
Activité 1 : Masques, Timer et Interruption

Periph'Team - INSA de Toulouse

1 Aspect temps réel de l'application

Pour fonctionner, l'application a besoin d'une horloge temps réel qui donne le cadencement toutes les 10 ms. Pour cela nous avons besoin d'un **timer** bien calibré et d'un **système d'interruptions opérationnel**. La figure 1 montre un diagramme de séquences qui illustre l'exécution attendue par la fonction `main`. On y voit la succession des appels de fonctions entre les divers modules du logiciel. Le timer, dernier élément du diagramme, est le seul objet matériel.

Lorsque les interruptions et le timer sont bien configurés, les interruptions du timer agissent selon l'évolution de la figure 2.

A chaque débordement du *timer* (toute les 10 ms), le *NVIC* lance le *handler* associé au timer (dans *MyTimer.c*), puis lance la tâche *Chrono_Task_10ms* contenue dans *Chrono.c*.

2 Travail à faire

2.1 Préparation

- Récupérer le projet *KEIL*, *ProjKEIL_Base_Chrono.zip* sous le répertoire *MDK-ARM* du projet, le déployer, lancer le projet *KEIL* (.uvprojx) et vérifier que la compilation et l'édition de liens se font sans erreurs (il y a quelques warning attendus à ce stade).

2.2 Manipulation de bits au niveau des registres de périphériques

- Lire les documents (textes et/ou vidéos) concernant les masques. Vous devez par exemple comprendre parfaitement ce type d'écriture :

```
TIM1->CR2|=0x1 ;
TIM1->CR2=0xFFFF ;
TIM1->CR2 &= ~0x0040 ; // pas genial comme lisibilite
TIM1->CR2=0xFFFF ;
TIM1->CR2&=~(1<<6) ; // plus lisible non ?
GPIOB->CRL=(0xFFFF) ;
GPIOB->CRL&=~(0xF<<8) ;
GPIOB->CRL|= (0x5<<8) ;
GPIOB->ODR|=GPIO_ODR_ODR1 ; // grand style !
```

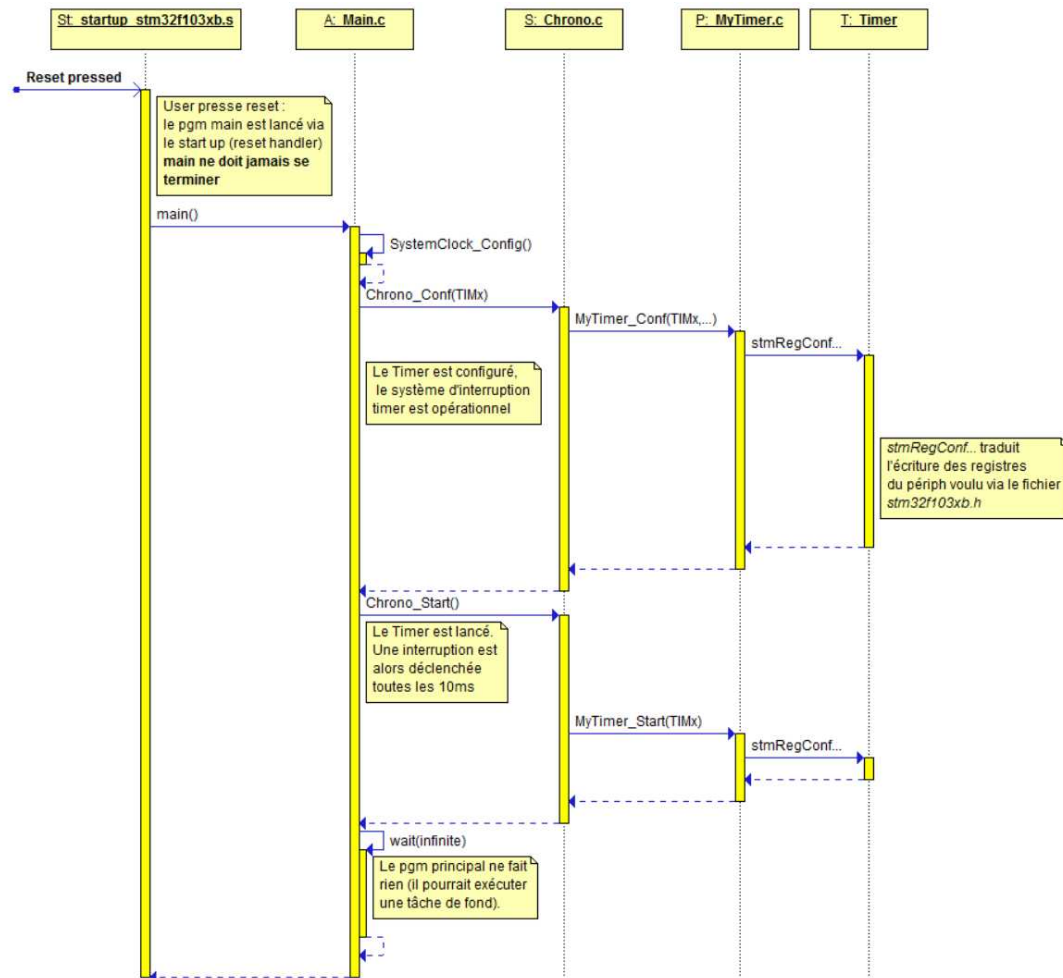



Figure 1: Diagramme de séquence du programme main depuis le reset

- ▶ Testez les lignes précédentes.
- ▶ Ecrivez les lignes de code pour mettre à 1 le bit *CPOL* du *SPI control register1 SPI_CR1* et à 0 le bit *DFF* de ce même registre.
- ▶ Mettez la valeur 2 dans le champs de bits *STOP* du *Control register 2 USART_CR2*.
- ▶ Tester et vérifier à chaque fois en simulation que le code écrit produit bien ce qui est attendu. Pour cela vous devez parvenir à visualiser les registres utiles. Il y a au moins trois méthodes pour ça (expérimentez les trois) :
 - vous trouvez l'adresse physique de ces registres et vous visualisez dans une *memory window*,¹
 - vous cliquez sur l'icône *system viewer windows* , et choisissez le périphérique que vous souhaitez,
 - vous ouvrez le périphérique souhaité : *peripherals > ...* la vérification est moins évidente puisque chaque bit est éclaté par fonction. On ne peut donc pas interpréter la position des bits dans le registre. Par contre, lors de la programmation concrète de périphérique, ces fenêtres périphériques seront extrêmement utilisées (presque exclusivement).

Toutes ces affectations de bits sont sans intérêt pour la suite du projet, les lignes de code seront donc à effacer par la suite.

¹Pour déterminer l'adresse d'un registre, utilisez le *datasheet* de *STM103RB* pour trouver son adresse physique de base et le *reference manual RM008* pour trouver les adresses relatives des registres. Une seconde méthode consiste à utiliser le fichier *stm32f103xb.h*, à y chercher le registre puis à "reconstruire" l'adresse effective.

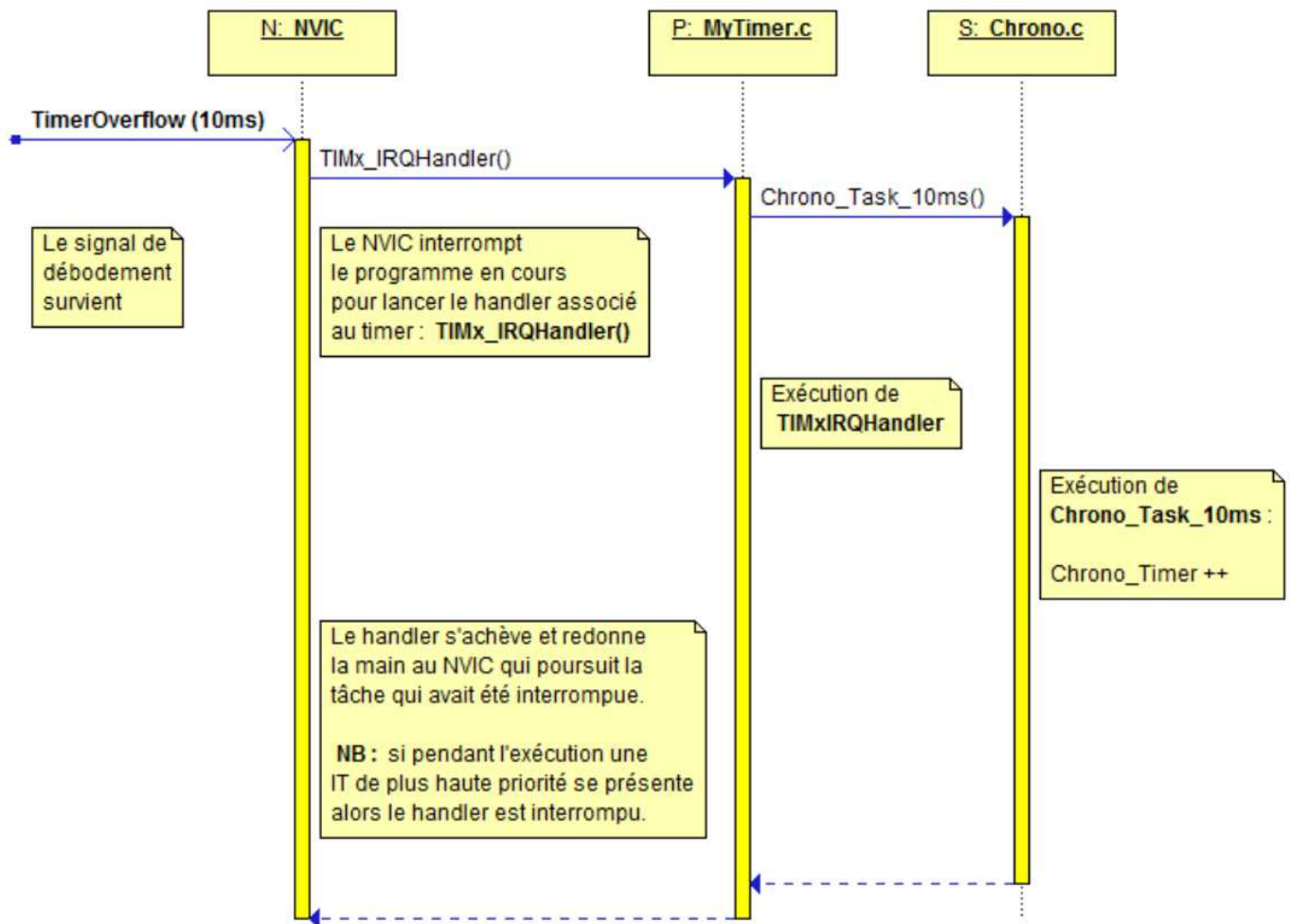


Figure 2: Diagramme de séquence du système d'interruption Timer

NB : Toutes les opérations qui viennent d'être simulées sous *KEIL* sont normalement impossibles en l'état. Cela ne fonctionne que grâce à une faille du simulateur. En effet, pour écrire dans un registre de périphérique (quel qu'il soit), **il est indispensable de valider son horloge en tout premier lieu sans quoi aucune écriture n'est possible.**

NB : Attention, tous les registres ne sont pas accessibles en lecture/écriture (r/w). Donc si lors de tests, les diverses fenêtres du simulateur donnent des résultats incohérents, vérifier les propriétés r/w dans le *reference manual RM008.pdf*.

Maintenant que les rudiments de manipulation de registres sont maîtrisés (périphériques, registres de périphérique, bits, champs de bits, masque pour mettre à 1, à 0) vous pouvez passer à la programmation du périphérique Timer.

3 Le module MyTimer.c (voir Moodle : Mooc Periph, cours timer, interruption...)

3.1 Le timer

- ▶ Lire les documents concernant la *programmation C & structuration logicielle* pour comprendre la philosophie de la conception.
- ▶ Lire les documents, vidéos ... concernant les timers. Vous devez avoir compris :
 - le cœur d'un timer, sa constitution de base, sa fonction essentielle,

- les registres et bits qui permettent de configurer le timer (comment le démarre-t-on, comment règle-t-on sa durée, up/down etc ...). Le reference manual *RM008.pdf* sera indispensable. Le *header file*, *stm32f103xb.h* sera également très utile pour la définitions des bits.
- ▶ Rédigez les premières fonctions (*MyTimer_Conf*, *MyTimer_Start*, *MyTimer_Stop*) en vous basant sur *MyTimer.h* (voir les prototypes et les commentaires). On précise que les registres qui contrôlent les horloges des divers périphériques font partie du périphérique *Reset & Clock Control (RCC)*.
- ▶ On pourra procéder à des essais élémentaires en simulation pour vérifier que le Timer choisi évolue correctement (bien vérifier que l'horloge du timer soit activée avant toute chose!!).

3.2 Les interruptions

- ▶ Lire les documents, vidéos ... concernant les interruptions. Vous devez avoir compris :
 - le concept d'interruption (le mécanisme),
 - sa mise en œuvre depuis le périphérique en passant par le *NVIC*,
 - la notion de priorité,
 - le mécanisme de *callback* en exploitant un pointeur de fonction.

En complément du *RM008.pdf*, vous aurez besoin du *programming manual PM0056.pdf* qui traite des détails du *NVIC*. Outre le fichier *stm32f103xb.h*, il sera nécessaire de consulter aussi le fichier *core_cm3.h* (lui même inclus dans *stm32f103xb.h*) qui contient notamment la définition de la structure du *NVIC*.

- ▶ Rédiger le reste des fonctions du module concernant les interruptions. Vous devrez mettre en œuvre le mécanisme d'interruptions du *STM32* ainsi qu'utiliser des pointeurs de fonction pour pouvoir appeler le *callback* *Chrono_Task_10ms* depuis le *handler* d'interruption associé.
- ▶ A ce stade, vous pouvez vérifier le fonctionnement précis du timer associé à la mise en place des interruptions timer en simulation :
 - Pour cela placez un point d'arrêt dans le callback et vérifiez avec le chronomètre de KEIL l'intervalle de temps entre deux entrées successives dans l'interruption. Celui ci doit être de 10ms précise.

3.3 Finalisation de cette première partie du chronomètre

- ▶ Compléter toutes les fonctions du module *Chrono.c*.
- ▶ Vérification :
 - Lancez tout simplement le chronomètre et visualisez en simulation et en réel l'évolution de la variable structurée *Chrono_Time*.

Attention : Cette variable est globale mais *STATIC*. Cela signifie que bien qu'étant globale (nécessaire pour être vue par le handler et aussi par la fonction de configuration), elle est privée au fichier donc non référencée dans les variables globales du projet. Ainsi, pour la voir évoluer sur le debugger, on pourra momentanément enlever l'attribut *STATIC* (le temps de constater le bon fonctionnement). Si l'on veut travailler vraiment proprement (sans enlever *STATIC*), on peut utiliser dans le *main* la fonction *Chrono_Read*. Il suffit de déclarer un pointeur sur *type Time* (global !) et de l'affecter avec l'adresse de *Chrono_Time* retournée par *Chrono_Read*.