TP 6 Interruptions

Nous révisons aujourd'hui le TP précédent en nous servant d'interruptions.

1 UART avec Interruptions

Le code du TP précédent fonctionnait en *polling* : l'attente dans les fonctions `HAL_UART_Receive` et `HAL_UART_Transmit` est active et bloque l'exécution du programme (on ne peut rien faire pendant ce temps). Modifiez ce code de façon à utiliser des interruptions, c'est à dire à employer l'algorithme réactif suivant : écouter les messages entrants ; quand un octet x est reçu, renvoyer "x `mississippi, `"; quand le message a été transmis, écouter les messages entrants.

1. Activez l'interruption `USART1_IRQn` dans le NVIC (fonction `HAL_NVIC_EnableIRQ`) ; et définissez une fonction

   ```
   void USART1_IRQHandler()
   ```

   pour traiter cette interruption. Elle se contentera d'appeler la fonction correspondante de HAL, `HAL_UART_IRQHandler` (consulter le fichier `stm32f3xx_hal_uart.c` pour plus d'informations).
2. Définissez une fonction

   ```
   void HAL_UART_RxCpltCallback(UART_HandleTypeDef *UartHandle)
   ```

   qui sera appelée par HAL à la réception d'une trame UART. Cette fonction fera clignoter une LED et appellera `HAL_UART_Transmit_IT` pour transmettre le message adéquat.
3. Définissez une fonction

   ```
   void HAL_UART_TxCpltCallback(UART_HandleTypeDef *UartHandle)
   ```

   qui sera appelée par HAL à la fin de la transmission de votre message. Cette fonction lancera l'écoute d'un message d'un octet, et réarmera de ce fait l'interrruption.
4. Juste avant la boucle principale de votre programme, amorcez l'écoute d'un message avec un appel à `HAL_UART_Receive_IT`.
5. Dans la boucle principale, endormissez le processeur dès que possible à l'aide de l'instruction `__WFI()`.
6. Testez le tout : le comportement doit être le même qu'au TP précédent. Ce qu'on a gagné :
   — on consomme moins de courant (car jamais d'attente active)
   — le processeur est disponible pour une autre tâche pendant les lectures/écritures sur l'UART.
7. Faites clignoter une autre LED à l'aide de SysTick. Bravo, vous avez un programme multitâche !

2 Serveur d'impression de messages

Dan un nouveau répertoire, copier les drivers que vous avez écrit précédemment (LEDs, bouton, UART etc.), et partez d'un `main` vide.

1. Implémenter dans un fichier `io.c` les deux fonctions de lecture et d'écriture, que `printf()` et `scanf()` s'attendront à pouvoir utiliser :

   ```
   int _read(int file, char *data, int len)
   int _write(int file, char *data, int len)
   ```

   ces deux fonctions utiliseront l'API blocante de l'UART pour respectivement recevoir et envoyer `len` caractères sur celui-ci.

2. Dans `main`, écrire un programme simple en utilisant `printf` et `scanf`, qui demande à l'utilisateur un mot [1] et un nombre, et compte jusqu'à ce nombre :

   ```
   Quel mot? mississippi
   Jusqu'à combien dois-je compter? 5
   1 mississippi 2 mississippi 3 mississippi 4 mississippi 5 mississippi
   Quel mot?
   ```

   À chaque ligne affichée, faites clignoter les LEDs.

Un problème potentiel de votre programme est que les appels à `printf` sont bloquants (puisqu'ils reposent sur votre implémentation de `_write()`, qui l'est) : tant que le message n'est pas effectivement envoyé, la fonction `printf()` ne retourne pas. Vous pouvez visualiser ceci grâce aux LEDs, qui clignotent à une fréquence qui dépend de la longueur du mot entré (i.e. de la taille du message envoyé).

Nous allons maintenant écrire une version non bloquante de `_write()`, qui garde les chaînes de caractère à écrire dans une file, implémentée avec un *buffer tournant*. Écrire un message revient alors a l'ajouter simplement dans la file ; de façon asynchrone, l'UART "consomme" les chaînes de la file, jusqu'à ce qu'il n'y en ait plus.

3. Déclarer la structure de donnée des "tâches d'impression" (un tableau de caractère et sa longueur), ainsi qu'un tableau de 64 cases qui contiendra des tâches, et deux indices dans ce tableau : celui de la tâche la plus récemment ajoutée, et celui de la plus vieille :

   ```
   #define QUEUE_SIZE 64
   typedef struct { char data[32]; int length } task;
   task tasks[QUEUE_SIZE];
   int first_task; // 0 < first_task < QUEUE_SIZE
   int last_task; // 0 < last_task < QUEUE_SIZE
   ```

4. Écrire une fonction `void enqueue_task(char *data, int len)` qui ajoute une tâche à la file (elle copiera la chaîne de caractère dans la file).
5. Écrire une fonction `void dequeue_task()` qui retire un élément de la file s'il y en a un, et l'envoie à l'UART pour transmission non bloquante.
6. Dans le *callback* de la transmission, appelez `dequeue_task()`, de façon à continuer à envoyer tant qu'il y a des tâches dans la file.
7. Écrire finalement le corps de la fonction `_write()`, qui ajoute une tâche à la file, et déclenche l'écriture si nécessaire. Testez le comportement du programme maintenant, et comparez la vitesse de clignotement des LEDs.
8. Que se passe-t-il si on ajoute plus de 64 tâches d'impression ? Comment y remédier ?

1. on prévoira un buffer mémoire de 32 caractères pour le stocker.