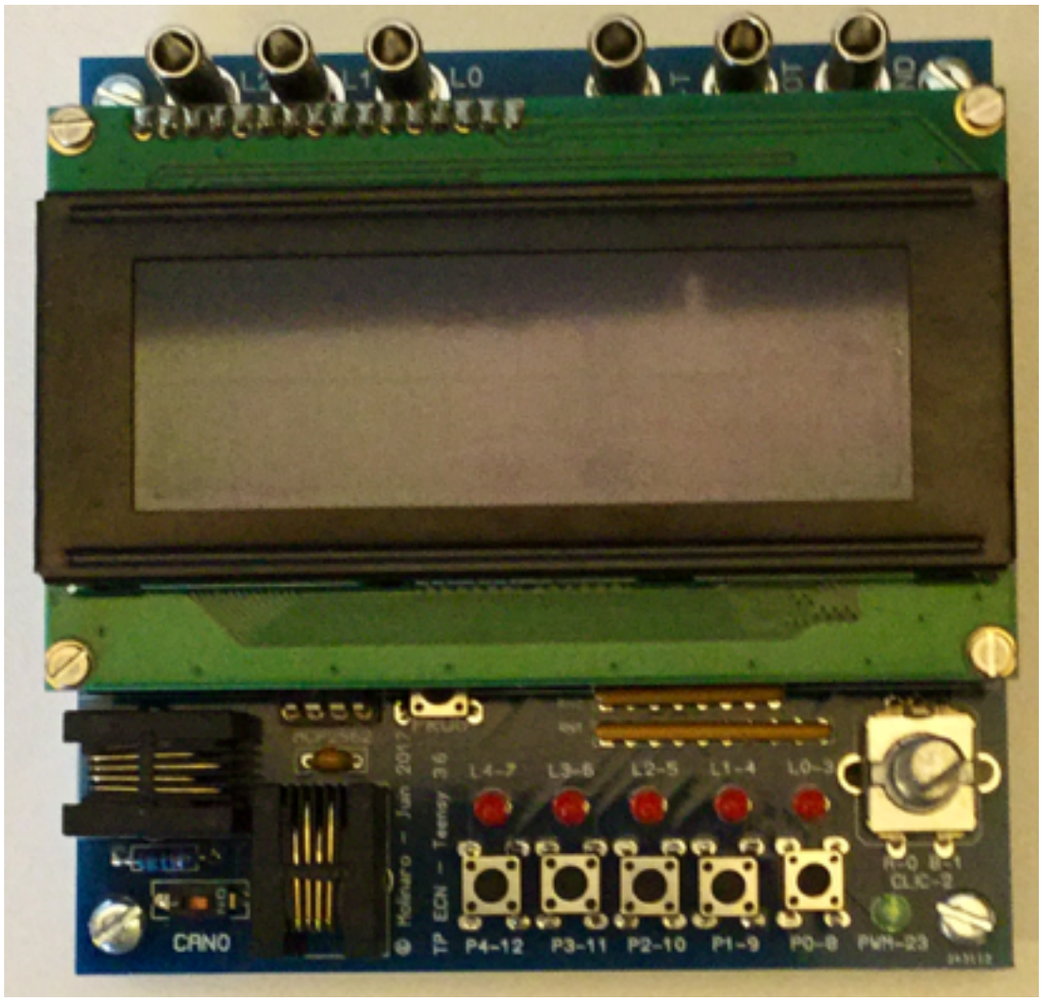
Temps Réel



Programme *14-wait*

Pierre Molinaro

Juin 2018

Description de cette étape

Dans cette étape, on ajoute l'attente temporelle passive. Jusqu'à l'étape précédente, les fonctions **busyWaitDuring** et **busyWaitUntil** effectuent des attentes actives, c'est-à-dire qu'elles accaparent le processeur durant l'attente — d'ailleurs la led Teensy qui visualise l'occupation processeur est active.

Refaites tourner le programme de l'étape précédente : les tâches s'exécutent les unes après les autres, dans l'ordre décroissant de leur priorité.

Le code que vous avez à écrire est d'implémenter l'attente passive, les fonctions **busyWaitDuring** et **busyWaitUntil** étant renommées par conséquent **waitDuring** et **waitUntil**.
Attention, l'exclusion mutuelle de l'accès à l'afficheur LCD n'est pas (encore) assurée : votre code ne doit pas appeler les fonctions de l'afficheur LCD en concurrence.

Les fonctions **busyWaitDuring_initMode** et **busyWaitDuring_faultMode** sont inchangées.

Dupliquer le projet de l'étape précédente et renommez-le par exemple **14-wait**.

Écriture de la fonction `waitDuring`

Dans les fichiers **time.h** et **time.cpp**, supprimez la fonction **busyWaitDuring** et remplacez-la par la fonction **waitDuring** qui appelle simplement la fonction **waitUntil** :

```
void waitDuring (USER_MODE_ const uint32_t inDelayMS) {
  waitUntil (MODE_ gUptime + inDelayMS) ;
}
```

Il vous faut aussi remplacer toutes les occurrences de l'appel à la fonction **busyWaitDuring** et par l'appel de **waitDuring**, notamment dans **lcd.cpp**.

Écriture de la fonction `waitUntil`

La fonction **waitUntil** est appelée en mode **USER**, or le blocage d'une tâche ne peut être effectué que par des routines en mode **KERNEL**. Il faut donc créer un service, comme cela a été fait à l'étape précédente avec **taskSelfTerminates**.

Déclarez dans le fichier **time.h** le service ; pour uniformiser, adoptez les prototypes suivants :

```
void waitUntil (USER_MODE_ const uint32_t inDeadlineMS) asm ··· ;
void kernel_waitUntil (KERNEL_MODE_ const uint32_t inDeadlineMS) asm ··· ;
```

Écrire l'implémentation du service dans le fichier **time.cpp** (supprimez la fonction **busyWaitUntil**) :

```
void kernel_waitUntil (KERNEL_MODE_ const uint32_t inDeadlineMS) {
  if (inDeadlineMS > gUptime) {
    kernel_blockOnDeadline (MODE_ inDeadlineMS) ;
  }
}
```

Le fonctionnement est simple : si l'échéance **inDeadlineMS** n'est pas atteinte, la tâche appelante est bloquée.

Il reste à écrire la fonction **kernel_blockOnDeadline** qui va bloquer une tâche en attente de l'échéance, et aussi à modifier la routine d'interruption temps-réel de façon qu'elle libère les tâches dont l'échéance est atteinte.

CENTRALE NANTES

Écriture de la fonction `kernel_blockOnDeadline`

La fonction **kernel_blockOnDeadline** est simple à écrire : elle insère la tâche en cours dans la liste des tâches bloquées sur échéance, écrit l'échéance dans le descripteur de tâche, puis bloque la tâche en cours.

Déclarez dans le fichier **time.cpp** la liste des tâches bloquées sur échéance :

```
static TaskList gDeadlineWaitingTaskList ;
```

Écrire ensuite l'implémentation de **kernel_blockOnDeadline** dans le fichier **time.cpp** :

```
void kernel_blockOnDeadline (KERNEL_MODE_ const uint32_t inDeadline) {
  XTR_ASSERT_NON_NULL_POINTER (gRunningTaskControlBlockPtr) ;
//--- Insert in deadline list
  gRunningTaskControlBlockPtr->mDeadline = inDeadline ;
  gDeadlineWaitingTaskList.enterTask (MODE_ gRunningTaskControlBlockPtr) ;
//--- Block task
  kernel_makeNoTaskRunning (MODE) ;
}
```

XTR_ASSERT_NON_NULL_POINTER est une macro qui fait appel à une assertion (voir le début du fichier **time.cpp**).

Il reste à modifier la routine d'interruption temps-réel.

Modifier la routine d'interruption temps-réel (1/2)

Celle-ci est déclarée dans **time.h**. Depuis l'étape 07 où elle a été ajoutée, elle est déclarée comme suit :

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

Elle s'exécute en mode **SECTION**. Or dans ce mode, les routines de l'exécutif ne peuvent pas être appelées : le mode **SECTION** est réservé aux routines extérieures à l'exécutif.

Ici, nous voulons appeler des routines de l'exécutif qui permettent de débloquer des tâches. Ce n'est pas exactement le mode **KERNEL**, car celui-ci permet aussi de bloquer la tâche en cours (ce qui n'aurait aucun sens dans une routine d'interruption). C'est le mode **IRQ** qui est réservé aux routines d'interruption qui peuvent débloquer des tâches. Il faut aussi changer l'annotation en `//$interrupt-service` , car il faut prendre en charge un éventuel changement de contexte. Les changements à effectuer sont en bleu :

```
//$interrupt-service SysTick
void systickInterruptServiceRoutine (IRQ_MODE) asm ("interrupt.service.SysTick") ;
```

CENTRALE NANTES

Modifier la routine d'interruption temps-réel (2/2)

Mais ce n'est pas tout pour cette fonction. En effet, elle appelle les fonctions dont l'adresse est placée dans la section r**eal.time.interrupt.routine.array** (voir étape 08). Pour faciliter l'inscription d'une routine à cette section, la macro suivante a été déclarée dans le fichier **time.h** :

```
#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 ;
```

C'est-à-dire que les routines doivent être déclarées pour être exécutées en mode **SECTION**. Or, maintenant, elle doivent s'exécuter en mode **IRQ** :

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

CENTRALE NANTES

Libérer les tâches dont l'échéance est atteinte (1/2)

On va donc inscrire une routine dans la section r**eal.time.interrupt.routine.array** ; ainsi, elle est exécutée à chaque occurrence de l'interruption temps-réel :

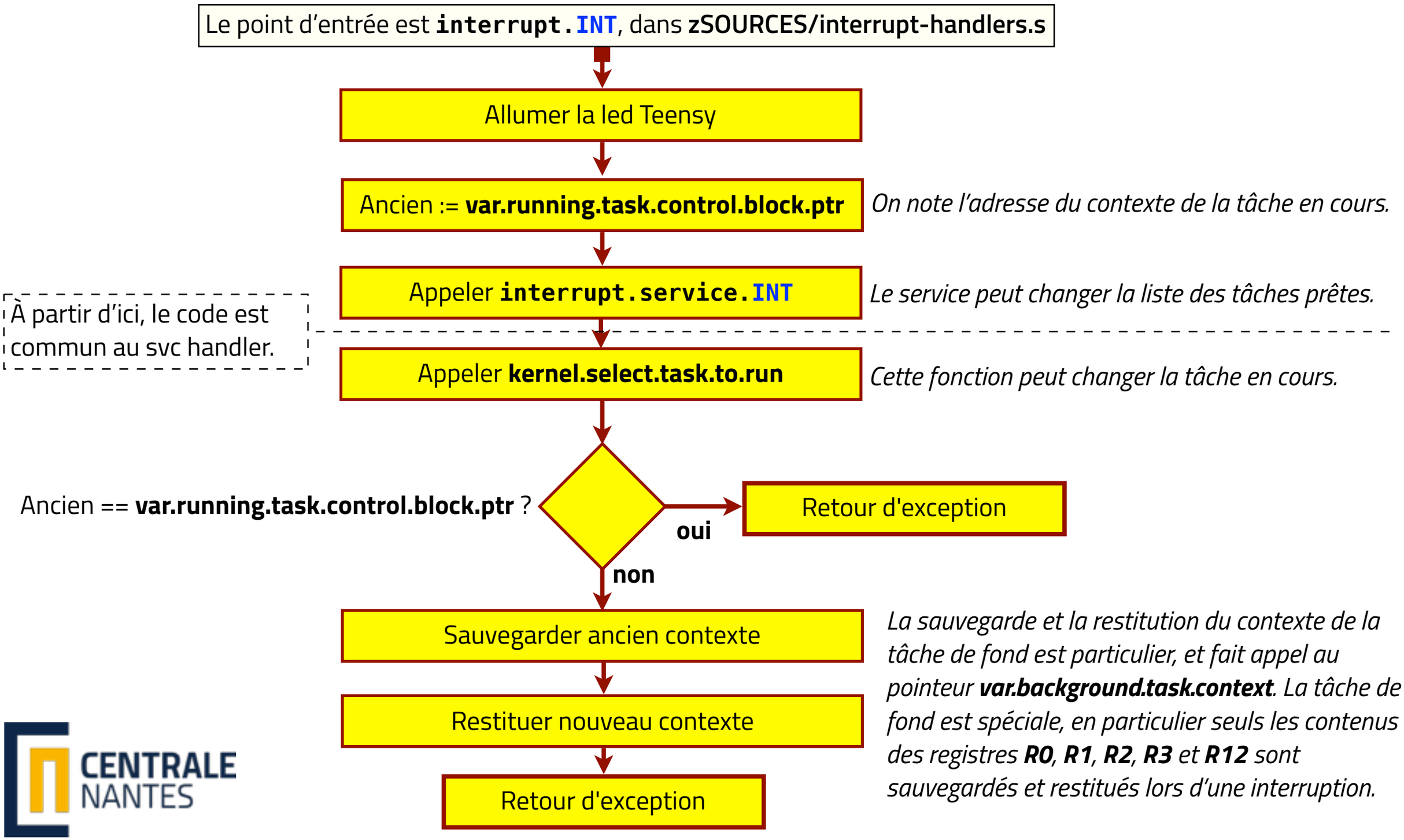
```
static void irq_makeTasksReadyFromCurrentDate (IRQ_MODE_ const uint32_t inCurrentDate) {
  ........
}

MACRO_REAL_TIME_ISR (irq_makeTasksReadyFromCurrentDate) ;
```

Que faut-il faire dans cette fonction ? Parcourir la liste des tâches bloquées, pour libérer celles dont l'échéance est atteinte.

Organigramme d'une routine d'interruption, mode IRQ

Cette routine est déclarée dans un fichier d'en-tête par :

```
//$interrupt-service INT
void routine (IRQ_MODE) asm ("interrupt.service.INT") ;
```



CENTRALE NANTES

Libérer les tâches dont l'échéance est atteinte (2/2)

On utilise la classe **`TaskList::Iterator`** qui implémente un itérateur de liste de tâches :

- le constructeur initialise l'itérateur ;
- la méthode **nextTask** renvoie :
  - **nullptr** si on est arrivé à la fin de la liste ;
  - le pointeur du descripteur courant, et avance au descripteur suivant.

La fonction **irq_makeTasksReadyFromCurrentDate** est :

```
static void irq_makeTasksReadyFromCurrentDate (IRQ_MODE_ const uint32_t inCurrentDate) {
  TaskList::Iterator iterator (MODE_ gDeadlineWaitingTaskList) ;
  TaskControlBlock * task ;
  while ((task = iterator.nextTask (MODE))) {
    if (inCurrentDate >= task->mDeadline) {
    //--- Remove task from deadline list
      gDeadlineWaitingTaskList.removeTask (MODE_ task) ;
   //--- Make task ready
      kernel_makeTaskReady (MODE_ task) ;
    }
  }
}
```

CENTRALE NANTES

La classe itérateur de liste `TaskList::Iterator` (1/2)

La classe **TaskList** est déclarée dans **task-list--32-tasks.h**, on y rajoute la déclaration de l'itérateur :

```
class TaskList {
  ...........
//-------------------------------------- Iterator
  public: class Iterator {
    public: inline Iterator (IRQ_MODE_ const TaskList & inTaskList) :
    mIteratedList (inTaskList.mList) {
    }

    public: TaskControlBlock * nextTask (IRQ_MODE) ;

  //--- Private property
    private: uint32_t mIteratedList ;

  //--- No copy
    private: Iterator (const Iterator &) ;
    private: Iterator & operator = (const Iterator &) ;
  } ;
} ;
```

CENTRALE NANTES

La classe itérateur de liste `TaskList::Iterator` (2/2)

Insérez l'implémentation de la méthode `TaskList::Iterator::nextTask` dans le fichier **task-list--32-tasks.cpp** :

```
TaskControlBlock * TaskList::Iterator::nextTask (IRQ_MODE) {
  TaskControlBlock * taskPtr = nullptr ;
  if (mIteratedList != 0) {
    const uint32_t taskIndex = (uint32_t) __builtin_ctz (mIteratedList) ;
    TASK_LIST_ASSERT (taskIndex < TASK_COUNT, taskIndex) ;
    const uint32_t mask = 1U << taskIndex ;
    mIteratedList &= ~ mask ;
    taskPtr = descriptorPointerForTaskIndex (taskIndex) ;
  }
  return taskPtr ;
}
```

Les modifications et ajouts sont terminées, recompilez et exécuter le programme. Notez que maintenant les tâches s'exécutent en parallèle (en fait, pseudo parallélisme), et que la led Teensy est très faiblement éclairée (activité processeur réduite).