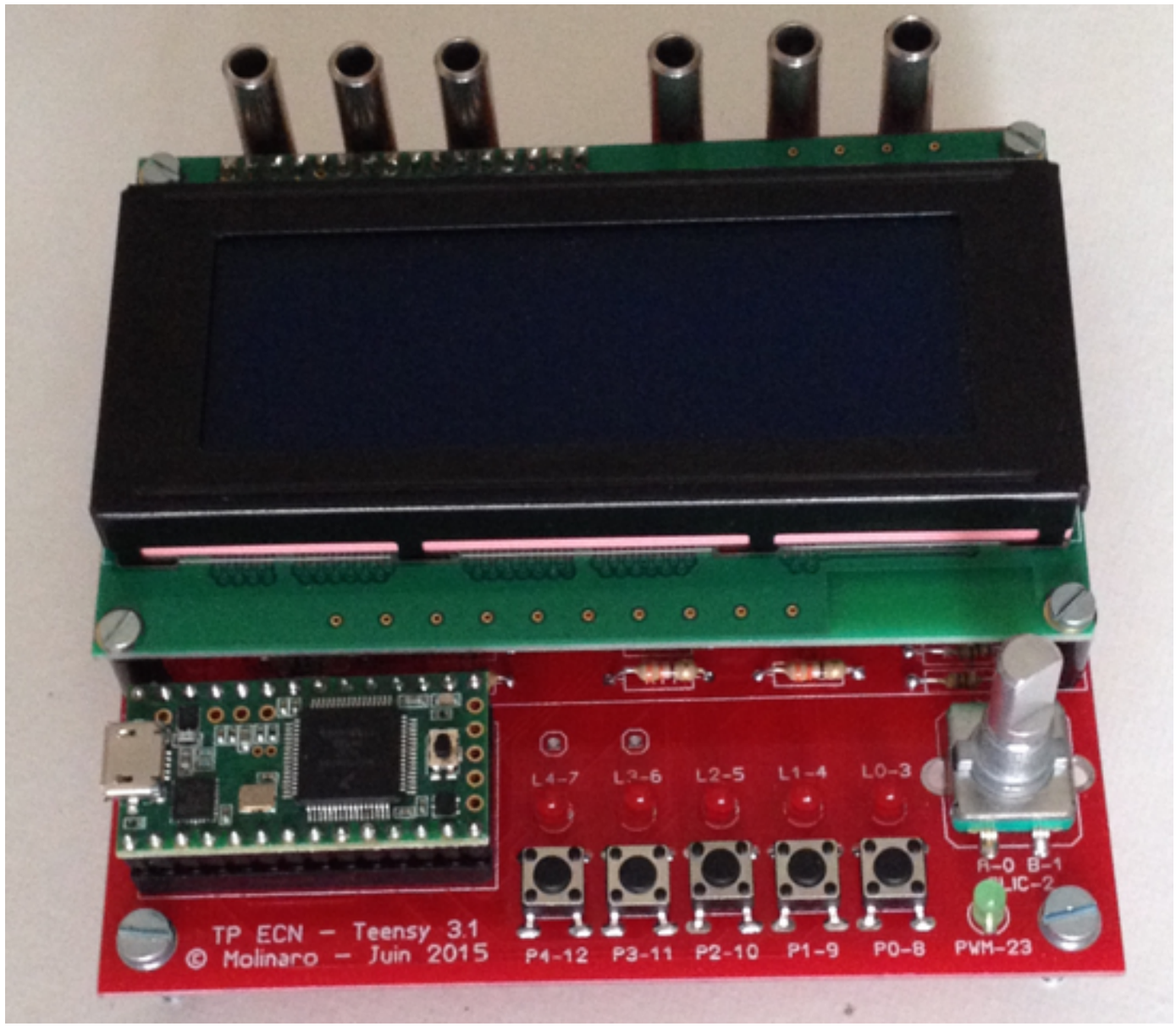
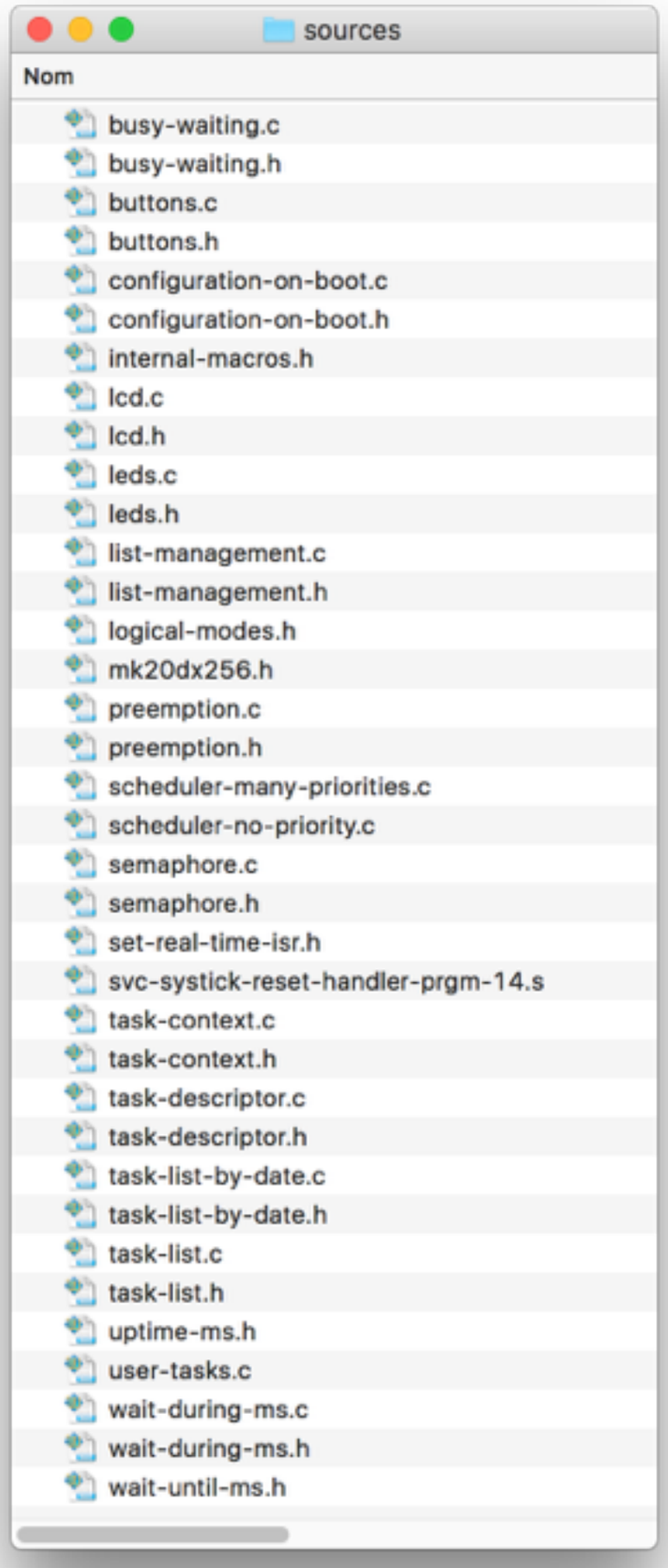
Temps Réel


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 du sémaphore de Dijkstra.*

Travail à faire :

Reprendre le programme précédent, le compléter avec au moins deux tâches qui écrivent sur l'afficheur LCD. Que constate-t'on ? Deux choses :

- les caractères affichés peuvent être incorrects ;
- les affichages peuvent être entremêlés.

On utilisera donc les sémaphores à deux endroits différents pour obtenir un affichage correct.

Un exemple d'utilisation des sémaphores (1/2)

À exécuter sur votre ordinateur, et non pas sur la carte de TP

```
#include <iostream>
#include <thread>

//----------------------------------------*

static const int NOMBRE_THREADS = 10 ;

//----------------------------------------*

static void codeThread (int tid) {
  std::cout << "thread " << tid << std::endl ;
}

//----------------------------------------*

int main() {
//--- Déclaration des threads
  std::thread t [NOMBRE_THREADS] ;
//--- Démarrage des threads
  for (int i=0 ; i<NOMBRE_THREADS ; i++) {
    t[i] = std::thread (codeThread, i) ;
  }
//--- Message
  std::cout << "main\n";
//--- Attente de la fin de l'exécution des threads
  for (int i=0 ; i<NOMBRE_THREADS ; i++) {
    t[i].join () ;
  }
//---
  return 0;
}
```

Compilation : `g++ main.cpp -o main`

Lire attentivement le programme ci-contre, et le faire tourner sur votre ordinateur de bureau.

11 threads se déroulent en parallèle, chacun d'eux affiche un message.

Voici le résultat de deux exécutions :

```
main
thread 9
thrttttttethhhhhhahrrrrrrrdreeeeeee eaaaaaaa1adddddddd
        23074568
```

```
mttatthttttthtihhrhhhhhrhnrrerrrrrer
eeaeeeeeaeaadaaaaadadd ddddd d  9     8 03
25164
7
```

Les caractères sont corrects, mais les affichages sont entremêlés.

Note : sur votre plateforme, vous pouvez être amené à utiliser les options de compilation suivantes :

- `-std=c++11` (les threads sont définis à partir du C++ 11) ;
- `-lpthread` (édition des liens avec la librairie libpthread).

Un exemple d'utilisation des sémaphores (2/2)

À exécuter sur votre ordinateur, et non pas sur la carte de TP

```
#include <iostream>
#include <thread>

//----------------------------------------------*

static const int NOMBRE_THREADS = 10 ;
static std::mutex semaphore ; // Sémaphore initialisé à 1

//----------------------------------------------*

static void codeThread (int tid) {
  semaphore.lock () ; // P(semaphore)
  std::cout << "thread " << tid << std::endl ;
  semaphore.unlock () ; // V (semaphore)
}

//----------------------------------------------*

//----------------------------------------------*

int main() {
//--- Déclaration des threads
  std::thread t [NOMBRE_THREADS] ;
//--- Démarrage des threads
  for (int i=0 ; i<NOMBRE_THREADS ; i++) {
    t[i] = std::thread (codeThread, i) ;
  }
//--- Message
  semaphore.lock () ; // P(semaphore)
  std::cout << "main\n";
  semaphore.unlock () ; // V (semaphore)
//--- Attente de la fin de l'exécution des threads
  for (int i=0 ; i<NOMBRE_THREADS ; i++) {
    t[i].join () ;
  }
//---
  return 0;
}
```

On ajoute maintenant un sémaphore d'exclusion mutuelle (en bleu).

Voici le résultat de deux exécutions :

```
main
thread 0
thread 7
thread 8
thread 6
thread 9
thread 1
thread 3
thread 4
thread 2
thread 5
```

```
thread 2
main
thread 4
thread 8
thread 9
thread 5
thread 0
thread 1
thread 6
thread 3
thread 7
```

Les affichages sont corrects, l'ordre peut varier d'une exécution à une autre.

Note : sur votre plateforme, vous pouvez être amené à utiliser les options de compilation suivantes :

- `-std=c++11` (les threads sont définis à partir du C++ 11) ;
- `-lpthread` (édition des liens avec la librairie libpthread).

Écriture d'un sémaphore

Nous allons maintenant écrire le sémaphore de Dijkstra, plus précisément :

- la déclaration du type `Semaphore` ;
- la routine d'initialisation `init_semaphore`.
- la primitive `P` ;
- la primitive V.

Écrire les déclarations dans un fichier `semaphore.h` et les implémentations dans un fichier `semaphore.c`.

Note. Primitive : fonction élémentaire d'un exécutif, implémentée par un *Service Call*.

Déclaration d'un sémaphore

```
#ifndef SEMAPHORE_DEFINED
#define SEMAPHORE_DEFINED

//------------------------------------------------------------------------------*

#include "task-list.h"
#include "logical-modes.h"

//------------------------------------------------------------------------------*

#include <stdint.h>
#include <stdbool.h> // Utile à l'étape suivante

//------------------------------------------------------------------------------*

typedef struct  {
  task_list mWaitingTaskList ;
  uint32_t mValue ;
} Semaphore ;

//------------------------------------------------------------------------------*

... Déclaration de la routine d'initialisation et des services P et V...

//------------------------------------------------------------------------------*

#endif
```

Initialisation d'un sémaphore

```
#include "semaphore.h"
#include "list-management.h"
#include "uptime-ms.h" // Utile à l'étape suivante

//-----------------------------------------------------------------------------*

void init_semaphore (INIT_MODE_
                     Semaphore * inSemaphore,
                     const uint32_t inInitialValue) {
  initTaskList (MODE_ & inSemaphore->mWaitingTaskList) ;
  inSemaphore->mValue = inInitialValue ;
}

//-----------------------------------------------------------------------------*

... Implémentation des services P et V
```

L'initialisation d'un sémaphore s'effectue dans le mode `INIT`, de façon qu'il soit configuré avant le démarrage des tâches ; la fonction d'initialisation présente deux arguments, un pointeur vers un sémaphore, et une valeur initiale **positive ou nulle**.

Primitive P d'un sémaphore

```
void P (USER_MODE_ Semaphore * inSemaphore) ;

//-----------------------------------------------------------------------------*

void svc_P (KERNEL_MODE_ Semaphore * inSemaphore) {
  if (inSemaphore->mValue > 0) {
    inSemaphore->mValue -- ;
  }else{
    kernel_runningTaskWaitsOnFIFOList (MODE_ & inSemaphore->mWaitingTaskList) ;
  }
}
```

La primitive `P` entraîne le blocage de la tâche appelante sur la liste `mWaitingTaskList` si la valeur du sémaphore `mValue` est nulle lors de l'appel. `svc_P` s'exécute en mode `KERNEL` ce qui interdit à une routine d'interruption de l'appeler.

Primitive V d'un sémaphore

```
void V (USER_MODE_ Semaphore * inSemaphore) ;

//-----------------------------------------------------------------------------*

void svc_V (IRQ_MODE_ Semaphore * inSemaphore) {
  const bool found = kernel_firstWaitingTaskBecomesReady (MODE_ & inSemaphore->mWaitingTaskList, 1) ;
  if (! found) {
    inSemaphore->mValue ++ ;
  }
}
```

La primitive `V` rend prête la première tâche bloquée sur la liste `mWaitingTaskList` ; si celle-ci est vide au moment de l'appel, le sémaphore est incrémenté. `svc_V` s'exécute en mode `IRQ` car une routine d'interruption peut l'appeler. `svc_V` peut être aussi appelé en mode `KERNEL`.

Première modification du code

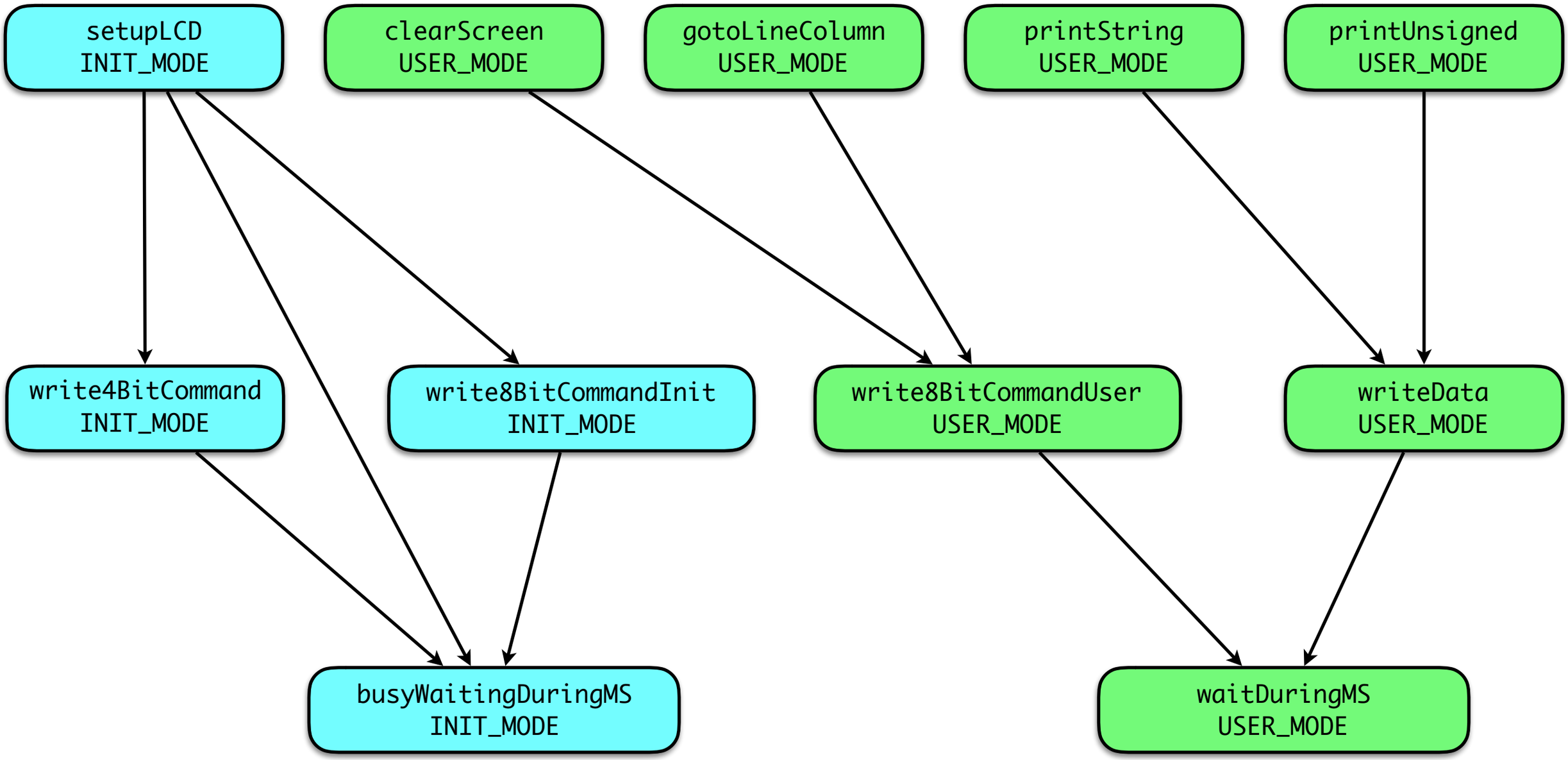
Reprendre le programme précédent, le compléter avec deux tâches qui écrivent sur l'afficheur LCD. Que constate-t'on ? Deux choses :

- les caractères affichés peuvent être incorrects ;
- les affichages peuvent être entremêlés.

Nous allons d'abord corriger le premier point ; pour cela, nous allons modifier `lcd.c`. En examinant le code, on remarque que tous les appels en mode USER aboutissent à un appel de `write8BitCommandUser` ou de `writeData`. Les caractères affichés sont incorrects car l'exécution de ces fonctions peut être préemptée.

Graphe d'appel partiel des fonctions de `lcd.c`

Rappel, page figurant dans la description de l'étape précédente.



Le `BUSY_WAITING` mode a disparu, la fonction `write8BitCommand` doit être dupliquée et adaptée à chaque mode.

Modification de `lcd.c`

Nous allons donc ajouter un sémaphore d'exclusion mutuelle pour protéger l'appel de `write8BitCommandUser` et de `writeData`.

Déclarer le sémaphore :

```
static Semaphore gSemaphoreExclusionMutuelle ;
```

L'initialiser dans la routine `setupLCD` :

```
static void setupLCD (INIT_MODE) {
  init_semaphore (MODE_ & gSemaphoreExclusionMutuelle, 1) ;
  ...
}
```

Protéger la routine `write8bitCommandUserMode` :

```
static void write8bitCommandUserMode (USER_MODE_ const uint8_t inCommand) {
  P (MODE_ & gSemaphoreExclusionMutuelle) ;
    ...
  V (MODE_ & gSemaphoreExclusionMutuelle) ;
}
```

Procéder de même avec la routine `writeData`.

Résultat attendu après la modification de `lcd.c`

Maintenant, les caractères affichés doivent être corrects, mais les affichages sont toujours entremêlés.

On est dans la même situation que dans la première version du programme d'exemple à exécuter sur votre ordinateur de bureau (c'est-à-dire sans le semaphore de type `std::mutex`).

Modification du code de vos tâches

Dans le fichier `user-tasks.c`, ajouter un sémaphore d'exclusion mutuelle de façon à obtenir un affichage correct.

Maintenant, vous pouvez écrire des tâches qui utilisent chacune l'afficheur LCD, sous réserve que l'accès se fasse en exclusion mutuelle. Remarquer que la led *Teensy* reste toujours faiblement éclairée.

Exercice n° I

Modifier le code des tâches de façon à ce que :

- une seule tâche effectue des attentes sur `waitUntilMS`, les autres attendent sur une synchronisation émise par cette tâche.

Exercice n°2

Imaginer un dispositif logiciel de façon que la primitive V d'un sémaphore soit appelée quand on appuie sur le poussoir P0, plus précisément quand le poussoir passe de l'état *relâché* à *appuyé*. Si le poussoir est continuellement appuyé, la primitive V n'est pas appelée.