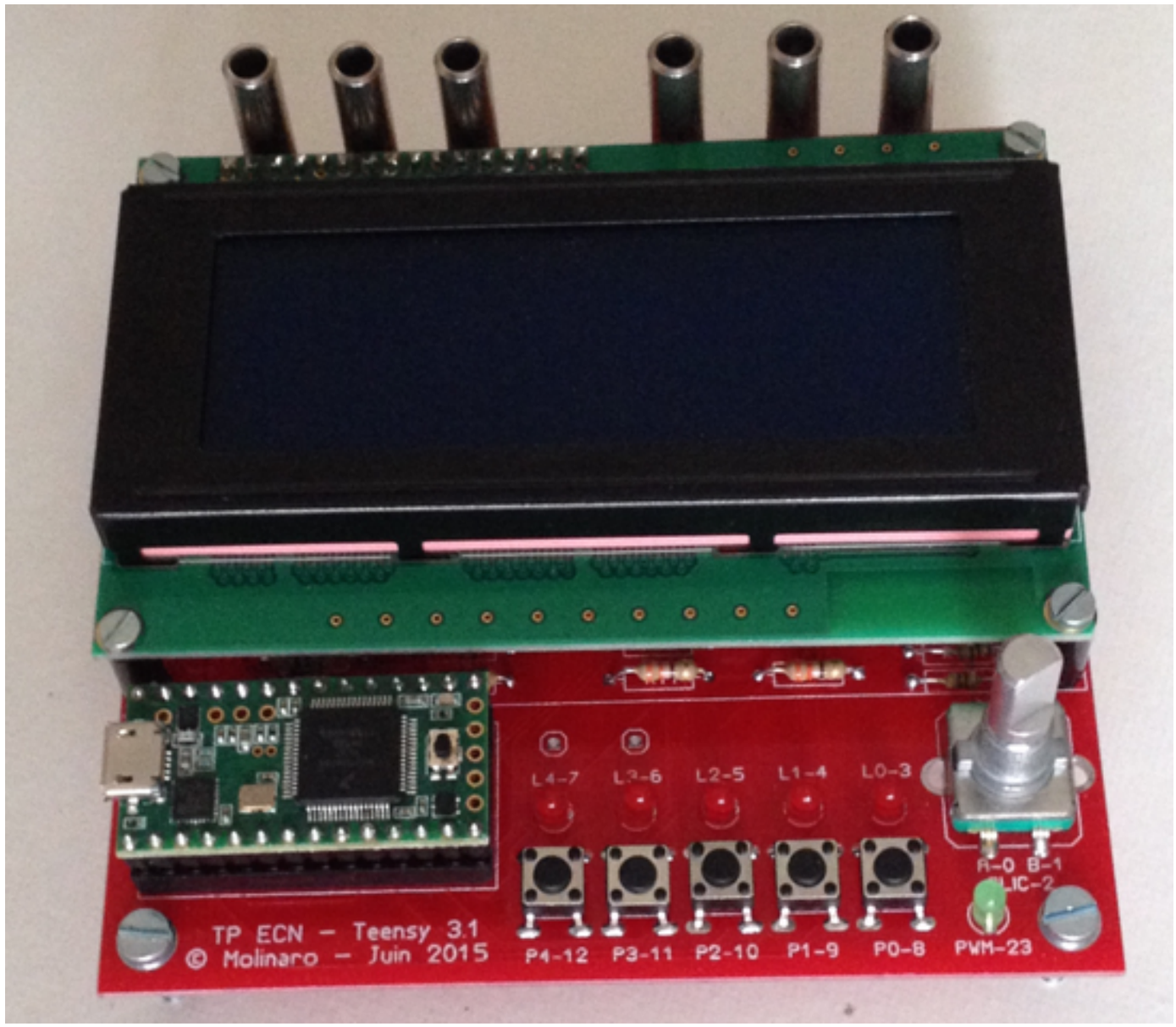
Temps Réel


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 :

- *premier exécutif, qui ne sait que lancer une seule tâche qui ne doit jamais terminer.*

Travail à faire :

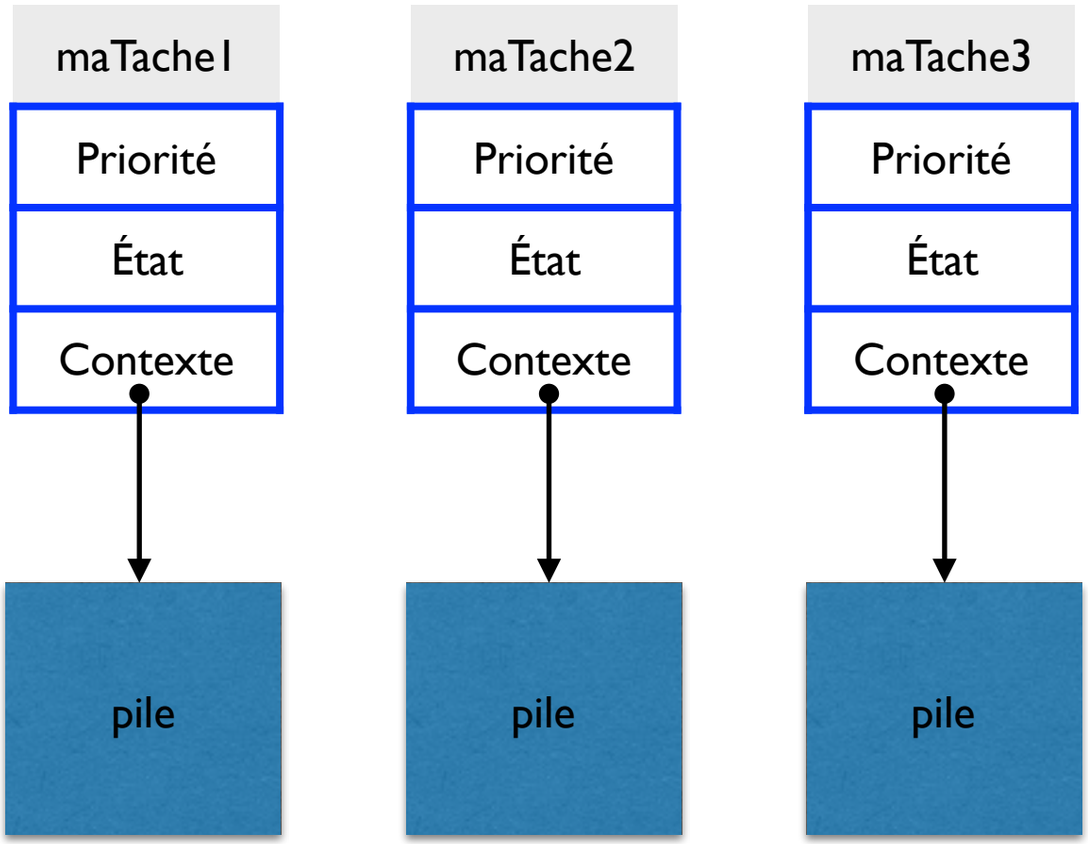
- déclarer une tâche et sa pile, écrire le code d'initialisation et son code qui fait clignoter une led.

Tâche

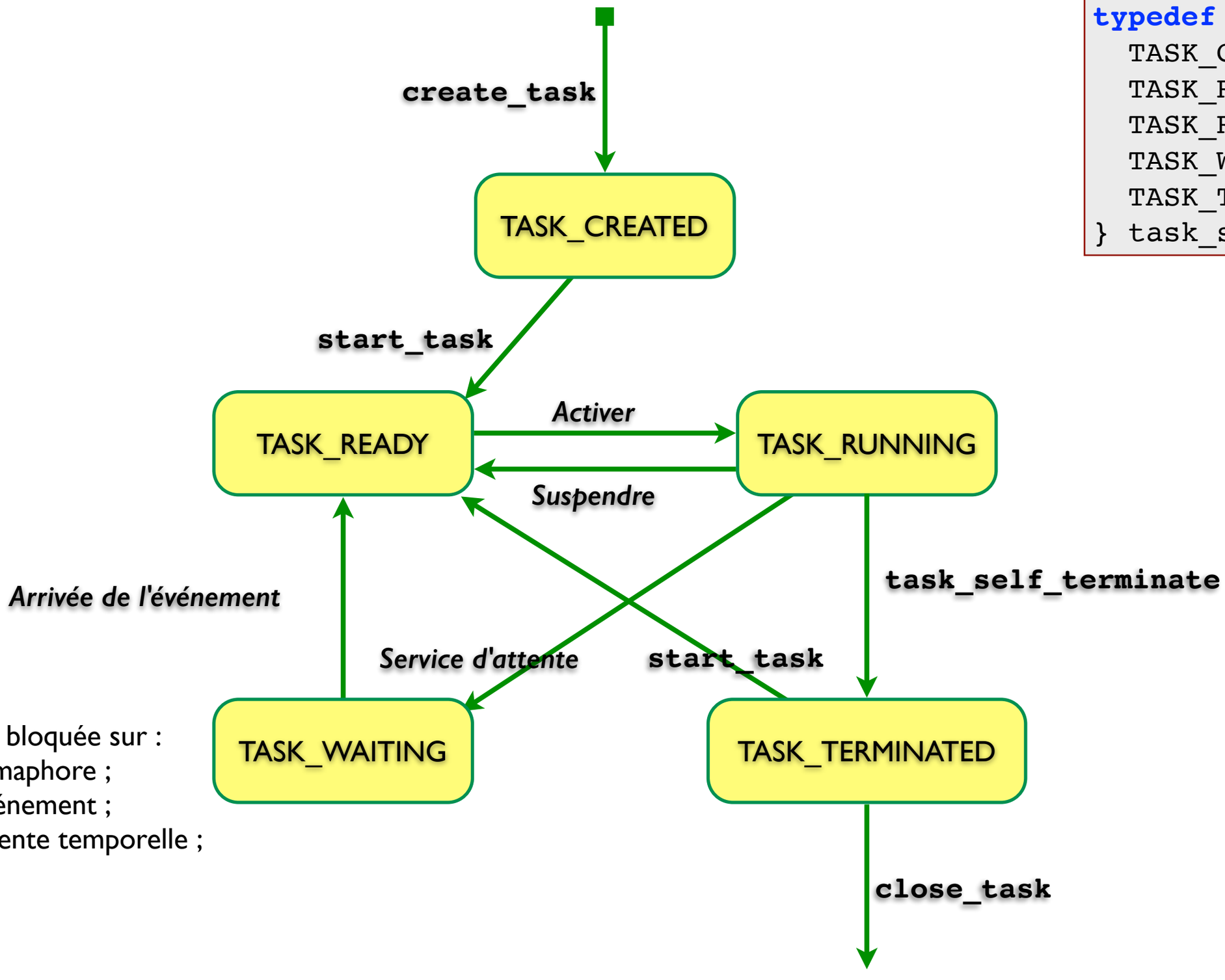
Une ***tâche*** est une exécution particulière d'un code.

L'***état*** précise dans quelle situation d'exécution se trouve le code associé.

L'ensemble des données décrivant une tâche est contenue dans un ***descripteur de tâche***.



États d'une tâche




```
typedef enum {
  TASK_CREATED,
  TASK_READY,
  TASK_RUNNING,
  TASK_WAITING,
  TASK_TERMINATED
} task_state ;
```

Tâche bloquée sur :

- sémaphore ;
- événement ;
- attente temporelle ;
- …

Contexte d'une tâche

Le ***contexte d'une tâche*** est l'ensemble des informations relatives à un avancement particulier de l'exécution d'un programme, c'est à dire la valeur de tous les registres internes : R0 à R12, R13 (SP), R14 (LR), R15 (PC), PSR (*Program Status Register*).

Le descripteur de tâche possède un champ *contexte* qui mémorise le contexte d'une tâche lorsqu'elle n'est pas en exécution.

Quand une tâche ***passe*** dans l'état ***TASK_RUNNING***, le contenu du champ contexte de son descripteur est copié dans les registres du processeur.

Quand une tâche ***est*** dans l'état ***TASK_RUNNING***, le contenu du champ contexte de son descripteur n'est pas significatif, car le contexte est contenu dans les registres du processeur.

Quand une tâche ***quitte*** l'état ***TASK_RUNNING***, le contenu des registres du processeur est copié dans le champ contexte de son descripteur.

```
typedef struct {
  uint32_t mR4 ;
  uint32_t mR5 ;
  uint32_t mR6 ;
  uint32_t mR7 ;
  uint32_t mR8 ;
  uint32_t mR9 ;
  uint32_t mR10 ;
  uint32_t mR11 ;
  uint32_t mSP_USR ;
  uint32_t mLR_USR ;
} task_context ;
```

Pour des raisons pratiques, les registres R0 à R3, R12, et R15 (PC) sont mémorisés dans la pile de la tâche.

Création, démarrage d'une tâche : code à écrire

À partir de cette étape :

- les fichiers `setup-loop.h` et `set-loop.c` doivent être supprimés. Les fonctions `setup` et `loop`, propres à l'exécution sans exécutif, n'existent plus ;
- un fichier `user-tasks.c` est utilisé, qui déclare et lance les tâches.

Voici le fichier `user-tasks.c` pour cette étape :

```
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
  }
}

static task_descriptor gTaskDescriptor1 ; // Descripteur de la tâche
static uint32_t gStack1 [128] ; // Pile de la tâche

static void initTasks (INIT_MODE) {
  kernel_create_task (MODE_ & gTaskDescriptor1, 1, gStack1, sizeof (gStack1)) ;
  kernel_start_task (MODE_ & gTaskDescriptor1, task1, 0, 0) ;
}

MACRO_INIT_ROUTINE (initTasks) ;
```

Création d'une tâche

La ***création d'une tâche*** initialise son descripteur (à l'exception du contexte de la tâche), dont son état à *TASK_CREATED*.

```
static task_descriptor gTaskDescriptor1 ; // Descripteur de la tâche
static uint32_t gStack1 [128] ; // Pile de la tâche

static void initTasks (INIT_MODE) {
  kernel_create_task (MODE_ & gTaskDescriptor1, 1, gStack1, sizeof (gStack1)) ;
  ……
}
```

Adresse du descripteur de tâche

Priorité de la tâche : la valeur 0 est la plus prioritaire

Adresse de la zone mémoire affectée à la pile de la tâche

Taille (en nombre d'octets) de la zone mémoire affectée à la pile de la tâche

Chaque tâche a son propre descripteur et sa propre pile. Plusieurs tâches peuvent avoir la même priorité. Dans cette étape, il n'y a qu'une seule tâche, aussi sa priorité n'a pas d'importance.

Démarrage d'une tâche

Le ***démarrage d'une tâche*** place son état à *TASK_READY*.

```
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
  }
}

static void initTasks (INIT_MODE) {
  ……
  kernel_start_task (MODE_ & gTaskDescriptor1, task1, 0, 0) ;
}
```

Adresse du descripteur de tâche

Nom de la fonction qui définit le code la tâche

Valeur qui sera affectée à `inArgument1`

Valeur qui sera affectée à `inArgument2`

Plusieurs tâches peuvent exécuter le même code : les deux arguments `inArgument1` et `inArgument2` peuvent servir pour différencier leurs exécutions.

Le resetHandler pour cette partie

Le resetHandler est écrit en assembleur (reset-handler-prgm-11.s).

```
resetHandler:
@--- Init micro controller
  bl configure_microcontroller
@--- set PSP
  ldr r0, =backgroundTaskStack + 1024
  msr psp, r0
@--- Set CONTROL register
  mov r0, #3
  msr control, r0
  isb
@---
  bl kernel_selectTaskToRun
@--- Launch task
  svc  #0
#--- Background task : infinite loop
infiniteLoop:
  b  infiniteLoop
```

La fonction kernel_selectTaskToRun, écrite en C, sélectionne la tâche prête la plus prioritaire, place son état à la valeur TASK_RUNNING, et l'adresse de son contexte est affecté à la variable gRunningTaskContextSaveAddress.

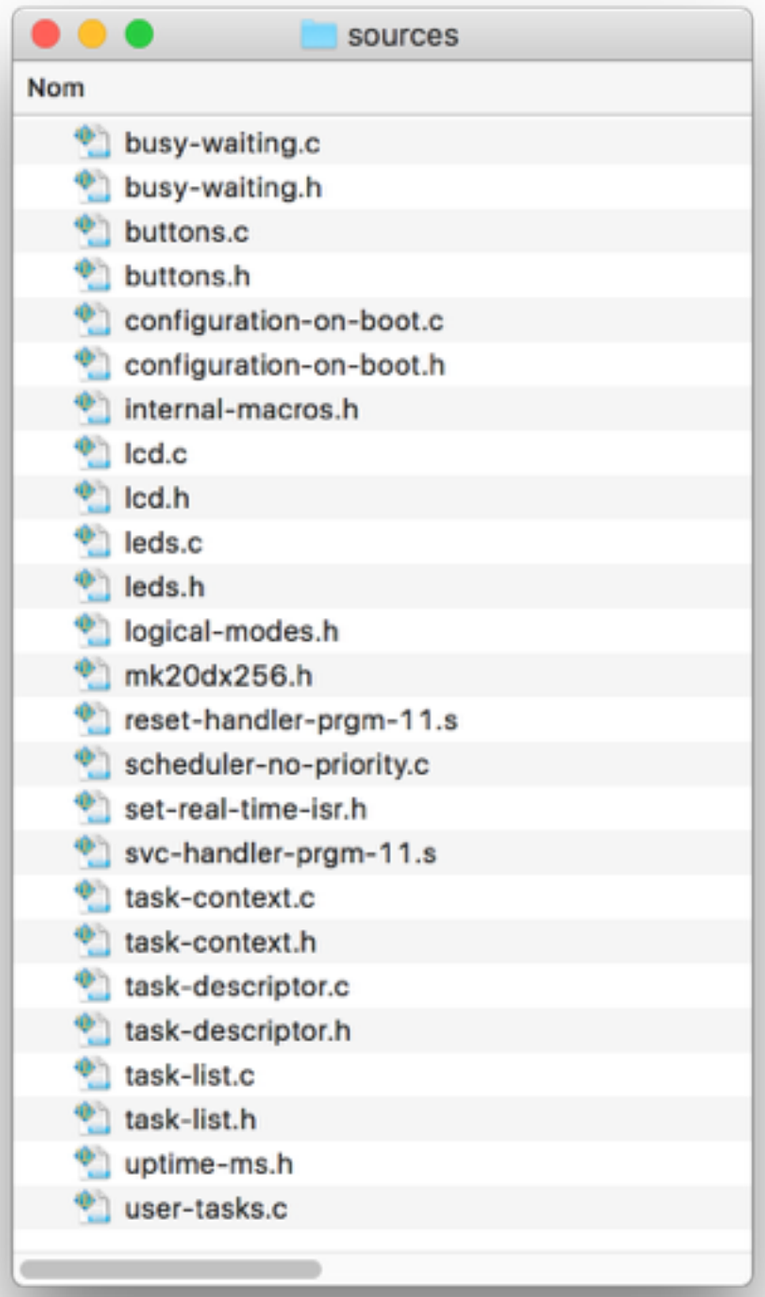
Le svcHandler pour cette partie

Le svcHandler est écrit en assembleur (svc-handler-prgm-11.s).

```
svcHandler:
@---------------------------------------- Activity led On (macro that uses only R4 and R5)
  ACTIVITY_LED_ON
@---------------------------------------- R1 <- new task context
  ldr   r1, =gRunningTaskContextSaveAddress
  ldr   r1, [r1]
@---------------------------------------- Restore context of activated task
  ldmia  r1!, {r4, r5, r6, r7, r8, r9, r10, r11, r12, lr}
  msr    psp, r12
@--- Return from interrupt
  bx     lr
```

Travail à faire

- récupérer sur le serveur pédagogique l'archive 11-sources.tbz qui contient :
  - ✳logical-modes.h ;
  - ✳reset-handler-prgm-11.s ;
  - ✳scheduler-no-priority.c ;
  - ✳svc-handler-prgm-11.s ;
  - ✳task-context.c et task-context.h ;
  - ✳task-descriptor.c et task-descriptor.h ;
  - ✳task-list.c et task-list.h ;
- seul fichier à compléter : user-tasks.c, de façon à faire clignoter une led (ne pas choisir la led *Teensy*). Vous pouvez utiliser toutes les routines des boutons poussoirs, leds, LCD, et busyWaitingDuringMS.



À partir de cette étape, la led Teensy est contrôlée par l'exécutif, de façon à montrer la charge du processeur.

Résultat attendu

La led choisie doit clignoter.

La led *Teensy* est allumée en permanence : allumée, elle indique le processeur exécute du code de l'exécutif ou des tâches ; éteinte, le processeur n'exécute aucun code de l'exécutif ni des tâches.

Dans cette étape, la tâche tourne en permanence, le led *Teensy* est allumée en permanence.