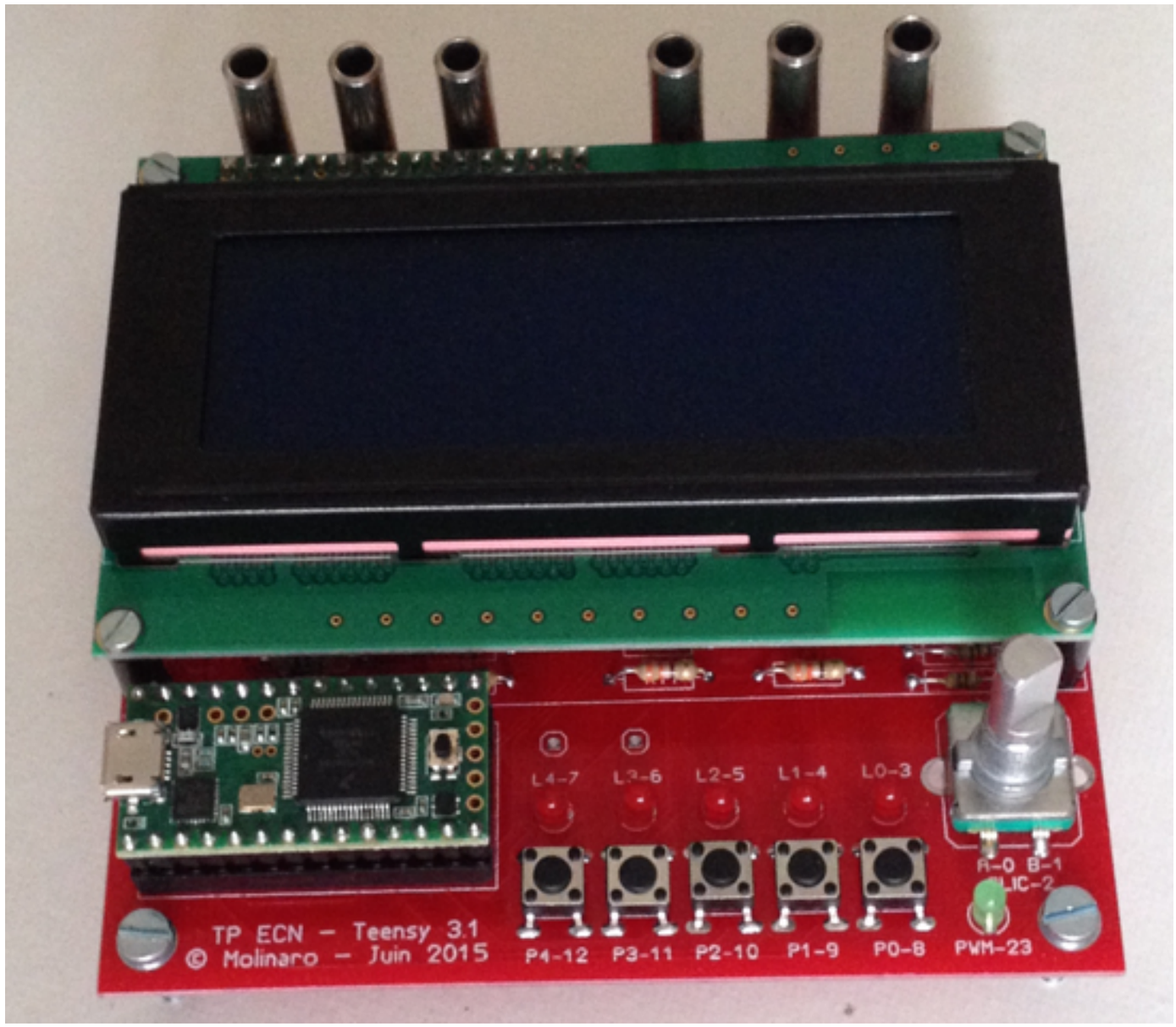
Temps Réel


L4-7
L3-6
L2-5
L1-4
L0-3
R-0 B-1
LIC-2
TP ECN - Teensy 3.1
© Molinaro - Juin 2015
P4-12
P3-11
P2-10
P1-9
P0-8
PWM-23

But de cette partie (1/2)

Objectif :

- *ajouter aux routines les annotations de mode logiciel.*

Travail à faire :

Réaliser un programme qui incrémente quatre variables globales :

- dans une routine d'interruption périodique ;
- dans la routine `loop`, par l'intermédiaire de quatre appels de service ;

Au bout de 5 secondes, la routine d'interruption périodique est désinstallée et les quatre variables sont affichées.

But de cette partie (2/2)

Dans `set-loop.c`, programme `09`

```
//-------------------------------------------------------------------------*

void svc_serviceA (void) {
  gCompteurA ++ ;
}

//-------------------------------------------------------------------------*

void loop (void) {
  serviceA () ;
  ……
}
```

La routine `svc_serviceA` est une routine de service, elle ne doit pas être appelée directement.

La routine `serviceA` est celle qui doit être appelée à partir de `loop`.

Si on écrit :

```
void loop (void) {
  svc_serviceA () ;
  ……
}
```

Pas d'erreur de compilation, mais erreur à l'exécution.

Le but de cette partie est de mettre en place un système d'annotation qui permette à la compilation de déceler ce type d'erreur.

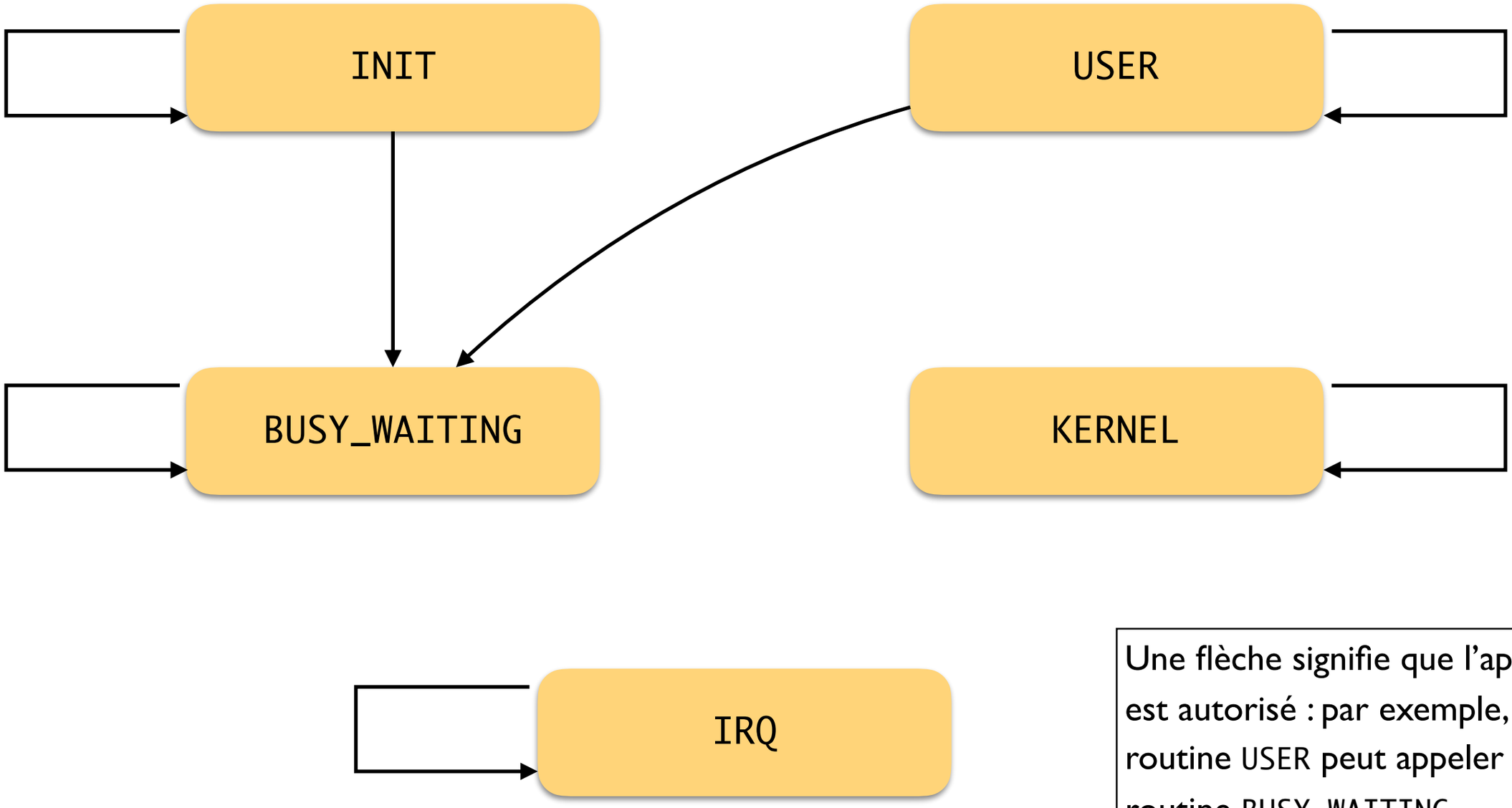
Les modes logiciels

Un mode logiciel est une information statique (c'est-à-dire établie à la compilation) qui s'applique à une fonction C, et qui caractérise la situation dans laquelle cette fonction s'exécutera.

Dans le programme `09`, on distingue les modes logiciels suivants :

- mode `INIT` : c'est le mode des routines `setup` et `loop` ;
- mode `USER` : c'est le mode des routines `setup` et `loop` ;
- mode `BUSY_WAITING` : l'attente active, qui peut être appelée par les routines d'initialisation et les routines `setup` et `loop` ;
- mode `IRQ` : les routines d'interruption ;
- mode `KERNEL` : les routines d'implémentation des services (appelées via `svc`).

Graphe des modes logiciels



Une flèche signifie que l'appel est autorisé : par exemple, une routine `USER` peut appeler une routine `BUSY_WAITING`.

Ce graphe sera modifié dans les programmes des étapes ultérieures.

Comment implémenter l'annotation de mode ?

Comment implémenter le graphe des modes, de façon que la vérification soit réalisée à la compilation ? **Par des classes C++ !**

```
class USER_mode_class {
  private : USER_mode_class (void) ;
  private : USER_mode_class & operator = (const USER_mode_class &) ;

  public : USER_mode_class (const USER_mode_class &) ;
} ;

class BUSY_WAITING_mode_class {
  private : BUSY_WAITING_mode_class (void) ;
  private : BUSY_WAITING_mode_class & operator = (const BUSY_WAITING_mode_class &) ;

  public : BUSY_WAITING_mode_class (const BUSY_WAITING_mode_class &) ;
  public : BUSY_WAITING_mode_class (const INIT_mode_class &) ;
  public : BUSY_WAITING_mode_class (const USER_mode_class &) ;
} ;
```

Par exemple, on déclare :

```
void busyWaitingDuringMS (const BUSY_WAITING_mode_class & inMode, const uint32_t inDurationInMS) ;
```

Et on pourrait écrire :

```
void loop (const USER_mode_class & inMode) {
  busyWaitingDuringMS (inMode, 500) ;
}
```

L'appel est correct grâce à ce constructeur.

Mise en œuvre de l'annotation de mode

```
void busyWaitingDuringMS (const BUSY_WAITING_mode_class & inMode, const uint32_t inDurationInMS) ;
```

Problème, la déclaration ci-dessus apporterait un inconvénient : l'annotation de mode apparait dans le code exécutable.

Solution retenue :

- utiliser un jeu de macros pour définir l'annotation de mode ;
- compiler chaque fichier source deux fois :
  - ✳ une première fois en C++, la macro `CHECK_ROUTINE_CALLS` étant définie, pour vérifier les annotations de mode ; le code engendré n'est pas utilisé ;
  - ✳ une seconde fois en C, les annotations de modes étant ignorées ; le code engendré est celui utilisé pour construire le code exécutable.

```
Checking (thumb) busy-waiting.c
/Users/pierremolinaro/dev-arm/dev-arm-Intel-Darwin-gccarm-5_2-2015q4-20151219-openocd-0.8.0/bin/arm-none-eabi-gcc -mcpu=cortex-m4 -
mthumb -x c++ -DCHECK_ROUTINE_CALLS -fomit-frame-pointer -Os -foptimize-register-move -Wall -Werror -Wreturn-type -Wformat -Wsign-
compare -Wpointer-arith -Wparentheses -Wcast-align -Wcast-qual -Wwrite-strings -Wswitch -Wuninitialized -ffunction-sections -fdata-
sections -fno-stack-protector -fshort-enums -fno-rtti -fno-exceptions -Woverloaded-virtual -Weffc++ -fno-threadsafe-statics -Wmissing-
declarations -c sources/busy-waiting.c -o zBUILDS/busy-waiting.c.check.oo -DSTATIC= -Isources -MD -MP -MF zBUILDS/busy-
waiting.c.check.oo.dep
Compiling (thumb) busy-waiting.c
/Users/pierremolinaro/dev-arm/dev-arm-Intel-Darwin-gccarm-5_2-2015q4-20151219-openocd-0.8.0/bin/arm-none-eabi-gcc -mcpu=cortex-m4 -
mthumb -fomit-frame-pointer -Os -foptimize-register-move -Wall -Werror -Wreturn-type -Wformat -Wsign-compare -Wpointer-arith -
Wparentheses -Wcast-align -Wcast-qual -Wwrite-strings -Wswitch -Wuninitialized -ffunction-sections -fdata-sections -fno-stack-protector
-fshort-enums -std=c99 -Wstrict-prototypes -Wbad-function-cast -Wmissing-declarations -Wimplicit-function-declaration -Wno-int-to-
pointer-cast -Wno-pointer-to-int-cast -Wmissing-prototypes -c sources/busy-waiting.c -o zBUILDS/busy-waiting.c.oo -DSTATIC= -Isources -
MD -MP -MF zBUILDS/busy-waiting.c.oo.dep
```

Comment écrire les annotations de mode (1/2)

Placer les annotations suivantes dans les en-têtes de routines.

Si la routine n'a pas d'argument, utiliser `INIT_MODE`, `USER_MODE`, `BUSY_WAITING_MODE`, `KERNEL_MODE`, `IRQ_MODE`. Par exemple :

```
void setup (USER_MODE) {
   ......
}
```

Si la routine a des arguments, placer comme premier argument : `INIT_MODE_`, `USER_MODE_`, `BUSY_WAITING_MODE_`, `KERNEL_MODE_`, `IRQ_MODE_`.

```
void busyWaitingDuringMS (BUSY_WAITING_MODE_
                          const uint32_t inDurationInMS) {
   ......
}
```

Attention, pas de virgule.

Comment écrire les annotations de mode (2/2)

Placer les annotations suivantes dans les appels de routines.

Si la routine appelée n'a pas d'argument, utiliser `MODE`. Par exemple :

```
void loop (USER_MODE) {
  serviceA (MODE) ;
  ……
}
```

Si la routine appelée a des arguments, placer comme premier argument : `MODE_`.

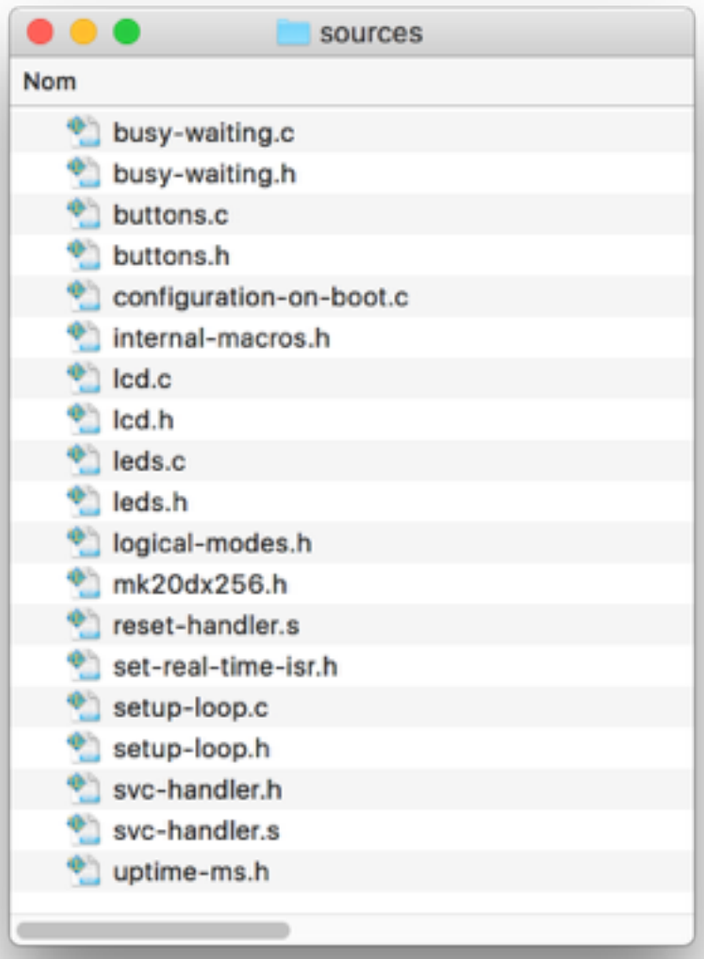
```
void loop (USER_MODE) {
  serviceA (MODE) ;
  busyWaitingDuringMS (MODE_ 500) ;
  ……
}
```

Attention, pas de virgule.

Travail à faire

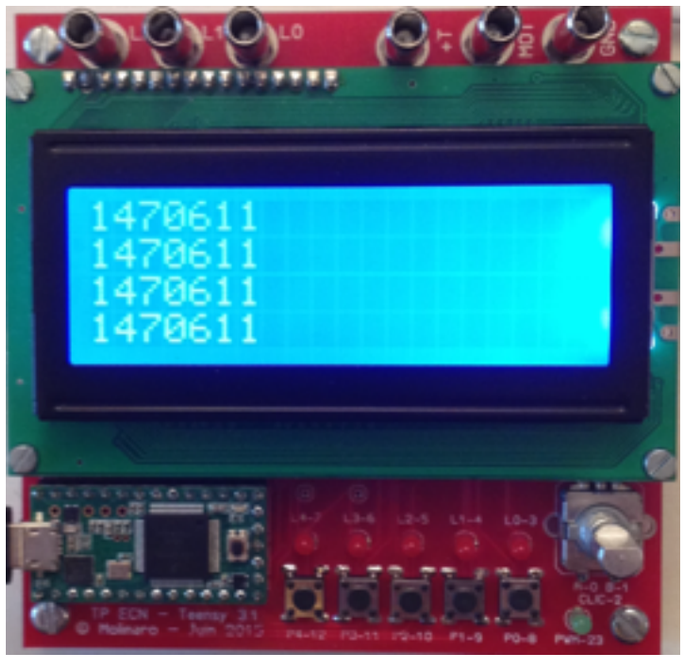
Comme de nombreuses modifications seraient à faire, une archive presque complète du nouveau programme est fournie sur le serveur pédagogique :

- récupérer sur le serveur pédagogique l'archive `10-modes-logiciels.tbz` ;
- seul fichier à compléter : `setup-loop.c`, de façon à réaliser le même programme que le `09`.



Résultat attendu

Les quatre variables globales sont incrémentées de manière atomique, on obtient à chaque fois quatre valeurs identiques. Ces valeurs peuvent changer d'une exécution à l'autre. L'affichage est le même que pour le programme précédent 09.



Pour ceux qui sont en avance

Voici quelques compléments pour ceux qui sont en avance.

Écrire un service qui retourne la valeur courante du compteur `sysTick`.

Utiliser ce service pour obtenir la durée d'exécution d'un service.

Obtenir la valeur courante du compteur `sysTick`

La valeur courante du compteur `sysTick`. est obtenue en lisant la valeur du registre de contrôle `SYST_CVR`.

Mais ce registre est lisible uniquement en mode système, sa lecture en mode utilisateur, est interdite (et déclenche une exception *Usage Fault*).

La seule façon est d'écrire un service qui retourne sa valeur.

```
uint32_t systickCurrentValue (USER_MODE) ;
uint32_t svc_systickCurrentValue (KERNEL_MODE) {
  return SYST_CVR ;
}
```

La valeur retournée est un entier compris entre `0` et 95999. Il décroit à la fréquence de 96 MHz, et lorsqu'il atteint `0`, il est rechargé à 95999.

Écrire un programme affichant la valeur obtenue par un appel de ce service.

Calculer la durée d'exécution d'un service

Pour commencer, on va calculer la durée d'exécution du service `systickCurrentValue`.

Écrire dans la routine `setup` (ou `loop`) le code suivant :

```
busyWaitingDuringMS (MODE_ 1) ;
// Ici, le compteur vient juste d'être rechargé
const uint32_t t1 = systickCurrentValue (MODE) ;
const uint32_t t2 = systickCurrentValue (MODE) ;
// Comme c'est en fait un décompteur, t1 > t2
printUnsigned (MODE_ t1 - t2) ;
```

La valeur affichée est la durée du service `systickCurrentValue`, exprimée en nombre de cycles à 96 MHz. Il suffit de diviser ce nombre par 96 pour obtenir la durée en µs.

Calculer la durée d'exécution d'un service

On va maintenant calculer la durée d'exécution d'un service d'incrémentation de variable, par exemple `serviceA`.

Écrire dans la routine `setup` (ou `loop`) le code suivant :

```
busyWaitingDuringMS (MODE_ 1) ;
// Ici, le compteur vient juste d'être rechargé
const uint32_t t1 = systickCurrentValue (MODE) ;
serviceA (MODE) ;
const uint32_t t2 = systickCurrentValue (MODE) ;
// Comme c'est en fait un décompteur, t1 > t2
printUnsigned (MODE_ t1 - t2) ;
```

La valeur affichée est la somme de la durée du service `systickCurrentValue` et du service `serviceA`, exprimée en nombre de cycles à 96 MHz.