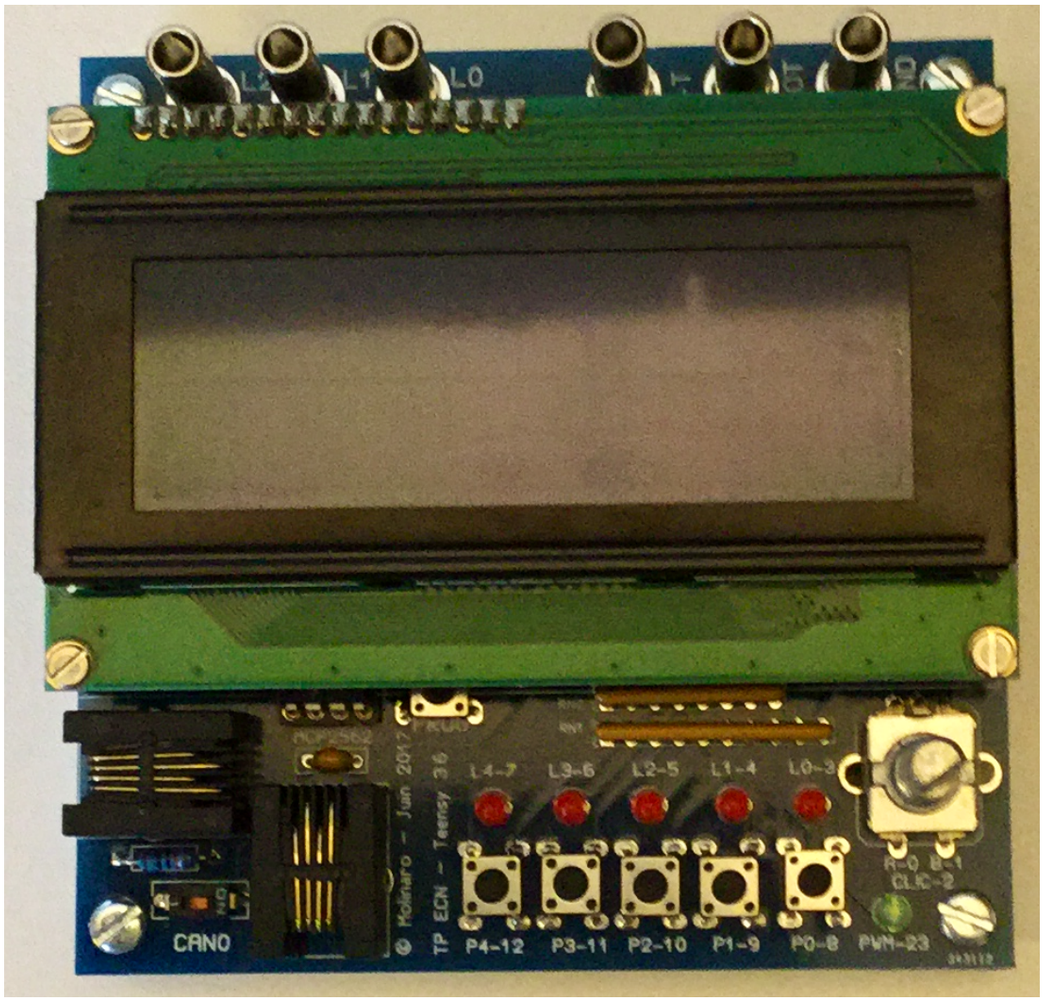
Temps Réel



Programme *13-task-termination*

Pierre Molinaro
Janvier 2019

Description de cette étape

Dans cette étape, on ajoute la terminaison des tâches. Dans l'étape précédente, la terminaison d'une tâche provoquait un plantage.

La terminaison d'une tâche est effectuée via un *service de l'exécutif*, c'est donc l'occasion d'exposer comment un service est écrit.

Le code que vous avez à écrire est le suivant :

- déclarer et implémenter plusieurs tâches ;
- chaque tâche fait clignoter une dizaine fois une led, puis se termine.

Le comportement attendu est : la tâche la plus prioritaire fait clignoter sa led et se termine, permettant l'exécution de la tâche suivante, … jusqu'à ce que la dernière tâche se termine. Il n'y a alors que la tâche de fond qui s'exécute, la led Teensy est quasiment éteinte.

Comment est réalisée la terminaison d'une tâche

On va écrire un *service de l'exécutif*, qui met en œuvre la terminaison de la tâche qui l'appelle. Ce sera l'occasion de montrer comment un service de l'exécutif doit être écrit.

Ensuite, on modifiera la construction du contexte initial d'une tâche, de façon que ce service soit automatiquement appelé.

Écrire un service de l'exécutif

Écrire un *service de l'exécutif* s'effectue en quatre opérations.

① Dans un fichier d'en-tête (ici, **xtr.h**), déclarer le prototype du service appelé en mode **USER** :

```
void taskSelfTerminates (USER_MODE) asm ("task.self.terminates") ;
```

② Dans le même fichier d'en-tête, déclarer le prototype de la fonction implémentant le service :

```
void service_taskSelfTerminates (KERNEL_MODE) asm ("service.task.self.terminates") ;
```

Cette fonction s'exécute en mode **KERNEL**. Son nom C++ (**service_taskSelfTerminates**) est libre, son nom assembleur doit être le nom assembleur de la fonction correspondante appelée en mode **USER**, précédé par `service.`.

③ Déclarer le service. Rappelons (étape 01) que tous les fichiers d'en-tête sont examinés par un script Python afin d'analyser différentes déclarations ; l'annotation `//$service` permet de déclarer un service de l'exécutif.

```
//$service task.self.terminates
```

④ Implémenter le service. Dans un fichier C++ (ici **xtr.cpp**), écrire la fonction :

```
void service_taskSelfTerminates (KERNEL_MODE) {
  ····· // Voir le contenu pages suivantes
}
```

CENTRALE NANTES

Où est écrit `taskSelfTerminates` ?

Dans les quatre opérations de la page précédente, la fonction **taskSelfTerminates** déclarée en ① n'est pas implémentée en C++. La déclaration du service en ④ permet aux scripts Python d'ajouter sa prise en charge dans le fichier **zSOURCES/interrupt-handlers.s** (c'est le nom associé à l'annotation `//$service` qui apparaît) :

```
task.self.terminates:
  svc #1
  bx  lr
```

Le *svc handler* est donc appelé avec un argument égal à 1. Dans le même fichier assembleur, regardez la table des services :

```
svc.dispatcher.table:
  .word start.phase2 @ 0
  .word service.task.self.terminates @ 1
```

Lorsque l'instruction `svc  #1` est exécutée, le *svc handler* appelle le service qui est à l'indice 1 dans la table des services.

Note : dans les étapes suivantes, des services seront ajoutés. L'ordre de numérotation des services en fonction de l'ordre d'analyse des fichiers d'en-tête par les scripts Python : le service de terminaison de tâche pourra se voir attribuer un autre indice.

Écrire le service de terminaison de tâche

Plus précisément, ce service va terminer la tâche qui l'appelle.

L'écriture est simple :

```
void service_taskSelfTerminates (KERNEL_MODE) {
  kernel_makeNoTaskRunning (MODE) ;
}
```

La fonction **kernel_makeNoTaskRunning** n'est pas présente dans le fichier **xtr.cpp**, il faut l'ajouter :

```
static void kernel_makeNoTaskRunning (KERNEL_MODE) {
  gRunningTaskControlBlockPtr = nullptr ; // No running task
}
```

Pourquoi mettre **gRunningTaskControlBlockPtr** à **nullptr** suffit ? En fait, appeler **service_taskSelfTerminates** invoque le *svc handler* (voir page suivante). Le *svc handler* (voir son algorithme dans le PDF de l'étape12) appelle d'abord la fonction d'implémentation du service (ici **kernel_makeNoTaskRunning**), puis la fonction **kernel.select.task.to.run**. Celle-ci — aussi décrite dans le PDF de l'étape12 — voyant **gRunningTaskControlBlockPtr** à **nullptr** retire une tâche de la liste des tâches prêtes et la rend en cours.

Appeler le service de terminaison de tâche (1/2)

Maintenant, le service de terminaison de tâche existe, mais n'est pas appelé. Il y a deux façons de le faire.

La première est d'écrire l'appel du service de terminaison comme dernière instruction de chaque fonction implémentant le code d'une tâche. Par exemple :

```
static void task1 (USER_MODE) {
  ......
  taskSelfTerminates (MODE) ;
}
```

C'est une écriture que nous ne retiendrons pas, car elle est soumise au bon vouloir du programmeur : un oubli, et c'est le plantage.

Appeler le service de terminaison de tâche (2/2)

La seconde façon est d'inclure dans l'exécutif l'exécution automatique du service de terminaison de tâche. Pour cela, on s'appuie sur une caractéristique de l'*ABI* des processeurs ARM : l'adresse de retour d'un sous-programme est le contenu du registre **LR** du processeur lors de l'entrée dans ce sous-programme.

Autrement dit : lors de l'établissement du contexte initial d'une tâche, il faut mettre dans la sauvegarde du registre **LR** l'adresse de la fonction **taskSelfTerminates**. Pour cela, éditez la fonction **kernel_set_task_context** du fichier **xtr.cpp** et ajouter l'affectation au champ **mLR** :

```
static void kernel_set_task_context (INIT_MODE_
                                     TaskContext & ioTaskContext,
                                     const uint32_t inStackBufferAddress,
                                     const uint32_t inStackBufferSize,
                                     RoutineTaskType inTaskRoutine) {
 ............
//--- Self termination
  ptr->mLR = (uint32_t) taskSelfTerminates ;
}
```

Note : une ABI (*Application Binary Interface*) définit (entre autres) les conventions d'appel des fonctions.

Liens :

https://fr.wikipedia.org/wiki/Application_binary_interface

http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042f/IHI0042F_aapcs.pdf

CENTRALE NANTES

Pour les curieux : le retour de sous-programme (1/3)

Quand un sous-programme est appelé, l'adresse de retour est placée dans le registre **LR** (*Link Register*, ou **R14**). Le jeu d'instructions du Cortex-M4 ne définit pas d'instruction particulière de retour de sous-programme, le retour est simplement un branchement à l'adresse contenue dans **LR**.

On peut trouver un exemple illustratif en regardant comment la fonction **configureFaultRegisters** (fichier **fault-handlers--assertion.cpp**) est compilée :

```
static void configureFaultRegisters (BOOT_MODE) {
 SCB_CCR |= (1 << 4) | (1 << 3) ;
}
```

Le code assembleur engendré est dans **zASBUILDS/fault-handlers--assertion.cpp.s.list** :

```
  19                _ZL23configureFaultRegistersv:
  23 0000 024A          ldr r2, .L2
  24 0002 1368          ldr r3, [r2]
  25 0004 43F01803      orr r3, r3, #24
  26 0008 1360          str r3, [r2]
  27 000a 7047          bx  lr
```

CENTRALE NANTES

Pour les curieux : le retour de sous-programme (2/3)

À titre d'information, il y a des variantes qui sont adoptées pour optimiser (la vitesse, la taille) du code.

Par exemple, si le sous-programme appelle lui-même un autre sous-programme, il faut évidemment sauvegarder **LR**. Par exemple, la fonction **printSpaces** (fichier **lcd.cpp**) :

```
void printSpaces (USER_MODE_ const uint32_t inCount) {
  uint32_t count = inCount ;
  while (count > 0) {
    printChar (MODE_ ' ') ;
    count -- ;
  }
}
```

Le code assembleur engendré est dans **zASBUILDS/lcd.cpp.s.list** :

```
454                 _Z11printSpacesm:
457 0000 10B5           push   {r4, lr}
458 0002 0446           mov r4, r0
459                 .L33:
460 0004 24B1           cbz r4, .L31
461 0006 2020           movs   r0, #32
462 0008 FFF7FEFF       bl _ZL9writeDatah
463 000c 013C           subs   r4, r4, #1
464 000e F9E7           b  .L33
465                 .L31:
466 0010 10BD           pop {r4, pc}
```

`pop {r4, pc}` est fonctionnellement équivalent à la séquence `pop {r4, lr} ; bx lr`.

Pour les curieux : le retour de sous-programme (3/3)

Une autre optimisation peut être faite par le compilateur quand la dernière instruction de la fonction C ou C++ est l'appel d'une autre fonction : au lieu d'effectuer un appel de sous-programme, un simple branchement est effectué. Par exemple, la fonction **printHex2** (fichier **lcd.cpp**) :

```
void printHex2 (USER_MODE_ const uint32_t inValue) {
  printHex1 (MODE_ inValue >> 4) ;
  printHex1 (MODE_ inValue) ;
}
```

Le code assembleur engendré est dans **zASBUILDS/lcd.cpp.s.list** :

```
619                 _Z9printHex2m:
622 0000 10B5           push   {r4, lr}
623 0002 0446           mov r4, r0
624 0004 0009           lsrs   r0, r0, #4
625 0006 FFF7FEFF       bl _Z9printHex1m
626 000a 2046           mov r0, r4
627 000c BDE81040       pop {r4, lr}
628 0010 FFF7FEBF       b  _Z9printHex1m
```

CENTRALE NANTES