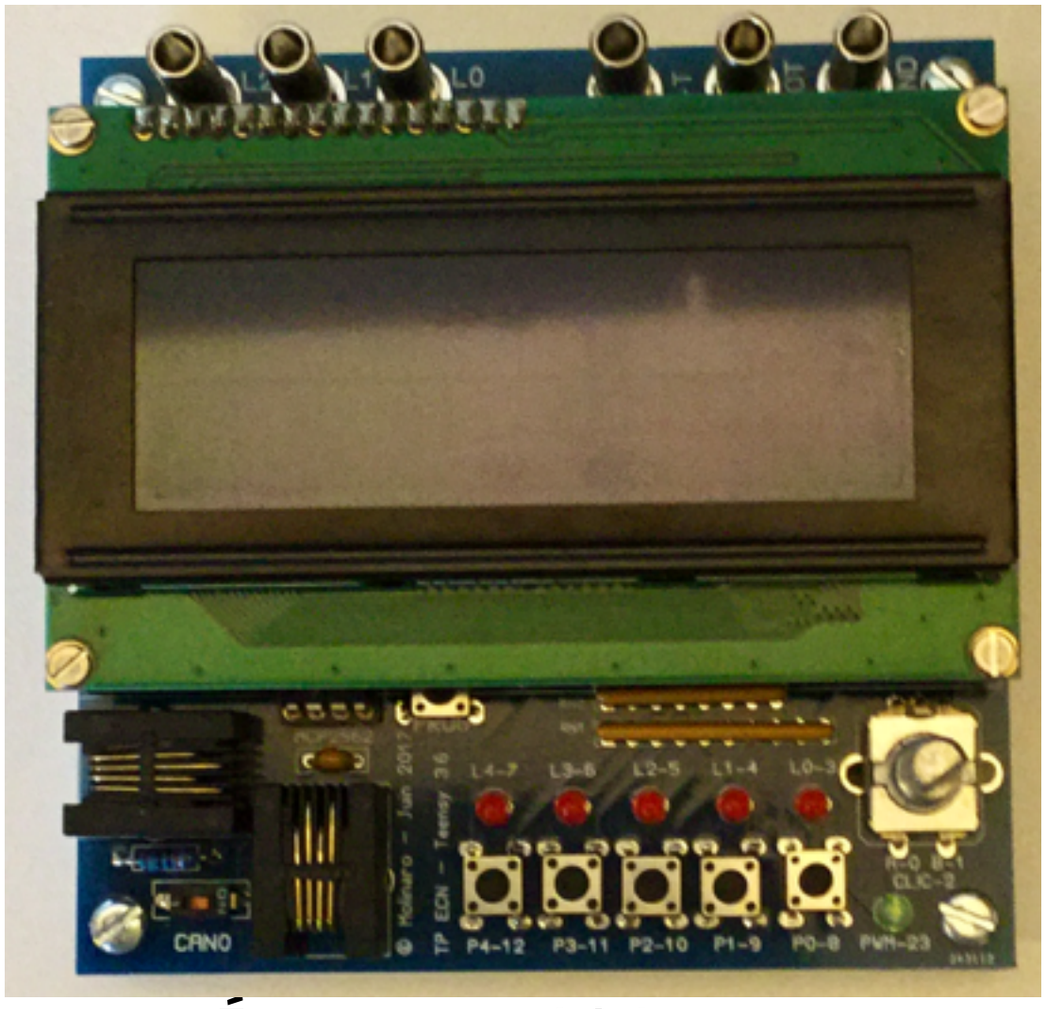
Temps Réel



Étape 07-systick-interrupt

Pierre Molinaro

Juin 2018

Description de cette étape

Objectif : compter le temps sous interruption.

Ceci permettra de disposer des fonctions :

- **busyWaitUntil**, qui exprime l'attente jusqu'à une date absolue ;
- **busyWaitDuring**, attente durant une délai (cette fonction existe déjà, elle est re-écrite) ;
- **millis**, qui retourne la date courante, c'est-à-dire le nombre de milli-secondes depuis le démarrage du compteur SysTick ;
- **systick**, qui retourne la valeur courante du compteur SysTick.

CENTRALE NANTES

Fonctions d'attente actuelles

Deux fonctions d'attente ont été implémentées à l'étape 06 (fichier **time.cpp**) :

```
void busyWaitDuring_initMode (INIT_MODE_ const uint32_t inDelayMS) {
  const uint32_t COUNTFLAG_MASK = 1 << 16 ;
  for (uint32_t i=0 ; i<inDelayMS ; i++) {
    while ((SYST_CSR & COUNTFLAG_MASK) == 0) {} // Busy wait, polling COUNTFLAG
  }
}

void busyWaitDuring (USER_MODE_ const uint32_t inDelayMS) {
  const uint32_t COUNTFLAG_MASK = 1 << 16 ;
  for (uint32_t i=0 ; i<inDelayMS ; i++) {
    while ((SYST_CSR & COUNTFLAG_MASK) == 0) {} // Busy wait, polling COUNTFLAG
  }
}
```

Dans cette étape, nous allons modifier le fichier **time.cpp** et la fonction **busyWaitDuring** de façon que le comptage du temps s'effectue sous interruption.

Activer l'interruption du compteur `SysTick`

Pour activer l'interruption SysTick, il suffit mettre le bit `TICKINT` du registre `SYST_CSR` à 1. Dans le fichier time.cpp, ajouter une fonction exécutée en mode **INIT** :

```
static void activateSystickInterrupt (INIT_MODE) {
  SYST_CSR |= SYST_CSR_TICKINT ;
}

MACRO_INIT_ROUTINE (activateSystickInterrupt) ;
```

Maintenant, l'interruption se déclenche toutes les milli-secondes. Examinons le fichier **teensy-3-6-interrupt-vectors.s** qui définit le vecteur d'interruptions :

```
  .word __system_stack_end
@--- ARM Core System Handler Vectors
  .word reset.handler @ 1
  .word interrupt.NMI @ 2
  ..............
  .word interrupt.PendSV @ 14
  .word interrupt.SysTick @ 15
@--- Non-Core Vectors
  .word interrupt.DMA0_DMA16 @ 16
  ..............
```

L'interruption SysTick est la n°15, et la routine d'interruption doit avoir le nom assembleur **`interrupt.SysTick`**.

Où est définie la fonction `interrupt.SysTick` ?

En effet, comme le vecteur d'interruption nomme **`interrupt.SysTick`**, cette fonction est définie par défaut -- sinon on aurait une erreur d'édition de liens.

Il faut regarder dans le fichier engendré par la compilation `zSOURCES/interrupt-handlers.s` :

```
interrupt.SysTick:
  movs r0, #15
  b    unused.interrupt
```

Et `unused.interrupt` est défini comme une boucle sans fin dans **unused-interrupt.s** :

```
unused.interrupt:
  b    unused.interrupt
```

C'est-à-dire que par défaut, le déclenchement d'une interruption bloque l'exécution.

Note : l'étape 09 modifiera le comportement par défaut au déclenchement d'une interruption : un message d'erreur sera imprimé sur l'afficheur LCD.

La fonction `interrupt.SysTick` (1/2)

Il faut signifier au système de compilation de ne pas engendrer le code par défaut pour l'interruption SysTick.

Ceci est fait en rajoutant une annotation particulière dans les fichiers d'en-tête. Dans le cas présent, on ajoute la ligne suivante dans **time.h** :

```
//$interrupt-section SysTick
```

Le compilateur ignore cette ligne : c'est un commentaire.

Le système de compilation examine ligne par ligne tous les fichiers d'en-tête à la recherche de type d'annotations (il y aura d'autres qui seront présentées dans les étapes suivantes). Il repère la chaîne `//$interrupt-section` qui doit être suivie par un nom valide d'interruption.

Si on recompile le programme avec l'annotation, on obtient une erreur d'édition de liens : le symbole i**nterrupt.section.SysTick** est indéfini. En effet, la fonction **interrupt.SysTick** dans `zSOURCES/interrupt-handlers.s` est devenue :

```
interrupt.SysTick:
  ldr   r0, =0x400FF084  @ Address of GPIOC_PSOR control register
  movs  r1, # (1 << 5)   @ Port D13 is PORTC:5
  str   r1, [r0]         @ turn on
  b     interrupt.section.SysTick
```

Nous commentons ce code page suivante.

CENTRALE NANTES

La fonction `interrupt.SysTick` (2/2)

Commentaires sur ce code :

```
interrupt.SysTick:
  ldr   r0, =0x400FF084  @ Address of GPIOC_PSOR control register
  movs  r1, # (1 << 5)   @ Port D13 is PORTC:5
  str   r1, [r0]         @ turn on
  b     interrupt.section.SysTick
```

Les trois premières instructions allument la led Teensy, pour montrer l'activité processeur. Notons qu'actuellement ce code est inutile (la led est en permanence allumée), mais sera utile en présence de l'exécutif.

La dernière ligne effectue un branchement vers la fonction nommée `interrupt.section.SysTick` : il faut donc définir cette fonction. Nous allons le faire page suivante.

Définir la fonction `interrupt.section.SysTick`

Déclarer la fonction dans **time.h** :

```
void systickInterruptServiceRoutine (SECTION_MODE) asm ("interrupt.section.SysTick") ;
```

Le nom C++ est libre, mais la fonction doit être déclarée dans le mode `SECTION`. Par contre, le nom assembleur est imposé.

Implémenter la fonction dans **time.cpp** :

```
static volatile uint32_t gUptime ;

void systickInterruptServiceRoutine (SECTION_MODE) {
  gUptime += 1 ;
}
```

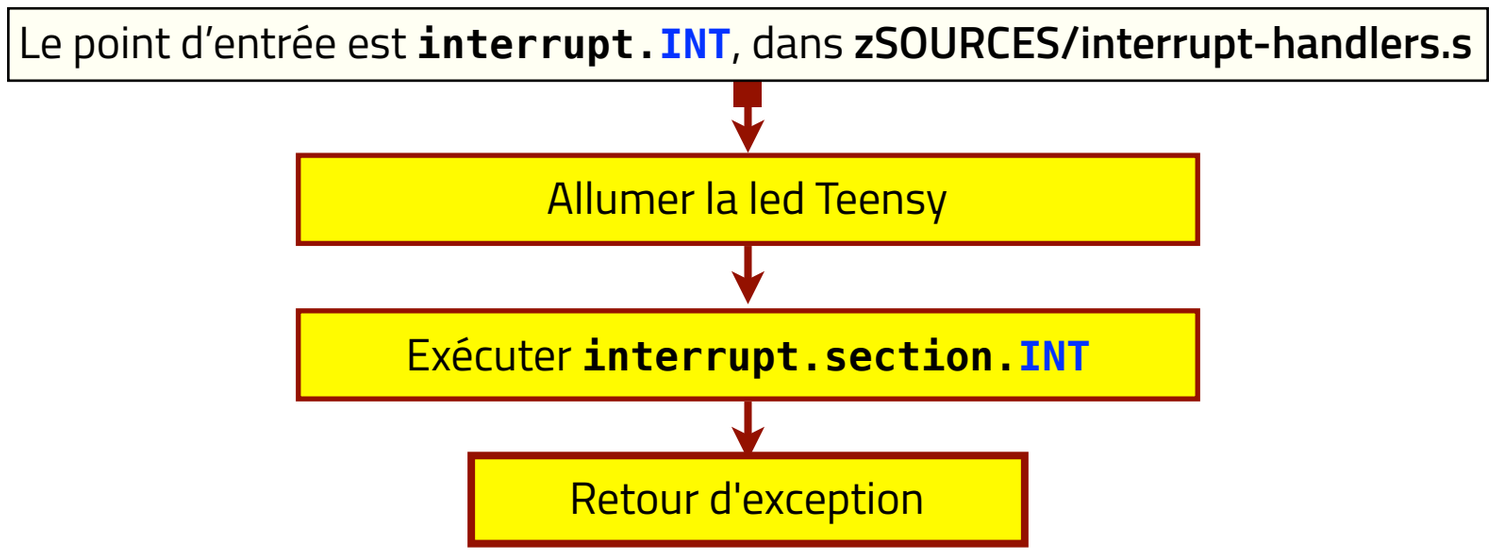
Ainsi, la variable `gUptime` est incrémentée toutes les milli-secondes : elle contient la date courante. Remarques sur sa déclaration :

- **static** limite la visibilité de la variable au fichier courant (ici **time.cpp**) ;
- **volatile** est **OBLIGATOIRE** car la variable va être partagée entre une routine d'interruption et des routines en mode USER (dont **busyWaitUntil**, voir page suivante).

Organigramme d'une routine d'interruption, mode SECTION

En résumé, une routine d'interruption en mode **SECTION** est déclarée dans un fichier d'en-tête par :

```
//$interrupt-section INT
void routine (SECTION_MODE) asm ("interrupt.section.INT") ;
```



Fonctions busyWaitUntil et busyWaitDuring

Deux fonctions de gestion du temps doivent être écrites :

- **busyWaitUntil**, qui exprime l'attente jusqu'à une date absolue ;
- **busyWaitDuring**, attente durant une délai (cette fonction existe déjà, elle est re-écrite).

La fonction **busyWaitUntil** est :

```
void busyWaitUntil (USER_MODE_ const uint32_t inDeadlineMS) {
  while (inDeadlineMS > gUptime) {}
}
```

Modifier l'implémentation de la fonction **busyWaitDuring** : il suffit d'appeler **busyWaitUntil** (avec les bons arguments…)

Fonctions `millis` et `systick`

Écrire une fonction **millis** qui retourne la date courante, c'est-à-dire le nombre de milli-secondes depuis le démarrage du compteur SysTick. Cette fonction peut être appelée dans n'importe quel mode (pas d'annotation de mode).

Écrire une fonction **systick** qui retourne la valeur du registre SYST_CVR, qui contient la valeur courante du compteur SysTick. Cette fonction peut être appelée dans n'importe quel mode (pas d'annotation de mode).

CENTRALE NANTES

Travail à faire

Dupliquer le répertoire de l'étape précédente et renommez-le par exemple **07-systick-interrupt**.

Implémenter les fonctions citées dans les pages précédentes (fichiers **time.h** et **time.cpp**).

Modifier la fonction **setup** de façon à écrire la valeur retournée par **millis** au début de la première ligne.

Modifier la fonction **loop** de façon à ce qu'elle attende durant 1000 ms, puis affiche au début de la deuxième ligne la valeur retournée par **millis**. Constater que la période d'exécution de **loop** n'est pas 1000 ms.

Ensuite, modifier la fonction **loop** pour obtenir une période d'exécution d'exactement 1000 ms.

CENTRALE NANTES