Activité 2 : Utilisation des *Timer* et gestion des Interruptions

Periph'Team - INSA de Toulouse

1 Objectifs pédagogiques

- ♢ Savoir programmer les unités *Timer* généraux au niveau registre (base de temps).
- ♢ Savoir programmer le *NVIC* pour gérer les interruptions (autorisation, priorité).
- ♢ Savoir mettre en œuvre une sortie en mode *PWM* à l'aide d'un *Timer*
- ♢ Savoir construire une bibliothèque niveau driver
- ♢ Savoir mettre au point son programme sous KEIL

2 Notion de *Timer* : les fondamentaux

Si les *GPIO* sont les indispensables d'un μcontrôleur, les *Timers* en sont les incontournables.... Vous trouverez un point complet sur Moodle dans la section *Timers* : *Généralités sur les Timers (pdf)*,

- ► Lisez attentivement le document.
- ► Faites un nouveau projet avec un nouveau répertoire en respectant la structure de répertoires proposée au TP GPIO. Il est possible de dupliquer le répertoire contenant le projet *GPIO* puis de modifier les fichiers, ou bien de faire un nouveau projet ("from scratch").
- ► Dans la procédure *main*, configurez le *Timer 2* pour qu'il ait une période de $500ms$. Pour cela faites un choix arbitraire (mais judicieux!) sur les valeurs de *PSC* et *ARR*. La configuration doit permettre le démarrage du *Timer*.
- ► Vérifiez en simulation que le registre de comptage *CNT* évolue (ce test est incomplet, mais il permet une première approche de vérification du code).

3 Rédaction du driver générique *MyTimer.c / MyTimer.h*

- ► Rédigez le driver *MyTimer.c* et *MyTimer.h* avec le header suivant imposé donné ci-dessous. Pour l'instant seules les fonctionnalités décrites dans la partie précédente sont à implémenter.

> *Les fonctions de démarrage et d'arrêt des MyTimer.c seront en réalité des macros, tout à fait adaptées à ce genre de fonctionnalité simple.*

```
#ifndef ...
#define ...

#include ...

typedef struct
{
        TIM_TypeDef * Timer;  // TIM1 à TIM4
        unsigned short ARR;
        unsigned short PSC;
} MyTimer_Struct_TypeDef;


/*
**************************************************************************************
        * @brief
        *       @param -> Paramètre sous forme d'une structure (son adresse) contenant les
            informations de base
        *       @Note  -> Fonction à lancer systématiquement avant d'aller plus en détail dans les
            conf plus fines (PWM, codeur inc...)
**************************************************************************************************
*/

void MyTimer_Base_Init(MyTimer_Struct_TypeDef * Timer);

#define MyTimer_Base_Start(Timer) (...)
#define MyTimer_Base_Stop(Timer) (...)

#endif
```

- Faites un essai identique à celui de la partie précédente pour un *Timer* au choix (en simulation).

4 La mise en œuvre des interruptions

Les tests opérés jusqu'à maintenant ne permettent pas de valider précisément les timings mis en jeu dans au niveau des *Timers*. Par ailleurs, pour l'instant, le driver ne sert pas à grand chose. Il est donc indispensable d'ajouter le mécanisme d'interruption lors des débordements. L'idée est donc, pour l'utilisateur final, de disposer d'un driver permettant :

- de régler la période du *Timer* (via *PSC* et *ARR*), ce qui est déjà fait dans la version du driver que vous avez écrite ;
- de déclencher une interruption à chaque débordement du *Timer*.

Vous trouverez un point complet sur Moodle dans la section Interruptions (*IT*) : *Généralités sur les interruptions*.

- Lisez attentivement le document.
- Expliquez :
  - sur quels bits et sur quels registres agir au niveau du *Timer* pour activer les interruptions ;
  - sur quels bits, sur quels champs de bits et sur quels registres agir au niveau du *NVIC* pour activer les interruptions ;
  - le mécanisme d'interruption depuis le débordement jusqu'à la l'exécution du *handler* conçu pour le traitement de cette interruption.
- Complétez le driver *MyTimer* avec la fonction d'activation d'interruption donné par le prototypage suivant :

```
/*
**************************************************************************************************

  * @brief
                * @param : - TIM_TypeDef * Timer :  Timer concerne
                          -  char Prio : de 0 a 15
            * @Note :  La fonction  MyTimer_Base_Init doit avoir ete lancee au prealable
```

```
*******************************************************************************************************

*/
void MyTimer_ActiveIT (TIM_TypeDef * Timer, char Prio);
```

> ***Remarque 1*** *: on écrira le* handler *d'interruption dans le driver* MyTimer. *Le nom du* handler *spécifique à chacun des 4* Timers *est disponible dans le fichier* startup *du projet. Dans ce* startup*, chaque* ***handler*** *est en effet défini précisément mais en* **weak**. *En redéfinissant un de ces* ***handler****, donc en réutilisant le même nom, cette nouvelle version sera acceptée par le linker et viendra alors écraser ceux définis dans le* startup.
>
> ***Remarque 2*** *: on pourra utiliser les fonctions toutes faites* NVIC_EnableIRQ(...) *et* NVIC_SetPriority(...) *disponibles dans les headers fournis dans le projet.*
>
> ***Remarque 3*** *: pensez à* ***toujours*** *rabaisser le flag d'interruption au niveau du* Timer *dès l'entrée dans le* handler. *A défaut la demande d'interruption ne sera jamais acquittée et le* handler *va se redéclencher en boucle sitôt le traitement terminé.*

- Vérifiez en simulation la périodicité d'appel au *handler* du *Timer* en mettant un point d'arrêt au début de ce *handler* et en utilisant la fonction de chronomètre de KEIL (petit insert en bas à droite dans l'application en mode debug).
- Procédez ainsi et vérifier le bon fonctionnement de votre driver pour les 4 *Timers*.

5 Utilisation des pointeurs de fonctions

Jusqu'à maintenant, vous avez compris et vous avez mis en œuvre des fonctionnalités *Timer* permettant de :

- régler la période du timer,
- déclencher une interruption de priorité choisie lors des débordement,
- faire une action lors de chaque interruption (même si pour l'instant le handler est vide...).

Il s'agit maintenant de donner à la couche driver une séparation nette avec le main. Or si l'utilisateur désire par exemple changer l'état d'une LED toutes les 500ms, il va être obligatoire d'écrire un code au niveau du driver (plus précisément dans le corps du *handler*). On souhaite cependant interdire cette manière de faire. On va chercher plutôt à ce que l'utilisateur écrive son code de clignotement en dehors du driver mais directement dans le fichier contenant la procédure principale (appelons *Principal.c*). Ce qui implique un appel vers une fonction de *Principal.c*, depuis le driver *MyTimer.c* lors d'une interruption. Plus précisément , le *handler* d'interruption doit appeler la fonction de changement d'état de la LED située dans *Principal.c*.

Cela ne peut se faire que par l'utilisation de **pointeurs de fonctions**.

Vous trouverez un point complet sur Moodle dans la section Cours - Détail de langage C & Structuration logicielle *Fiche méthode : les pointeurs de fonctions*.

- Si vous n'êtes pas à l'aise avec les pointeurs de fonctions, lisez le document et trouvez le moyen d'utiliser un pointeur de fonctions qui réponde à notre besoin.
- Modifiez le pilote pour intégrer la notion de pointeurs de fonctions. Le prototype de la fonction d'activation du *Timer* devient alors :

```
void MyTimer_ActiveIT (TIM_TypeDef * Timer, char Prio, void (*IT_function) (void));
```

Le dernier argument attendu dans cette fonction est donc l'adresse d'une fonction et permettra d'initialiser le pointeur de fonctions du *Timer* concerné à l'intérieur du driver.
Le driver doit permettre un fonctionnement adéquat pour les 4 *Timers*.

- Vérifiez en simulation la périodicité du *Timer* en créant une fonction *Callback* dans le programme principal (c'est à dire la fonction qui sera appelée par la couche driver lors de chaque interruption). Ce *Callback* ne fera que basculer l'état d'une LED (*toggle*). On utilisera bien entendu le driver *MyGPIO* pour configurer la broche souhaitée en sortie.
- Vérifiez également en mode réel et pour les 4 *Timers*.

6 La Pulse Width Modulation

Les fonctionnalités fondamentales d'un *Timer* ont été vues et mises en œuvre. Il s'agit maintenant d'exploiter d'autres possibilités plus avancées, avec en premier exemple, la *PWM*.

Vous trouverez un point complet sur Moodle dans la section Pulse Width Modulation (*PWM*) : *Cours sur la PWM*

- Ajoutez au driver *MyTimer* la fonction qui permet d'initialiser le *Timer* en mode *PWM*. **Attention** cette fonction ne fait ni l'initialisation des ports *GPIO* ni la configuration du *Timer* pour sa fréquence de débordement. Le protypage de cette fonction peut correspondre à :

```
/**
  * @brief
    * @param  ....
    * @Note   Active le channel spécifié suTimerr le timer spécifié
    *             la gestion de la configuration  I/O n'est pas faite dans cette fonction
    *             ni le réglage de la période de la PWM (ARR, PSC)
  */
void MyTimer_PWM(TIM_TypeDef * Timer, char Channel);
```

- Prévoyez le protypage et écrivez la fonction qui permette de lancer une *PWM* sur un canal particulier d'un *Timer* donné.
- Prévoyez aussi une fonction qui permette juste de modifier le rapport cyclique d'une *PWM*.
- Ecrivez une petite application test qui mette en place, par exemple, une *PWM* à 100kHz avec un rapport cyclique de 20% sur le *Timer* et le canal de votre choix.
- Vérifiez le bon fonctionnement de votre application, sur la cible simulée à l'aide de la fenêtre *Logic Analyser* et en réel sur un oscilloscope.