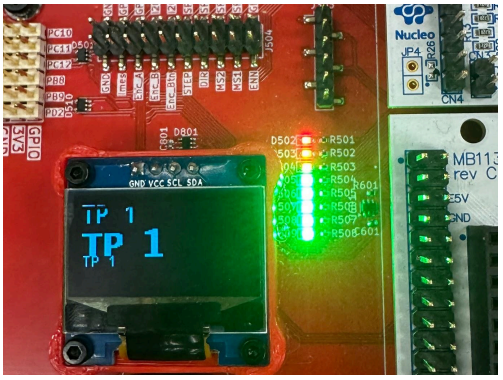
Kirthika MURUGENDRAN et Dignon TRAN

TP3 - Un framework multithreadé minimaliste

Table des matières



7.1. Introduction - préparation

7.1.1. Programme en polling

L'objectif de cette partie est de comprendre comment fonctionne un programme en polling, et trouver des solutions pour améliorer le programme.

Le polling permet au programme d'attendre un évènement en le testant régulièrement, pour renvoyer le résultat associé une fois réalisée (source : Polling (Wikipédia).
Dans notre cas, notre programme en polling permet de renvoyer un motif type "chenillard" en fonction de l'appui sur le bouton, ou encore de modifier la vitesse du chenillard lorsque que l'un des boutons a été appuyé.

La préparation n'a pas été effectuée. On suppose que les grandes lignes de ce programme en polling serait une série de if-else, voire une fonction switch avec différents cas correspondants aux actions et à leurs réponses.

Cependant, le programme est inefficace car il vérifie sans arrêt la condition et consomme du temps et de la mémoire (le programme est **actif**).

7.1.2. Programme en multi tâche simple

Le mode polling étant peu efficace, il faut installer un **scheduler**.

7.3. Le scheduler, usage du timer

7.3.1. Calcul des paramètres du timer

Figure : Fréquence d'horloge du timer (on lit une fréquence $f_{Timer} = 80\ MHz$, donc une période $T = 1.25 * 10^{-7} s = 125\ ns$)

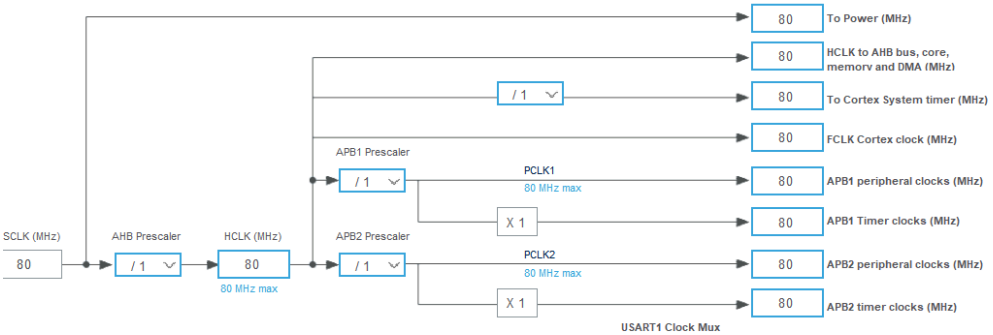
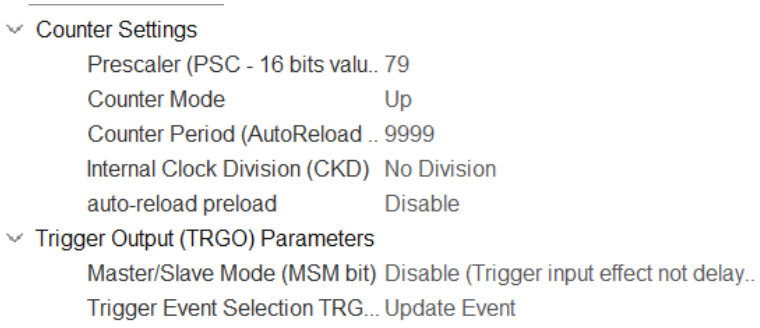


Figure : Paramètres du timer 4 (PSC et ARR)



Les interruptions permettent d'interrompre le fonctionnement normal pour effectuer une action prévue. Elles s'exécutent automatiquement.
On a un débit de 115200 bits/s pour la liaison considérée. On a une fréquence de 80 MHz. D'après les TP 1 et 2, une liaison série est codée sur 10 bits : 8 bits de données, 1 bit d'entrée et 1 bit de sortie. On a alors :
$N_{car} = \frac{115200}{10} = 11520$ caractères par seconde. Donc la plus longue chaîne de caractères que l'on puisse envoyer avec cette liaison, **pour une tâche**, est composée de **11 520 caractères**. Pour une tâche, le processeur se focalise à 100% sur elle. Cependant, nous avons 3 tâches, le processeur doit les traiter toutes les 3, et allonge donc le temps de traitement. Il faut multiplier la période par 3 : $T' = 3 * T = 375ns$.

Afin de pouvoir vérifier si les interruptions fonctionnent correctement, nous avons ajouté des fonctions *printf()* à chacune des 3 fonctions (*taskLed(), taskButton(), taskScreen()*). Les *printf()* s'exécutent indéfiniment, malgré que la fonction *HAL_TIM_PeriodElapsedCallback()* soit en dehors de la boucle infinie. En effet, il y a toujours un des cas qui est vérifié, ce qui renvoie la phrase inscrite dans le *printf()*.
Note : Dans la partie 7.1.2, on nous indique que le scheduler ne nous permet plus d'utiliser la fonction *printf()*. Or, notre programme renvoyait correctement les *printf()*.

Figure : Test des interruptions

```c
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
  /* USER CODE BEGIN Callback 0 */
    static int taskNumber=0;
        if(htim -> Instance == TIM4){
            taskNumber=(taskNumber+1)%3;
            switch(taskNumber){
            case 0 : taskLED();
                     break;
            case 1 : taskButton();
                     break;
            case 2 : taskScreen();
                     break;
            }
        }

  /* USER CODE END Callback 0 */
  if (htim->Instance == TIM6)
  {
    HAL_IncTick();
  }
  /* USER CODE BEGIN Callback 1 */

  /* USER CODE END Callback 1 */
}
```

```c
void taskLED(){
    printf("LED ON\n\r");
}

void taskButton(){
    printf("Button ON\n\r");
}

void taskScreen(){
    printf("Screen ON\n\r");
}
```

```
Led ON
Button ON
Screen ON
Led ON
Button ON
Screen ON
Led ON
Button ON
Screen ON
Led ON
Button ON
Screen ON
Led ON
```

7.4. La tâche de gestion des LEDs

7.4.1. Le driver logiciel des LEDs

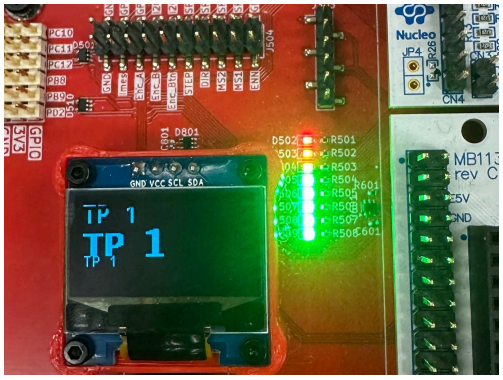


Figure : Les LED se sont allumées après exécution du programme

La fonction *LED_Update()* permet de faire allumer toutes les LED. On assigne les états définis dans le tableau *led_bar* (*port* et *pinNumber*) pour chacune des LED et on les rentre dans la fonction *HAL_GPIO_WritePin(),* qui permet de donner les états définis aux LED.

Figure : Fonction *LED_Update()* complétée

```
void LED_Update(){
    int i =0;
    for (i=0; i < LED_BAR_SIZE; i++){
        HAL_GPIO_WritePin(led_bar[i].port, led_bar[i].pinNumber, GPIO_PIN_SET);
    }
}
```

7.4.2. La tâche taskLED()

L'objectif de cette partie est d'afficher un motif sur la carte et faire fonctionner le chenillard.
.
La variable locale *numero_motif* augmente au fil du temps. Mais selon la valeur de *index_tableau_motif*, les valeurs prises par la variable sont différentes.

Figure : Capture de la console affichant les LED allumées au fil du temps

```
numero_motif = 0
numero_motif = 0
numero_motif = 0
numero_motif = 0
numero_motif = 0
numero_motif = 0
numero_motif = 0
numero_motif = 1
numero_motif = 1
numero_motif = 1
numero_motif = 1
numero_motif = 1
numero_motif = 1
numero_motif = 1
numero_motif = 2
numero_motif = 2
numero_motif = 2
numero_motif = 2
numero_motif = 2
numero_motif = 2
numero_motif = 2
numero_motif = 3
```

On a fait plusieurs tests, et on note :

index = 0: va de 0 à 15 avec une répétition de 7 fois pour chaque valeur
index = 1: va de 0 à 7 avec répétition 7 fois le même index
index = 2: va de 0 à 7 avec répétition de 7 sauf le 0 avec répétition de 6 de temps en temps et 7 fois d'autres fois, ce qui n'est pas normal puisque nous avons défini les tableaux de taille 8 (comprenant donc la LED 0 à la LED 7, donc les 8 LEDs).

À partir de index = 3, la console tourne à l'infini avec une répétition de 7 pour chacune des valeurs (les valeurs allaient au-delà de 1100).
Problème noté : Nous avons pris beaucoup de temps à faire afficher le chenillard sur la carte. Plusieurs erreurs et incompréhensions ont été notées, notamment dû au fait de faire en sorte que l'indice du tableau *led_bar* correspond à sa taille (on a notamment utilisé la division par 2, mais pour les plus grandes valeurs de *numero_motif*, on a pensé à effectuer une opération modulo 7 avec %).

Autre problème noté : une fois que nous avions réussi à faire afficher le chenillard, le motif ne s'affiche qu'une seule fois, alors qu'on voit un défilement à l'infini des valeurs de *numero_motif* dans la console. Le problème n'a pas pu être résolu pendant le TP car c'était la fin du TP.

7.5. La tâche de gestion des boutons

Dans la structure *BUTTON*, *isOn* et *hasBeenPressed* peuvent prendre les états 0 (donc éteint) et 1 (allumé). Le code suivant n'a pas pu être testé pendant la séance de TP. On a rencontré plusieurs problèmes de compilation venant de la définition de la structure *button*. Mais le code nous semble correspondre aux attentes de la question posée.

```
void BUTTON_Update() {
    int i;
    for (i = 0; i < BUTTON_SIZE; i++){
        int new_value = HAL_GPIO_ReadPin(joystick[i].port, joystick[i].pinNumber);
        joystick[i].hasBeenPressed = (1 - joystick[i].isOn)* new_value;
        joystick[i].isOn = new_value;
```

7.5.1. La tâche taskButton()

Cette partie n'a pas pu être entamée.

7.6. La tâche de gestion de l'écran

Cette partie n'a pas pu être traitée pendant le TP.

Commentaires

Le programme au 7.3. ne fonctionnait pas au début car il fallait inclure *main.h* dans le fichier *button.c* car tous les *define* sont dans le *main.h*. Le prof a ajouté cette remarque dans l'énoncé du TP dans la partie 7.5, alors qu'on avait besoin avant pour compiler les anciens programmes (car il semblait évident que lorsqu'on importe les fichiers, on les incluait tous en même temps pour gagner du temps).

Un des deux dans le binôme a aussi rencontré des erreurs de compilation, le prof a aidé à régler le problème en fermant les anciens projets (pour éviter que les erreurs signalées viennent des autres projets), et lorsqu'on a éteint puis rallumé stm32, le build est réapparu dans la barre d'outil et on a réussi à compiler le projet. Il ne compilait pas correctement car les modifications du programmes n'étaient pas sauvegardées donc le compilateur affichait les erreurs précédentes non encore résolues.
On a perdu beaucoup de temps sur la partie 7.4.2. à essayer de faire clignoter selon un motif car on n'a pas enlevé (mettre en commentaire) la fonction qui laisse les led à 1 (*LED_Test_All()*).

Conclusion

Le TP en lui-même ne semblait pas difficile à première vue, mais on rencontrait souvent des erreurs de compilation, ce qui nous a beaucoup ralenti durant le TP. Certaines erreurs auraient pu être évitées si on avait eu une meilleure compréhension (et une meilleure manipulation de STM32) des codes fournis par le prof dans l'énoncé du TP. Cependant, nous avons trouvé assez satisfaisant de voir les LEDs s'allumer.