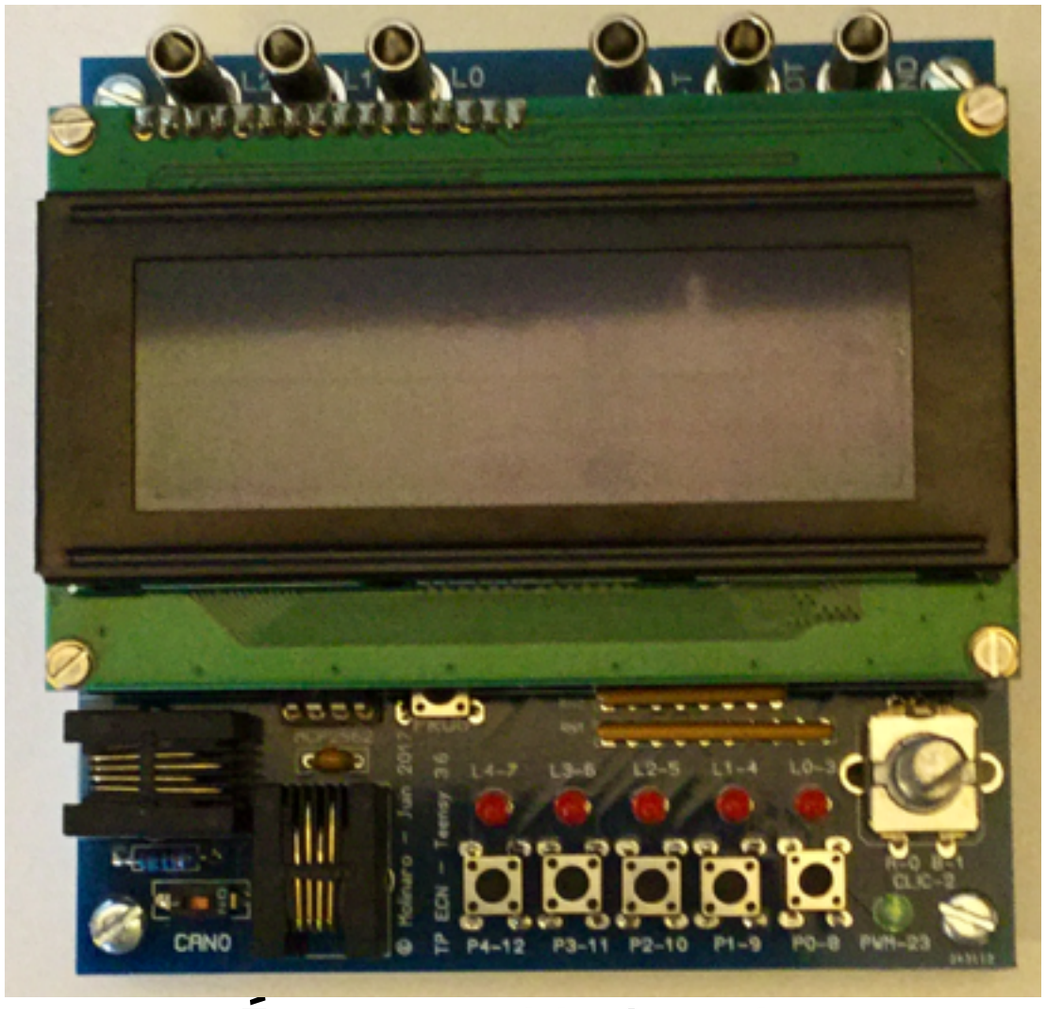
Temps Réel



Étape 09-critical-section

CENTRALE NANTES

Pierre Molinaro
Juin 2018

Description de cette partie

Malheureusement, le qualificatif **volatile**, s'il est nécessaire quand on déclare une variable partagée, n'est pas suffisant dans bien des situations.

Objectif :

- cette étape va illustrer un comportement pour lequel le qualificatif **volatile** n'est pas suffisant ;
- et va introduire la notion de section critique, et comment les spécifier dans les sources.

CENTRALE NANTES

Le programme d'exemple

Remplacer le fichier **setup-loop.cpp** par celui-ci (dans l'archive **09-files.tar.bz2**) :

*Les variables partagées sont bien déclarées avec le qualificatif **volatile**.*

```
#include "all-headers.

static volatile uint32_t gCount1 = 0 ;
static volatile uint32_t gCount2 = 0 ;
static volatile uint32_t gCount3 = 0 ;
static volatile uint32_t gCount4 = 0 ;
static volatile bool gPerformCount = true ;

static void rtISR (SECTION_MODE_
                   const uint32_t inUptime) {
  if (gPerformCount) {
    gCount1 += 1 ;
    gCount2 += 1 ;
    gCount3 += 1 ;
    gCount4 += 1 ;
  }
}

MACRO_REAL_TIME_ISR (rtISR) ;

void setup (USER_MODE) {
}
```

```
void loop (USER_MODE) {
  if (gPerformCount) {
    gCount1 += 1 ;
    gCount2 += 1 ;
    gCount3 += 1 ;
    gCount4 += 1 ;
    if (5000 <= millis ()) {
      gPerformCount = false ;
      gotoLineColumn (MODE_ 0, 0) ;
      printUnsigned (MODE_ gCount1) ;
      gotoLineColumn (MODE_ 1, 0) ;
      printUnsigned (MODE_ gCount2) ;
      gotoLineColumn (MODE_ 2, 0) ;
      printUnsigned (MODE_ gCount3) ;
      gotoLineColumn (MODE_ 3, 0) ;
      printUnsigned (MODE_ gCount4) ;
    }
  }
}
```

CENTRALE NANTES

Exécution du programme d'exemple

Exécutez le programme d'exemple, et examinez les valeurs affichées. Essayer plusieurs fois, en changeant la fréquence du processeur (paramètre **CPU-MHZ** dans le fichier **makefile.json**).

Voici un affichage obtenu (fréquence processeur : 180 MHz) :

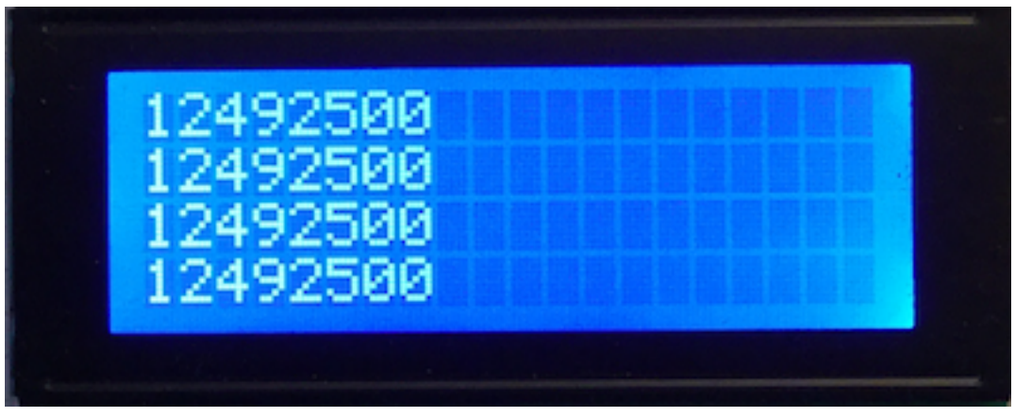
Pourquoi les valeurs sont-elles différentes ?

Le programme d'exemple

L'incrémentation sous interruption est commentée

```
#include "all-headers.h"

static volatile uint32_t gCount1 = 0 ;
static volatile uint32_t gCount2 = 0 ;
static volatile uint32_t gCount3 = 0 ;
static volatile uint32_t gCount4 = 0 ;
static volatile bool gPerformCount = true ;

static void rtISR (SECTION_MODE_
                   const uint32_t inUptime) {
  if (gPerformCount) {
//    gCount1 += 1 ;
//    gCount2 += 1 ;
//    gCount3 += 1 ;
//    gCount4 += 1 ;
  }
}

MACRO_REAL_TIME_ISR (rtISR) ;

void setup (USER_MODE) {
}
```

```
void loop (USER_MODE) {
  if (gPerformCount) {
    gCount1 += 1 ;
    gCount2 += 1 ;
    gCount3 += 1 ;
    gCount4 += 1 ;
    if (5000 <= millis ()) {
      gPerformCount = false ;
      gotoLineColumn (MODE_ 0, 0) ;
      printUnsigned (MODE_ gCount1) ;
      gotoLineColumn (MODE_ 1, 0) ;
      printUnsigned (MODE_ gCount2) ;
      gotoLineColumn (MODE_ 2, 0) ;
      printUnsigned (MODE_ gCount3) ;
      gotoLineColumn (MODE_ 3, 0) ;
      printUnsigned (MODE_ gCount4) ;
    }
  }
}
```



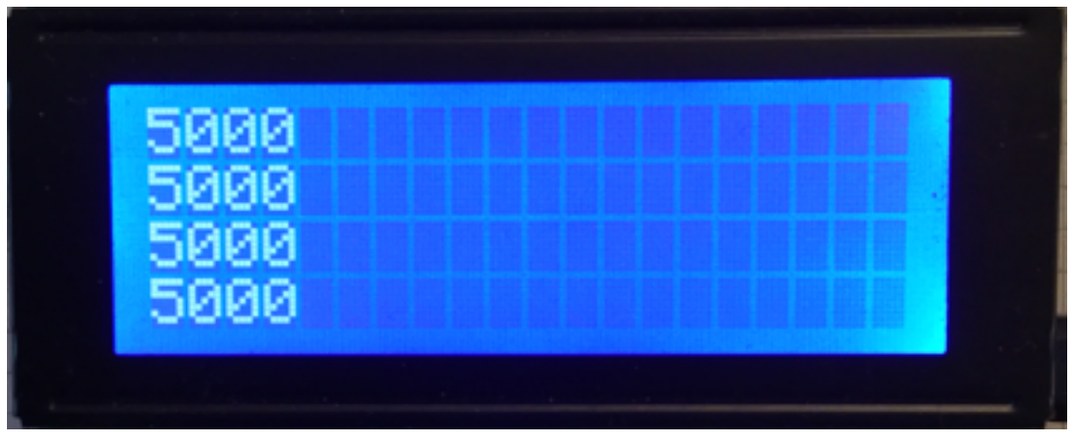
CENTRALE NANTES

Le programme d'exemple

L'incrémentation dans la tâche de fond est commentée

```
#include "all-headers.h"

static volatile uint32_t gCount1 = 0 ;
static volatile uint32_t gCount2 = 0 ;
static volatile uint32_t gCount3 = 0 ;
static volatile uint32_t gCount4 = 0 ;
static volatile bool gPerformCount = true ;

static void rtISR (SECTION_MODE_
                   const uint32_t inUptime) {
  if (gPerformCount) {
    gCount1 += 1 ;
    gCount2 += 1 ;
    gCount3 += 1 ;
    gCount4 += 1 ;
  }
}

MACRO_REAL_TIME_ISR (rtISR) ;

void setup (USER_MODE) {
}
```

```
void loop (USER_MODE) {
  if (gPerformCount) {
//    gCount1 += 1 ;
//    gCount2 += 1 ;
//    gCount3 += 1 ;
//    gCount4 += 1 ;
    if (5000 <= millis ()) {
      gPerformCount = false ;
      gotoLineColumn (MODE_ 0, 0) ;
      printUnsigned (MODE_ gCount1) ;
      gotoLineColumn (MODE_ 1, 0) ;
      printUnsigned (MODE_ gCount2) ;
      gotoLineColumn (MODE_ 2, 0) ;
      printUnsigned (MODE_ gCount3) ;
      gotoLineColumn (MODE_ 3, 0) ;
      printUnsigned (MODE_ gCount4) ;
    }
  }
}
```



CENTRALE NANTES

Pourquoi les valeurs sont-elles différentes ? (1/2)

Le processeur embarqué dans le micro-contrôleur est un processeur Cortex-M4. Or ce processeur ne peut pas effectuer l'incrémentation d'une variable en mémoire en une seule instruction :

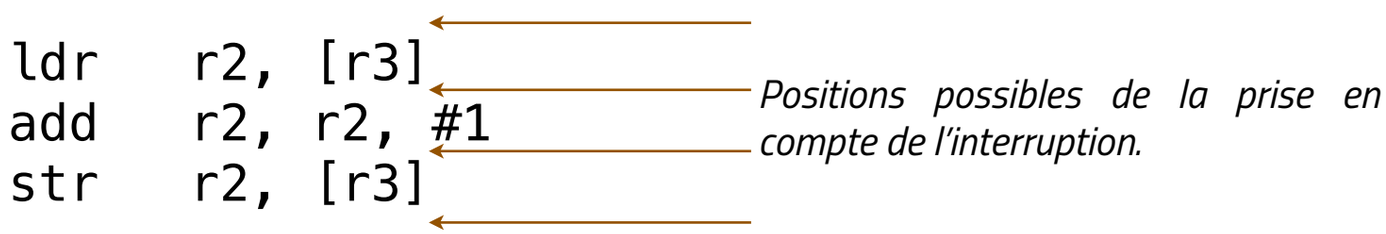
Code C

```
gCount1 += 1 ;
```

Code Assembleur ARM engendré

```
ldr r3, .....  @ R3 <- adresse de gCount1
ldr r2, [r3]   @ R2 <- valeur de Count1 (lecture mémoire)
add r2, r2, #1 @ R2 <- R2 + 1
str r2, [r3]   @ Écriture de R2 en mémoire
```

Le code assembleur ARM montre que l'incrémentation d'une variable est réalisée par plusieurs instructions successives. Une interruption ne peut pas interrompre l'exécution d'une instruction assembleur, mais elle peut interrompre la séquence d'exécution.



CENTRALE NANTES

Pourquoi les valeurs sont-elles différentes ? (2/2)

Pour comprendre l'origine du piège, on va abstraire le code assembleur relatif à l'incrémentation d'une variable :

Dans la routine `loop`

`y := gCount1`
`y ++`
`gCount1 := y`

Dans la routine `rtISR`

`x := gCount1`
`x ++`
`gCount1 := x`

Comme l'interruption peut être prise en compte à tout instant dans la routine **loop**, on peut avoir par exemple les scénarios suivants :

Un scénario

`y := gCount1`	
`y ++`	
`gCount1 := y`	
	`x := gCount1`
	`x ++`
	`gCount1 := x`

La variable est correctement incrémentée de 2 unités.

Un autre scénario

`y := gCount1`	
`y ++`	
	`x := gCount1`
	`x ++`
	`gCount1 := x`
`gCount1 := y`	

La variable est incrémentée d'une seule unité.

CENTRALE NANTES

Une possibilité de correction : `__atomic_fetch_add`

GCC pour Cortex-M4 possède des *intrinsics* qui engendre un code assurant que l'incrémentation se réalise de façon atomique.

```
void loop (USER_MODE) {
  if (gPerformCount) {
    __atomic_fetch_add (& gCount1, 1, __ATOMIC_ACQ_REL) ;
    __atomic_fetch_add (& gCount2, 1, __ATOMIC_ACQ_REL) ;
    __atomic_fetch_add (& gCount3, 1, __ATOMIC_ACQ_REL) ;
    __atomic_fetch_add (& gCount4, 1, __ATOMIC_ACQ_REL) ;
    if (5000 <= millis ()) {
      gPerformCount = false ;
      gotoLineColumn (MODE_ 0, 0) ;
      printUnsigned (MODE_ gCount1) ;
      gotoLineColumn (MODE_ 1, 0) ;
      printUnsigned (MODE_ gCount2) ;
      gotoLineColumn (MODE_ 2, 0) ;
      printUnsigned (MODE_ gCount3) ;
      gotoLineColumn (MODE_ 3, 0) ;
      printUnsigned (MODE_ gCount4) ;
    }
  }
}
```

Vérifier que les quatre compteurs ont alors toujours la même valeur.

Liens :

https://en.wikipedia.org/wiki/Intrinsic_function
https://gcc.gnu.org/onlinedocs/gcc/_005f_005fatomic-Builtins.html

CENTRALE NANTES

Seconde possibilité de correction : section critique

Une autre possibilité est rendre ininterruptible la séquence des incrémentations (ou chacune des incrémentations). Pour cela, on va définir des ***sections critiques***.

Le système de compilation permet de choisir un schéma de section critique parmi plusieurs possibles. Le choix s'effectue dans le fichier **makefile.json**, avec la clé **SECTION-SCHEME** :

- `"SECTION-SCHEME" : "disableInterrupt"` : les interruptions sont masquées dans la section critique [**C'EST LE SCHÉMA À UTILISER**] ;
- `"SECTION-SCHEME" : "swint"` : (*software interrupt*) ce schéma utilise l'interruption n°80 pour exécuter la section critique dans le mode *handler* du processeur ;
- `"SECTION-SCHEME" : "bkpt"` : (*breakpoint*) ce schéma utilise l'instruction `bkpt` pour exécuter la section critique dans le mode *handler* du processeur [**NE PAS UTILISER, NE FONCTIONNE PAS**].

CENTRALE NANTES

Comment spécifier une section critique

Pour spécifier une section critique, il faut :

- déclarer le prototype de la fonction appelée (en mode **`USER`**) ;
- implémenter la fonction qui exécute la section critique (en mode **`SECTION`**) ;
- déclarer la section critique au système de compilation par une annotation particulière ;
- quand on a une ou plusieurs sections critiques, préciser un schéma dans le fichier **makefile.json**, avec la clé **SECTION-SCHEME**.

CENTRALE NANTES

Comment spécifier une section critique : en pratique

La fonction qui exécute la section critique en mode `SECTION` (dans **setup-loop.cpp**) :

```
void section_incrementations (SECTION_MODE) {
  gCount1 += 1 ;
  gCount2 += 1 ;
  gCount3 += 1 ;
  gCount4 += 1 ;
}
```

Dans le fichier **setup-loop.h**, on déclare :

- le prototype de la fonction appelée (en mode **USER**) ;
- le prototype de la fonction qui exécute la section critique (en mode **SECTION**) ;
- la section critique au système de compilation par une annotation particulière.

```
//$section fonction.incrementations

void incrementations (USER_MODE) asm ("fonction.incrementations") ;

void section_incrementations (SECTION_MODE) asm ("section.fonction.incrementations") ;
```

Dans la fonction **loop** (fichier **setup-loop.cpp**) on appelle la fonction `incrementations` :

```
void loop (USER_MODE) {
  if (gPerformCount) {
    incrementations (MODE) ;
    if (5000 <= millis ()) {
      ..............
    }
  }
}
```

CENTRALE NANTES

Code engendré par l'annotation `//$section`

L'annotation `//$section` engendre le code dans **zSOURCES/interrupt-handlers.s**. Voici le code engendré quand le schéma choisi est `disableInterrupt` :

```
fonction.incrementations:
  .fnstart
@--- Save preserved registers
  push  {r6, lr}
@--- Save interrupt enabled state
  mrs   r6, PRIMASK
@--- Disable interrupt
  cpsid i
@--- Call section, interrupts disabled
  bl    section.fonction.incrementations
@--- Restore interrupt state
  msr   PRIMASK, r6
@--- Restore preserved registers and return
  pop   {r6, pc}
```

En examinant le code ci-dessus, il apparaît que la fonction **`fonction.incrementations`** peut être appelée interruptions masquées : l'état de masquage est sauvé dans R6, et restitué après l'appel de la section critique.

Premier travail à faire

Mettre en place la section critique pour incrémenter les variables, et vérifier que les quatre valeurs obtenues sont les mêmes.

Second travail à faire (1/2)

On va mettre en place un compteur 64 bits permettant d'obtenir une résolution de 1 µs. Ce compteur sera conservé dans toutes les étapes ultérieures. Les fichiers à modifier sont **time.h** et **time.cpp**.

Techniquement, ce compteur utilise deux compteurs 32 bits, qui sont chaînés pour obtenir une valeur 64 bits.

L'initialisation est réalisée par le code suivant (à insérer à la fin de **startSystick**) :

```
//------------------ Configure and chain PIT0 and PIT1 for 64-bit counting
//--- Power on PIT
  SIM_SCGC6 |= SIM_SCGC6_PIT ;
//--- Enable PIT module
  PIT_MCR = 0 ;
//--- Disable PIT0 and PIT1
  PIT_TCTRL (0) = 0 ;
  PIT_TCTRL (1) = 0 ;
//--- PIT0 and PIT1 down-count: initialize them with all 1's
  PIT_LDVAL (0) = UINT32_MAX ;
  PIT_LDVAL (1) = UINT32_MAX ;
//--- Enable PIT0 and PIT1: start counting, chain PI1 to PIT0, no interrupt
  PIT_TCTRL (1) = PIT_TCTRL_CHN | PIT_TCTRL_TEN ;
  PIT_TCTRL (0) = PIT_TCTRL_TEN ;
```

Second travail à faire (2/2)

Ensuite la lecture est effectuée par le code :

```
//--- To obtain the correct value, first read LTMR64H and then LTMR64L
  uint64_t result = PIT_LTMR64H ;
  result <<= 32 ;
  result |= PIT_LTMR64L ;
//--- PIT0 and PIT1 actually downcount
  result = ~ result ;
//--- Divide by the clock frequency in MHz for getting microsecond count
  result /= busMHZ () ;
```

Il n'y a pas de registre 64 bits, la valeur courante est détenue par les deux registres 32 bits `PIT_LTMR64H` et `PIT_LTMR64L`. La contrainte de lecture est que `PIT_LTMR64H` doit être lu en premier, et que cette lecture fige la valeur de `PIT_LTMR64L`. Donc, le code ci-dessus doit être exécuté en exclusion mutuelle.

Travail à faire : écrire une fonction micros, appelable en mode USER, qui retourne la valeur 64 bits des deux compteurs via une section critique.