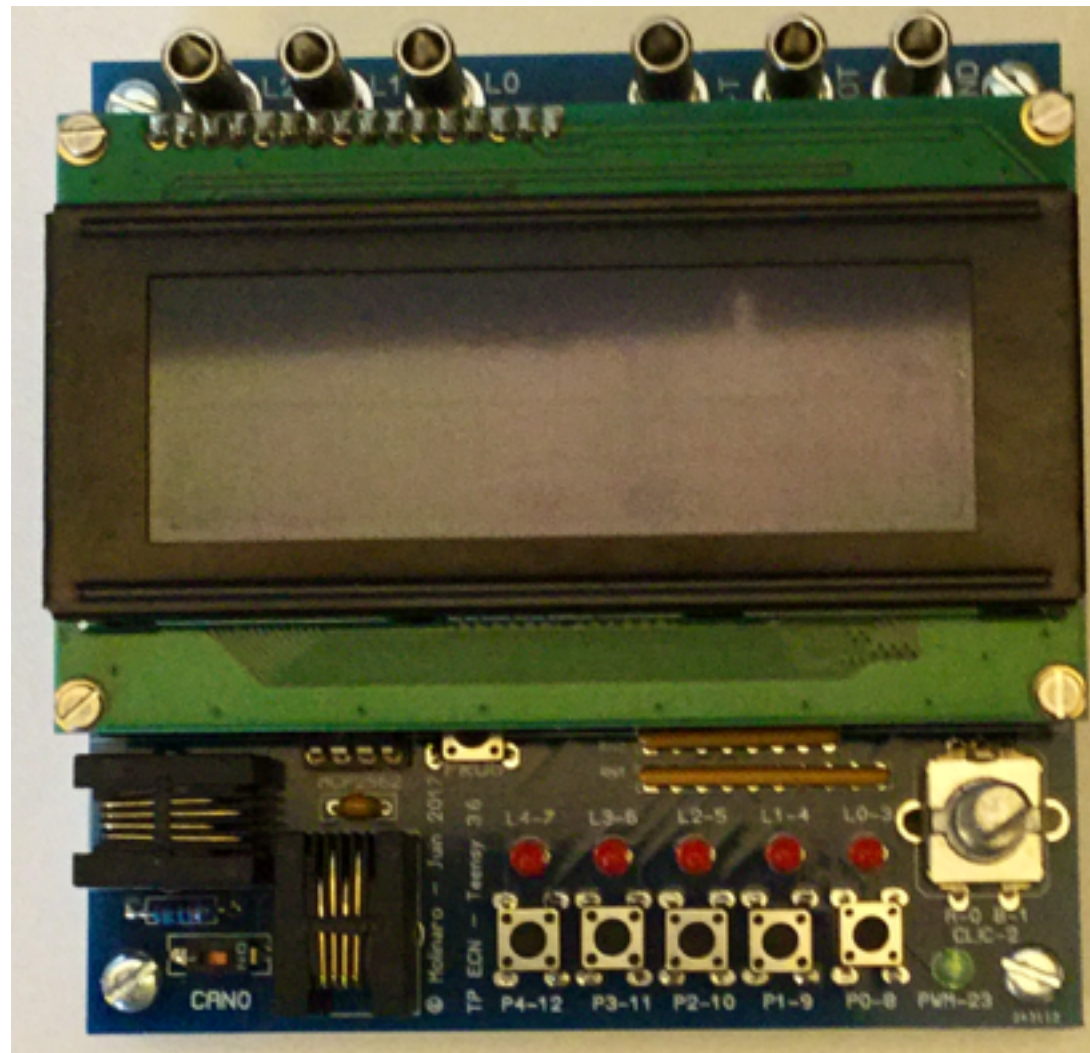


Temps Réel



Étape 08-volatile--systick-isr

Description de cette partie

Objectif : cette étape a un double objectif :

- comment inscrire des fonctions pour qu'elles soient exécutées à chaque occurrence de l'interruption SysTick ;
- montrer pourquoi les variables partagées doivent être déclarées avec le qualificatif **volatile**.

Exécution d'une fonction par l'interruption SysTick (1/4)

La routine exécutée lors du déclenchement de l'interruption SysTick a été écrite lors de l'étape précédente (fichier **time.cpp**) :

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

Dans cette étape, nous voulons pouvoir disposer d'un mécanisme similaire à `MACRO_BOOT_ROUTINE` et `MACRO_INIT_ROUTINE`, de façon que l'on puisse facilement inscrire une fonction pour qu'elle soit exécutée à chaque occurrence de l'interruption SysTick.

Exécution d'une fonction par l'interruption SysTick (2/4)

Le script de l'édition des liens **dev-files/teensy-3-6.ld** rassemble toutes les sections **real.time.interrupt.routine.array**, à l'image de ce qui a été fait pour les sections **boot.routine.array** et **init.routine.array** :

```
SECTIONS {
  .text : {
    .....
    . = ALIGN (4) ;
    __real_time_interrupt_routine_array_start = . ;
    KEEP (*(real.time.interrupt.routine.array)) ;
    . = ALIGN (4) ;
    __real_time_interrupt_routine_array_end = . ;
    .....
  } > flash
}
```

Dans cette étape, nous voulons pouvoir disposer d'un mécanisme similaire à **MACRO_BOOT_ROUTINE** et **MACRO_INIT_ROUTINE**, de façon que l'on puisse facilement inscrire une fonction pour qu'elle soit exécutée à chaque occurrence de l'interruption SysTick.

Exécution d'une fonction par l'interruption SysTick (3/4)

Pour exploiter les sections **real.time.interrupt.routine.array**, on ajoute dans le fichier time.h la définition de la macro **MACRO_REAL_TIME_ISR**, similaire à **MACRO_BOOT_ROUTINE** et **MACRO_INIT_ROUTINE** :

```
#define MACRO_REAL_TIME_ISR(ROUTINE) \
    static void (* UNIQUE_IDENTIFIER) (SECTION_MODE_ const uint32_t inUptime) \
    __attribute__((section ("real.time.interrupt.routine.array"))) \
    __attribute__((unused)) \
    __attribute__((used)) = ROUTINE ;
```

Et il faut ajouter à la fonction **systickInterruptServiceRoutine** le parcours du tableau débutant à **__real_time_interrupt_routine_array_start** et l'exécution des fonctions qu'il contient :

```
void systickInterruptServiceRoutine (SECTION_MODE) {
    const uint32_t newUptime = gUptime + 1 ;
    gUptime = newUptime ;
    //--- Run real.time.interrupt.routine.array section routines
    extern void (* __real_time_interrupt_routine_array_start) (SECTION_MODE_ const uint32_t inUptime) ;
    extern void (* __real_time_interrupt_routine_array_end) (SECTION_MODE_ const uint32_t inUptime) ;
    void (* * ptr) (SECTION_MODE_ const uint32_t) = & __real_time_interrupt_routine_array_start ;
    while (ptr != & __real_time_interrupt_routine_array_end) {
        (* ptr) (MODE_ newUptime) ;
        ptr ++ ;
    }
}
```

Exécution d'une fonction par l'interruption SysTick (4/4)

Voici un exemple d'utilisation de la macro `MACRO_REAL_TIME_ISR` :

```
static void rtISR (SECTION_MODE_ const uint32_t inUptime) {  
    .....  
}  
  
MACRO_REAL_TIME_ISR (rtISR) ;
```

Remarquer que la fonction présente l'argument `inUptime`, qui contient la valeur de l'instant courant.

Règle d'utilisation du qualificatif volatile

Toute variable globale partagée en une routine d'interruption et une routine de la tâche de fond (fonctions dans le mode USER) doit être déclarée volatile.

Si cette règle n'est pas respectée, le programme pourra peut-être fonctionner correctement (ou pas).

Dans la suite de cette étape, on propose un exemple de programme qui illustre cette règle.

Le programme d'exemple (1/3)

Remplacer le fichier **setup-loop.cpp** par celui-ci (dans l'archive **08-files.tar.bz2**) :

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

static uint32_t gCount ; // Volontairement, volatile est absent

static void rtISR (SECTION_MODE_ const uint32_t inUptime) {
    gCount += 1 ;
}

MACRO_REAL_TIME_ISR (rtISR) ;

void setup (USER_MODE) {
    printString (MODE_ "Hello!") ;
}

void loop (USER_MODE) {
    gCount += 500 ;
    gotoLineColumn (MODE_ 1, 0) ;
    printUnsigned (MODE_ gCount) ;
    busyWaitDuring (MODE_ 500) ;
}
```

Volontairement, le programme ci-dessus est bogué : la variable globale **gCount** est partagée entre la routine d'interruption **rtISR** et la routine de fond **loop**, et n'a pas été déclarée **volatile**.

Exécutez le programme, et constatez qu'il fonctionne correctement, malgré l'oubli de **volatile**.

Le programme d'exemple (2/3)

Modifier maintenant la fonction **setup** en ajoutant la ligne écrite en bleu :

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

static uint32_t gCount ; // Volontairement, volatile est absent

static void rtISR (SECTION_MODE_ const uint32_t inUptime) {
    gCount += 1 ;
}

MACRO_REAL_TIME_ISR (rtISR) ;

void setup (USER_MODE) {
    printString (MODE_ "Hello!") ;
    while (gCount < 3000) {}
}

void loop (USER_MODE) {
    gCount += 500 ;
    gotoLineColumn (MODE_ 1, 0) ;
    printUnsigned (MODE_ gCount) ;
    busyWaitDuring (MODE_ 500) ;
}
```

Exécutez le programme, et constatez qu'il ne fonctionne plus correctement : en fait, l'exécution est bloquée sur la ligne qui a été ajoutée.

Le programme d'exemple (3/3)

Ajouter le qualificatif **volatile** à la déclaration de **gCount** :

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

static volatile uint32_t gCount ; // Correct !

static void rtISR (SECTION_MODE_ const uint32_t inUptime) {
    gCount += 1 ;
}

MACRO_REAL_TIME_ISR (rtISR) ;

void setup (USER_MODE) {
    printString (MODE_ "Hello!") ;
    while (gCount < 3000) {}
}

void loop (USER_MODE) {
    gCount += 500 ;
    gotoLineColumn (MODE_ 1, 0) ;
    printUnsigned (MODE_ gCount) ;
    busyWaitDuring (MODE_ 500) ;
}
```

Exécutez le programme, et constatez qu'il fonctionne correctement.

Discussion : code engendré en présence de volatile

Le cœur du problème est le code engendré par la compilation de la fonction **setup**.

```
static volatile uint32_t gCount ; // Correct !  
.....  
void setup (USER_MODE) {  
    printString (MODE_ "Hello!") ;  
    while (gCount < 3000) {}  
}
```

Le code engendré est (appeler **1-build-as.py** pour l'obtenir), les commentaires en vert ont été ajoutés :

```
setup.function:  
    push{r3, lr} @ Sauvegarde de R3 et de LR dans la pile  
    ldr r0, .L7 @ R0 ← adresse de la chaîne "Hello!"  
    bl _Z11printStringPKc @ Appel de la fonction printString  
    ldr r1, .L7+4 @ R1 ← Adresse de gCount  
    movw r3, #2999 @ R3 ← 2999  
.L5:  
    ldr r2, [r1] @ R2 ← valeur de gCount (lecture mémoire)  
    cmp r2, r3 @ Comparaison entre R2 et R3  
    bls .L5 @ Branch unsigned Lower or Same : if (r2 <= r3) goto .L5  
    pop {r3, pc} @ Restauration de R3 et retour
```

Discussion : code engendré sans volatile

Maintenant, en enlevant le qualificatif **volatile** :

```
static uint32_t gCount ; // Bug
.....
void setup (USER_MODE) {
    printString (MODE_ "Hello!") ;
    while (gCount < 3000) {}
}
```

Le code engendré est (appeler **1-build-as.py** pour l'obtenir), les commentaires en vert ont été ajoutés :

```
setup.function:
    push{r3, lr}  @ Sauvegarde de R3 et de LR dans la pile
    ldr r0, .L7      @ R0 <- adresse de la chaîne "Hello!"
    bl  _Z11printStringPKc @ Appel de la fonction printString
    ldr r3, .L7+4 @ R3 <- Adresse de gCount
    ldr r2, [r3]  @ R2 <- valeur de gCount (lecture mémoire)
    movw r3, #2999 @ R3 <- 2999
.L5:
    cmp r2, r3      @ Comparaison entre R2 et R3
    bls .L5         @ Branch unsigned Lower or Same : if (r2 <= r3) goto .L5
    pop {r3, pc}    @ Restauration de R3 et retour
```

Dans la boucle **.L5**, la valeur de **gCount** n'est pas rechargée par une lecture mémoire : la variable est lue une fois avant la boucle, et si le test est vrai le bouclage est infini.

La nécessité de volatile

Toute variable globale partagée en une routine d'interruption et une routine de la tâche de fond (fonctions dans le mode USER) doit être déclarée volatile.

Voir le document suivant (page 42) :

https://users.ece.cmu.edu/~koopman/pubs/koopman14_toyota_ua_slides.pdf