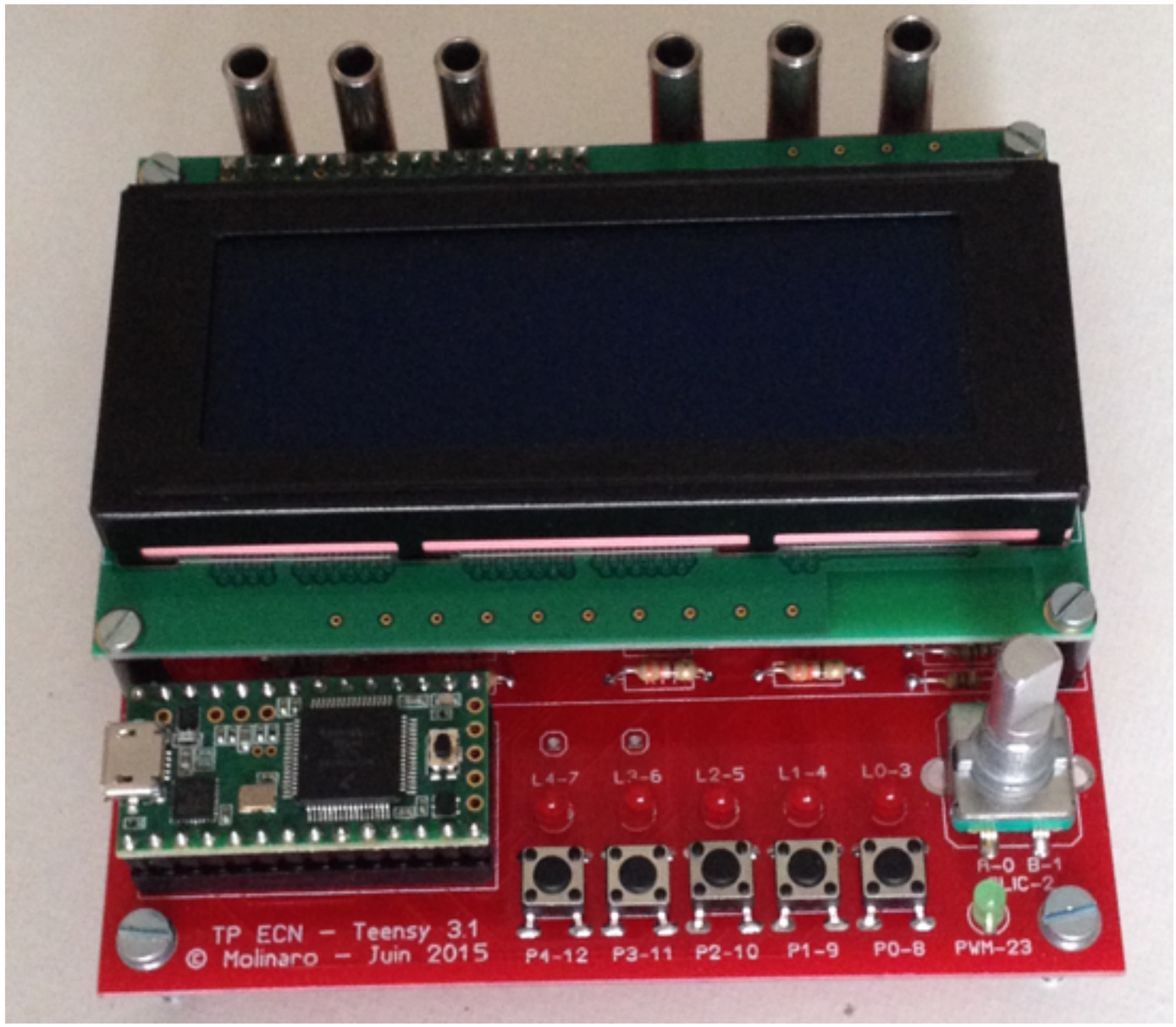
Temps Réel


L4-7
L3-6
L2-5
L1-4
L0-3
R-0 B-1
LIC-2
TP ECN - Teensy 3.1
© Molinaro - Juin 2015
P4-12
P3-11
P2-10
P1-9
P0-8
PWM-23

But de cette partie

Objectif :

- *ajout de la préemption, de façon à réaliser un exécutif coopératif.*

Travail à faire :

- écrire plusieurs tâches (au moins deux) qui font clignoter chacune une led.

Exécutif coopératif

Un *exécutif coopératif* fonctionne grâce au bon vouloir des tâches qui permettent à une autre de s'exécuter en appelant la primitive de préemption.

```
static void task1 (USER_MODE_ const uint32_t inArgument1, const uint32_t inArgument2) {
  while (1) { // INDISPENSABLE pour une tâche qui ne termine pas
    // Code de la tâche
    preempt (MODE) ;
  }
}
```

Deux ordonnanceurs sont aussi fournis à partir de cette étape :

- `scheduler-no-priority.c`, qui implémente un ordonnanceur sans priorité, c'est-à-dire qui ignore la valeur de la priorité qui est fournie lors de l'initialisation du descripteur de tâche ;
- `scheduler-many-priorities.c`, qui tient compte des priorités (aussi, seules les tâches de plus forte priorité s'exécutent).

Attention ! Dans votre fichier `makefile.json`, n'inscrire qu'un seul de ces deux fichiers.

Fonctionnement du `svcHandler` (étape 12)

Le pointeur *`gRunningTaskContextSaveAddress`* désigne le contexte de la tâche en cours

Allumer led *Teensy*

Ancien := ***gRunningTaskContextSaveAddress***

Appeler ***svc_XXX***

Appeler ***kernel_selectTaskToRun***

Ancien != ***gRunningTaskContextSaveAddress***

non

oui

Ancien != NULL ?

non

oui

Sauvegarder ancien contexte

gRunningTaskContextSaveAddress != NULL ?

non

oui

Restituer nouveau contexte

Retour d'exception

Si ce pointeur vaut `NULL`, il n'y a pas de tâche en cours.
Initialement, il vaut `NULL`.

Fichier `svc-handler-prgm-12.s` fourni sur le serveur pédagogique.

L'exécutif est en cours de réalisation, dans cette étape il plante si il n'y a pas de tâche à exécuter.

Fonctionnement du `svcHandler` (étape 12)

Pour comprendre le fonctionnement du `scvHandler`, il faut remarquer qu'il y a deux fonctions qui sont exécutées :

(1) d'abord `svc_XXX`, qui est propre au service `XXX` qui est appelé en mode `USER` : par exemple, si une tâche appelle `prempt`, `svc_preempt` est appelé ;
(2) ensuite, `kernel_selectTaskToRun`.

Le code qui suit est la sauvegarde du contexte de la tâche appelante (qui vient de perdre le processeur), et la restitution du contexte de la tâche qui va s'exécuter (qui a gagné le processeur, et dont l'état passe à `TASK_RUNNING`). Si c'est la même tâche qui continue son exécution, il n'y a ni sauvegarde ni restitution de contexte.

Voici ce qui fait `kernel_selectTaskToRun` :

- si il n'y a pas de tâche en cours, la tâche prête la plus prioritaire est sélectionnée pour être exécutée ;
- si il y a une tâche en cours :
  - ✳ si elle est plus prioritaire ou de même priorité que la tâche prête la plus prioritaire, la tâche en cours continue ;
  - ✳ si elle est strictement moins prioritaire que la tâche prête la plus prioritaire, celle-ci est sélectionnée pour être exécutée.

Comment écrire le service de préemption

Détails pratiques :

- écrire `svc_preempt` dans un fichier `preemption.c` ;
- le début du fichier `svc-handler-prgm-12.s` définit la fonction `preempt` qui appelle `svc #0`, comme vu dans les étapes précédentes.

Voici comment écrire `svc_preempt` :

```
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
  }
}
```

Une remarque concernant `kernel_makeTaskReady` :

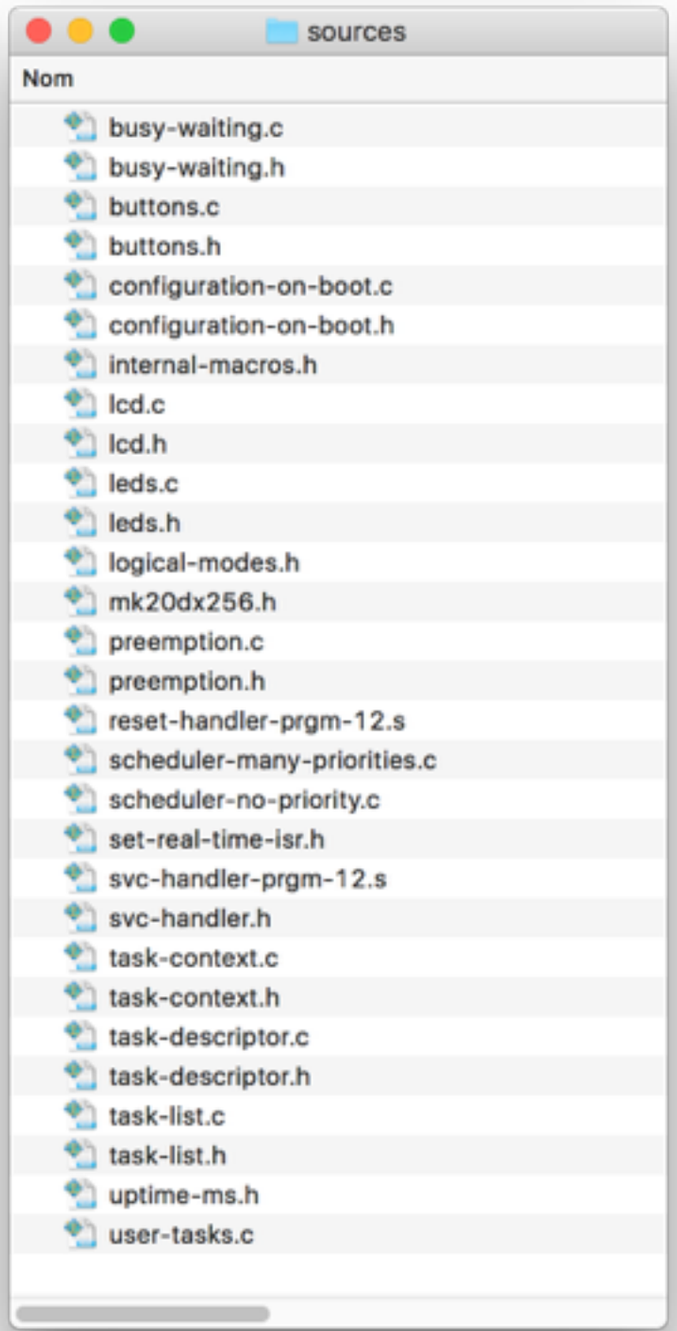
- la liste des tâches prêtes est ordonnée par priorité ;
- la tâche rendue prête est insérée dans la liste des tâches prêtes, après les tâches de même priorité.

Ainsi `prempt` active les tâches de plus forte priorité les unes après les autres (*round-robin*).

Travail à faire

- récupérer sur le serveur pédagogique l'archive `12-sources.tbz` qui contient :
  - `reset-handler-prgm-12.s` ;
  - `scheduler-many-priorities.c` ;
  - `svc-handler-prgm-12.s` ;
- écrire le fichier `preemption.c`, qui définit `svc_preempt` ;
- compléter `user-tasks.c`, écrire plusieurs tâches qui font chacune clignoter une led.

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



Résultat attendu

Les leds doivent clignoter alternativement.

L'ordre dans lequel les tâches s'exécutent dépend de l'ordre dans lequel elles ont été rendues prêtes dans la routine d'initialisation.

Remarquer qu'avec l'ordonnanceur qui tient compte des priorités, déclarer des tâches de priorités différentes affament les tâches les moins prioritaires.

Dans cette étape, les tâches tournent en permanence, le led *Teensy* est allumée en permanence.