à 0) pour que les tâches puissent s'échanger des informations.

L'application servant de « test » à l'implémentation de la couche de communication est constituée de deux tâches qui doivent s'échanger correctement une donnée (vitesse de clignotement de LED). Plus précisément, l'application est découpée en deux :

- Une tâche 1 se charge de la lecture d'un potentiomètre branché sur le convertisseur `ADC0` (fonctions AdcSet, AdcOn et AdcRead), de l'écriture sur la fenêtre de « Debug On-the-Fly », et enfin, de l'envoi de cette donnée à la tâche 2.
- Une tâche 2 se charge de la gestion du clignotement des LED. Elle lit tout d'abord la donnée fournie par la tâche 1, puis l'écrit sur la fenêtre de debug et enfin, fait clignoter le chenillard à la valeur indiquée (entre 10ms et 1 seconde).

L'interface utilisateur de cette application est la suivante :
* LEDs rouges : 8 bits de poids fort de la valeur du potentiomètre,
* LEDs jaunes : chenillard
* Debug On-the-Fly : Valeur écrite/lue en mémoire ou valeur émise/reçue par message

1) Communication boîte aux lettres à mémorisation simple, sans synchronisation

Réaliser une première version de la couche communication dans laquelle l'échange de la donnée ne s'appuie pas sur un mécanisme de synchronisation (ce qui revient donc à écrire et à lire directement en mémoire à l'adresse convenue). Pour cela, développer les primitives :

`void` **`Send_1`**`(unsigned int MsgBox, unsigned short MsgData)`

Envoi **non bloquant** d'un message porteur d'une donnée associée MsgData (valeur de type « short int ») sur une boîte aux lettres nommée MsgBox

`unsigned short` **`Receive_1`**`(unsigned int MsgBox)`

Retourne la valeur lue, par une réception **non bloquante** d'un message sur la boîte aux lettres MsgBox

Noter qu'évidemment, la primitive `Receive_1` retourne une valeur **inexploitable** si la réception est faite **avant** l'émission par `Send_1`…

2) Communication avec synchronisation unilatérale par signaux logiciels

Au vu du résultat obtenu à la fin de la question précédente, il semble judicieux de synchroniser les tâches **avant** d'échanger la donnée.

Il est donc demander ici d'installer les primitives manipulant les signaux logiciels :

`void` **`Signal`**`(unsigned int SignalName)`

Permet l'envoi **non bloquant** du signal logiciel SignalName. Cet envoi revient à effectuer une mémorisation (simple) du signal, soit à faire : SignalName = 1

`void` **`Wait`**`(unsigned int SignalName)`

Cette primitive est potentiellement **bloquante** car elle permet l'attente active (scrutation) du signal considéré. Cette primitive est donc non bloquante que si le signal est présent.

Utiliser ensuite ces deux primitives Signal et Wait pour synchroniser **unilatéralement** les tâches, afin que les primitives précédentes `Send_1` et `Receive_1` échangent correctement la donnée. Vérifier le fonctionnement unilatéral en passant une des deux tâches en mode debug (qui a pour effet de la bloquer).

3) Communication avec synchronisation bilatérale par signaux logiciels

Réaliser une synchronisation **bilatérale** (rendez-vous par attentes croisées garantissant l'attente, même si l'une « arrive » avant l'autre au point de rendez-vous). Comparer.

4) Communication avec synchronisation bilatérale par variable partagée

Développer la seconde version de la couche communication dans laquelle l'échange de la donnée est synchronisé, en mode Rendez-Vous (synchronisation bilatérale) sur variable partagée. Le principe est ici d'utiliser une boîte (RdVBox) pour l'échange d'une variable RdV, puis d'appliquer la politique suivante sur cette variable : « on écrit une valeur et on attend l'autre ».

`void` **`Send_2`**`(unsigned int MsgBox, unsigned int RdVBox, unsigned short MsgData)`

Envoi **bloquant** d'un message porteur d'une donnée associée MsgData

`unsigned short` **`Receive_2`**`(unsigned int MsgBox, unsigned int RdVBox)`

Retourne la valeur lue, par une réception **bloquante**

5) Communication avec synchronisation par sémaphores

Installer le mécanisme de synchronisation par sémaphore.

`void` **`Init_Sem`**`(unsigned int Sem, unsigned int ValeurSem)`

Primitive d'initialisation de la valeur du sémaphore

`void` **`V`**`(unsigned int Sem)`

Primitive non bloquante d'incrémentation de la valeur du sémaphore

`void` **`P`**`(unsigned int Sem)`

Décrémente le sémaphore et bloque la tâche appelante si ce dernier devient négatif.

Utiliser ensuite ces deux primitives P et V pour synchroniser **bilatéralement** les tâches (donc, les mettre en rendez-vous), afin que les primitives de communication `Send_1` et `Receive_1` échangent correctement la donnée.