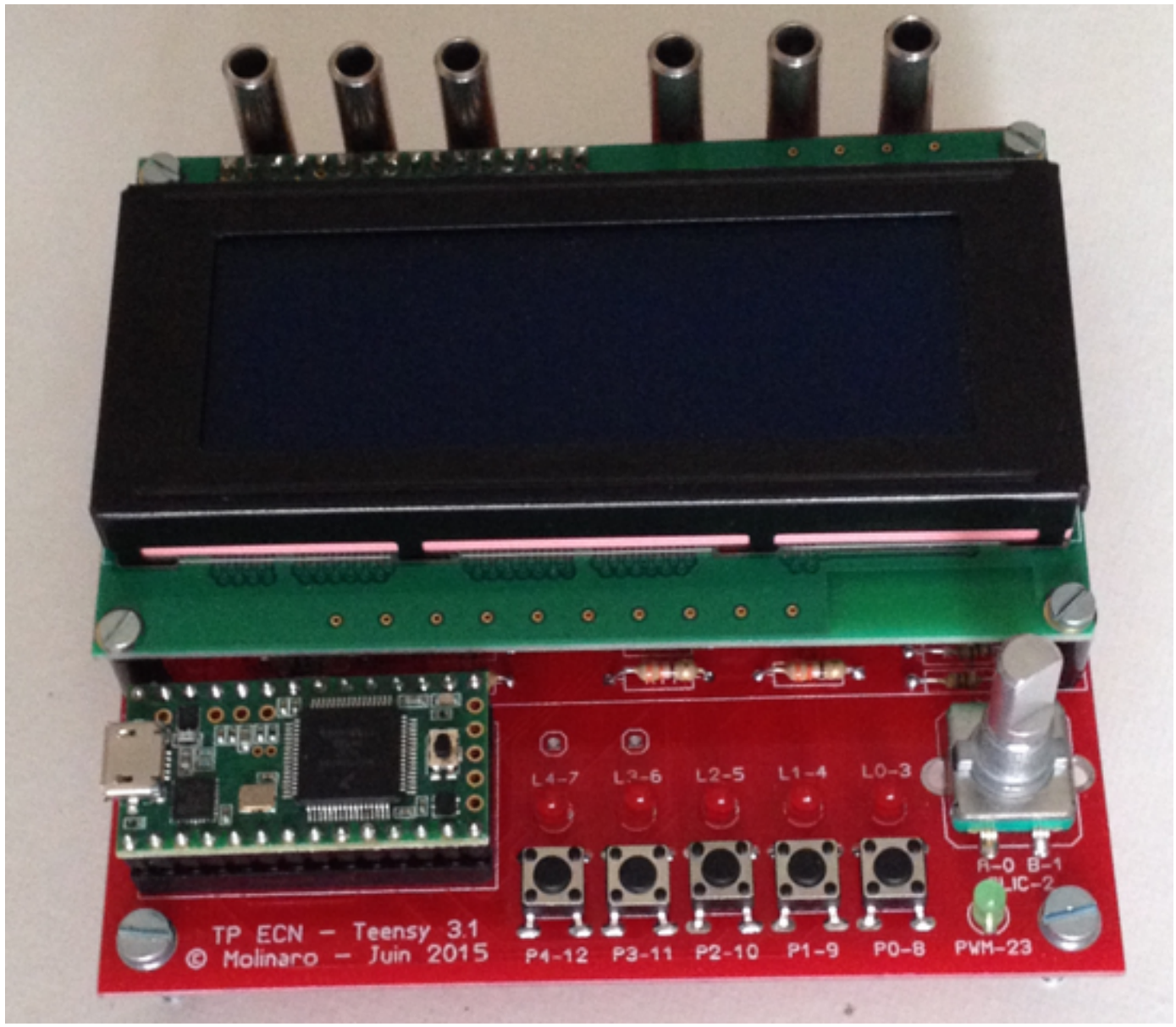
Temps Réel



Programme 15-attente-delai

Pierre Molinaro
Février 2016

But de cette partie

Objectif :

- *ajout de l'attente passive sur délai et sur échéance.*

Travail à faire :

- reprendre le programme précédent, conserver les mêmes tâches qui se terminent, en remplaçant les attentes actives des tâches en attentes passives.

Attente active / attente passive

Jusqu'à présent, l'attente est toujours réalisée par une boucle d'attente active (routine `busyWaitingDuringMS`, dans `busy-waiting.c`) :

```
void busyWaitingDuringMS (BUSY_WAITING_MODE_ const uint32_t inDurationInMS) {
  const uint32_t deadline = uptimeMS () + inDurationInMS ;
  while (uptimeMS () < deadline) {}
}
```

Cette attente est qualifié d'*active* car elle consomme du temps processeur.

Dans cette étape, nous allons mettre au point une attente *passive*, qui libère le processeur.

Attente de délai, attente d'échéance

Nous allons écrire deux routines d'attente passive.

La première est une attente de délai :

```
void waitDuringMS (USER_MODE_ const uint32_t inDurationInMS) ;
```

Cette routine sera à utiliser dans les tâches en remplacement de la routine `busyWaitingDuringMS`.

La seconde est une attente d'échéance :

```
void waitUntilMS (USER_MODE_ const uint32_t inDeadlineInMS) ;
```

Cette routine, que l'on utilise dans les tâches, permet d'obtenir des tâches qui respectent leur période.

Écriture de l'attente de délai

L'attente de délai s'exprime facilement en fonction de l'attente d'échéance :

```
void waitDuringMS (USER_MODE_ const uint32_t inDurationInMS) {
  waitUntilMS (MODE_ uptimeMS () + inDurationInMS) ;
}
```

Il reste donc à écrire la routine d'attente d'échéance `waitUntilMS`. Mais auparavant, nous allons faire le point sur les routines internes à l'exécutif.

Les opérations d'un ordonnanceur

Opération	Commentaire
`task_descriptor * kernel_runningTask (KERNEL_MODE) ;`	Retourne la tâche en cours d'exécution, ou NULL si il n'y en a pas.
`void kernel_makeTaskReady (IRQ_MODE_ task_descriptor * inTaskDescriptor) ;`	Insère la tâche dans la liste des tâches prêtes.
`void kernel_makeNoTaskRunning (KERNEL_MODE) ;`	Indiquer qu'il n'y a plus de tâche en cours d'exécution (parce qu'elle s'est bloquée ou s'est terminée)
`void kernel_selectTaskToRun (KERNEL_MODE) ;`	Si il y a une tâche prête plus prioritaire que la tâche en cours, effectuer une préemption.

Les deux ordonnanceurs implémentés

Ordonnanceur	**Sans priorité** (`scheduler-no-priority.c`)	**Avec priorité** (`scheduler-many-priorities.c`)
Type de liste	File d'attente	Liste triée par priorité décroissante
Insertion d'un nouveau descripteur	Insertion en fin	Insertion respectant la priorité, liste d'attente pour les tâches de même priorité
Retrait de la tâche à exécuter	Retrait de la tête de liste	Retrait de la tête de liste
Tester si il existe une tâche plus prioritaire que la tâche en cours	Toujours faux	Comparer avec la priorité de la tête de liste

Exemple d'utilisation : la préemption

Les routines précédentes ont permis d'écrire le service de preemption (étape 12) :

```
void svc_preempt (KERNEL_MODE) {
  task_descriptor * runningTask = kernel_runningTask (MODE) ; // Obtenir la tâche en cours
  if (NULL != runningTask) { // Si il y a une tâche en cours
    kernel_makeTaskReady (MODE_ runningTask) ; // La placer dans l'état prête
    kernel_makeNoTaskRunning (MODE) ; // Indiquer qu'il n'y a plus de tâche en cours
  }
}
```

list-management.c

Mais pour les autres services à écrire (sémaphore, étape 17), et attente passive (cette étape), on a besoin de gérer différents types de liste :

- listes d'attente, ou ordonnées par priorité (implémentées dans `task-list.c`) ;
- listes ordonnées par échéance (implémentées dans `task-list-by-date.c`).

Pour simplifier l'écriture de ces services, on utilise des routines façade, déclarées dans `list-management.h` et implementées dans `list-management.c`.

Ces routines partagent une instance de liste ordonnée par date (pour les curieux, `gTaskList-OrderedByDate` dans `list-management.c`).

Les routines façade de `list-management.c`

Opération	Commentaire

`void kernel_runningTaskWaitsOnFIFOList`
`(KERNEL_MODE_`
`task_list * inList) ;`
	Bloque la tâche en cours dans la liste FIFO `inList`.

`void kernel_runningTaskWaitsOnListOrderedByPriority`
`(KERNEL_MODE_`
`task_list * inList) ;`
	Bloque la tâche en cours dans la liste ordonnée par priorité `inList`.

`void kernel_runningTaskWaitsOnListOrderedByDate`
`(KERNEL_MODE_`
`const uint32_t inDate) ;`
	Bloque la tâche en cours dans la liste ordonnée par date.

`void kernel_runningTaskWaitsOnFIFOListAndListOrderedByDate`
`(KERNEL_MODE_`
`task_list * inList,`
`const uint32_t inDate) ;`
	Bloque la tâche en cours dans la liste FIFO `inList` et dans la liste ordonnée par date.

`void kernel_runningTaskWaitsOnListOrderedByPriorityAndListOrderedByDate`
`(KERNEL_MODE_`
`task_list * inList,`
`const uint32_t inDate) ;`
	Bloque la tâche en cours dans la liste ordonnée par priorité `inList` et dans la liste ordonnée par date.

`bool kernel_firstWaitingTaskBecomesReady`
`(IRQ_MODE_`
`task_list * inList,`
`const uint32_t inReturnCode) ;`
	Retire la première tâche de la liste `inList`, la rend prête, la retire éventuellement d'une liste ordonnée par date, et fixe son code de retour.

`void kernel_tasksWithEarlierDateBecomeReady`
`(IRQ_MODE_`
`const uint32_t inDate,`
`const uint32_t inReturnCode) ;`
	Retire et rend prêtes toutes les tâches de la liste ordonnée par date dont la date est antérieure ou égale à la valeur de l'argument `inDate`, et fixe leur code de retour.

L'utilité du code de retour apparaîtra lors de l'implémentation du service `Puntil`.

Écriture de l'attente d'échéance

L'attente d'échéance demande l'insertion de la tâche dans la liste ordonnée par date :

```
void svc_waitUntilMS (KERNEL_MODE_ const uint32_t inDeadlineInMS) {
  if (inDeadlineInMS > uptimeMS ()) {
    kernel_runningTaskWaitsOnListOrderedByDate (MODE_ inDeadlineInMS) ;
  }
}
```

La routine d'interruption `systickHandler` rend prêtes les tâches dont l'échéance est atteinte.

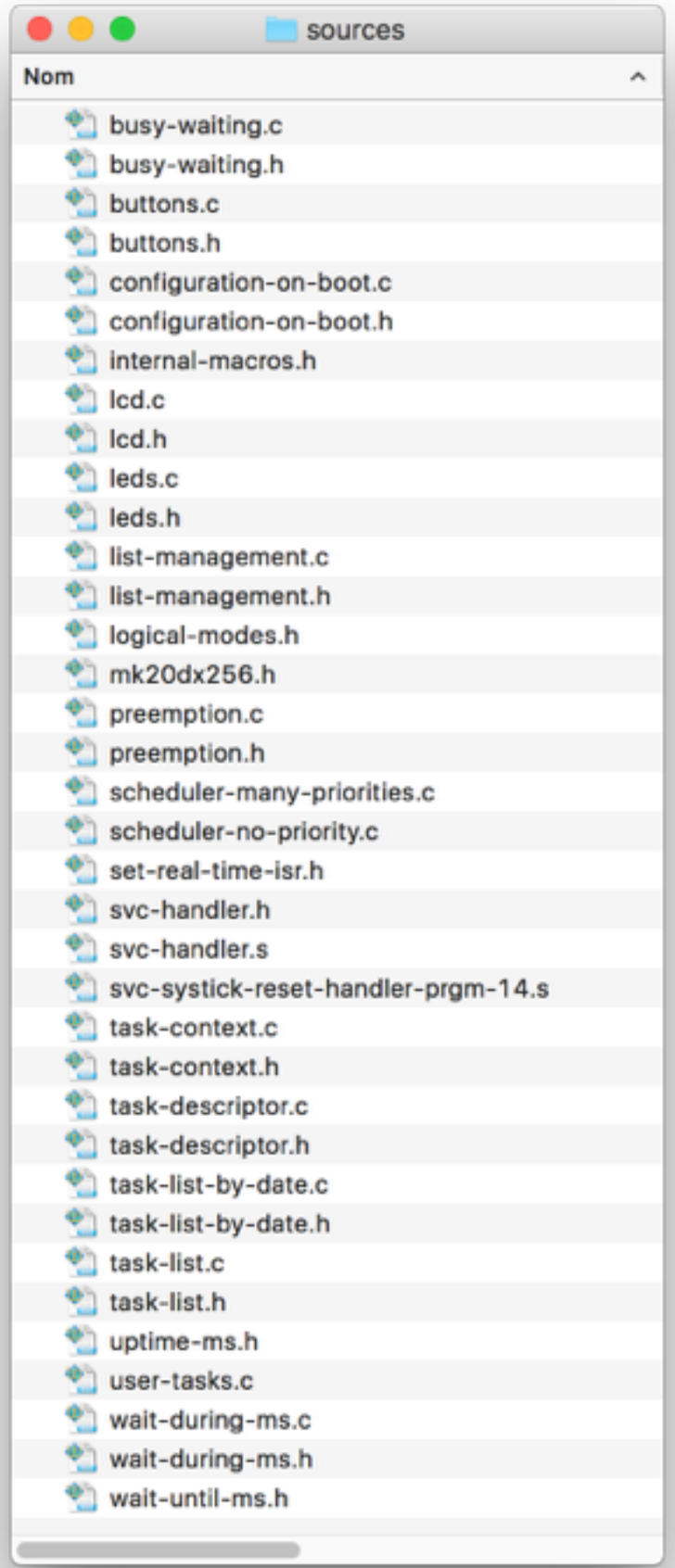
```
void systickHandler (IRQ_MODE) {
  gUpTimeInMilliseconds ++ ;
  kernel_tasksWithEarlierDateBecomeReady (MODE_ uptimeMS (), 0) ;
  if (NULL != gRealTimeISR) {
    gRealTimeISR (MODE) ;
  }
}
```

Le code de retour des tâches rendues prêtes est `0`, ce code n'est dans cette étape inutilisé ; il le sera pour le service `Puntil`.

Travail à faire

- corriger les erreurs de mode (voir page suivante) ;
- récupérer sur le serveur pédagogique l'archive `15-sources.tbz` qui contient :
  - ✳`task-list-by-date.h` et `task-list-by-date.c` ;
  - ✳`list-management.h` et `list-management.c` ;
- écrire la routine `waitDuringMS` dans un fichier `wait-during-ms.c`, et son prototype dans `wait-during-ms.h` ;
- écrire le prototype de la routine `waitUntilMS` dans un fichier `wait-until-ms.h` ;
- ajouter au fichier `configuration-on-boot.c` le code de la routine `waitUntilMS` et compléter la routine `systickHandler` ;
- compléter le fichier assembleur avec le `svc` du service `waitUntilMS`.

Rappel : la led *Teensy* est contrôlée par l'exécutif, de façon à montrer la charge du processeur.



Erreurs à corriger

Il y a des erreurs de déclaration de mode dans les routines fournies lors des étapes précédentes.

Dans `task-context.h` :

```
void kernel_set_return_code (IRQ_MODE_
                             task_context * inTaskContext,
                             const uint32_t inReturnCode) ;
```

Dans `task-list.h` :

```
struct task_descriptor * removeTaskAtHead (IRQ_MODE_
                                           task_list * inTaskList) ;

void removeFromCurrentTaskList (IRQ_MODE_
                                struct task_descriptor * inTaskDescriptor) ;
```

Modifier en conséquence l'implémentation de ces routines.

Résultat attendu

Les leds doivent clignoter ensemble, jusqu'à ce que les tâches se terminent, pas de changement avec l'étape précédente. Par contre, la led Teensy doit en permanence éclairer très faiblement (sauf si vous utilisez l'afficheur LCD, voir page suivante). À la fin des tâches, la led *Teensy* continue à éclairer très faiblement.

Rappel : depuis l'ajout de la préemption sur interruption temps-réel, au plus une tâche a droit d'utiliser l'afficheur LCD.

Travail complémentaire

Un travail complémentaire pour comprendre la différence entre `waitDuringMS` et `waitUntilMS`, et l'intérêt de `waitUntilMS` pour obtenir des exécutions qui respectent une période.

① Écrire deux tâches sans fin qui font chacune clignoter une led (ne pas utiliser l'afficheur LCD) :

- la tâche 1 dont l'intervalle de temps est fixé par `waitDuringMS (MODE_ 199)` ;
- la tâche 2 dont l'intervalle de temps est fixé par `waitDuringMS (MODE_ 200)`.

L'attente de la tâche 1 (`199 ms`) étant inférieur à celui de la tâche 2 (`200 ms`), on voit au bout d'un peu de temps que la tâche 1 prend de l'avance par rapport à la tâche 2.

② Maintenant, modifier le code de la tâche 1 pour qu'elle affiche à chaque itération une information, par exemple le nombre d'itérations. Qu'observe-t'on ?

- d'abord, la gestion de l'afficheur LCD appelle des attentes actives, la led *Teensy* s'éclaire de façon importante au moment où l'affichage a lieu ;
- ensuite, la tâche 1 ne prend plus d'avance par rapport à la tâche 2 ; pourquoi ?

③ Modifier le code des tâches en appelant `waitUntilMS` pour respecter la période.