OS temps-réel sur STM32

J.-M Friedt, 1er septembre 2019

Nous proposons d'aborder l'utilisation des environnements exécutifs pour développer sur microcontrôleur. Un environnement exécutif fournit au programmeur un certain nombre de fonctionnalités attendues d'un système d'exploitation – multitude de tâches capables d'échanger des données et donc de garantir l'intégrité des informations notamment par la protection par *mutex* ou l'utilisation de queues de transferts [1] – mais en ne générant qu'un exécutable monolithique aux ressources connues lors de la compilation. Nous n'aurons donc pas l'occasion de charger dynamiquement d'exécutable ou de bibliothèque, les ressources sont déclarées à la compilation. L'intérêt de travailler sur environnement exécutif au lieu de développer en C tient en l'organisation du travail et la portabilité du code :

1. plusieurs programmeurs travaillent chacun sur sa tâche, et la mise en commun des données acquises (exemple : producteur de données pour l'acquisition, consommateur pour le traitement) se fait au travers des mécanismes de partage fournis par l'environnement exécutif,
2. sous réserve de prendre soin de n'inclure aucun appel bas niveau (matériel) dans son programe, le code ne faisant que appel aux ressources algorithmiques fournies par FreeRTOS sera portable entre toutes les architectures supportées, dont une liste est consultable dans `FreeRTOS/Source/portable/GCC`. Les processeurs ARM y sont évidemment largement représentés, ainsi que AVR, Coldfire (Freescale), MSP430 ou PowerPC.

Dans le dépôt proposé à `https://github.com/jmfriedt/tp_freertos/` la séparation matériel-logiciel est maintenue en ne plaçant des fonctions contenant les appels au bas niveau *que* dans le répertoire `common`. La bibliothèque proposée historiquement par ST Microelectronics se nomme `libstm32` et ne sera plus utilisée avec l'avènement des versions stables de `libopencm3` [2], une bibliothèque générique pour processeurs Cortex M3 et M4 indépendante d'un vendeur particulier. Nous travaillerons sur l'archive de FreeRTOS version 10.2 disponible à `http://www.freertos.org/a00104.html` ou `sourceforge.net/projects/freertos/`. que nous supposerons situé au même niveau que l'arborescence du TP. Le code est testé comme fonctionnel sur STM32F100, STM32F103 et STM32F407, en choisissant le `Makefile` approprié.

1 Du C à FreeRTOS

FreeRTOS (`www.freertos.org`) amène un niveau d'abstraction additionnel par rapport à la programmation en C par l'ajout d'un *scheduler* et de la notion de tâches avec des priorités. Il s'agit d'un environnement exécutif, donc une méthode de travail qui donne l'impression du point de vue du développeur d'avoir accès à des méthodes fournies par un système d'exploitation, mais avec génération d'un binaire monolithique et donc déterministe qui ne peut pas charger dynamiquement des exécutables ou des bibliothèques. FreeRTOS est un OS temps réel, signifiant qu'il *borne les latences d'accès* à un processus.

FreeRTOS ne contient que très peu de fichier implémentant les ressources fournies par l'environnement exécutif : les six fichiers `.c` de `FreeRTOSv10.0.0/FreeRTOS/Source` sont portables, et dans le sous répertoire `portable/GCC` seuls le contenu des deux répertoires `ARM_CM3` et `ARM_CM4F`, selone que nous soyons sur Cortex-M3 ou Cortex-M4 avec unité de calcul flottant, nous intéressent. Ces derniers fichiers incluent des fonctions spécifiques à chaque implémentation matérielle, telles que le *timer* qui permet de cadence l'ordonnance préemptif (option `configUSE_PREEMPTION 1` dans `src/FreeRTOSConfig.h`). L'ensemble des options de `FreeRTOSConfig.h` est documenté dans `www.freertos.org/a00110.html`.

⚠La fréquence du processeur, qui détermine la fréquence du timer qui cadence l'ordonnanceur, est indiquée dans `src/FreeRTOSConfig.h`, indépendamment de la configuration que nous avons indiquée dans la configuration des horloges dans `common` (constante `configCPU_CLOCK_HZ`). Le timer qui se charge de cadencer l'ordonnanceur sur ARM est SysTickTimer. Cette variable détermine aussi la constante utilisée dans le délai implémenté par `portTICK_RATE_MS` en argument de `vTaskDelay()`.

La multitude des processeurs déclinés autour de l'architecture STM32 impose de prendre soin de bien définir les capacités du processeur et les périphériques disponibles. En particulier dans le `Makefile`, définir la nature du processeur (`-DSTM32F10X_LD_VL` pour les processeurs les plus petits, `-DSTM32F10X_MD` pour les composants de densité moyenne – MD) est un point fondamental sans lequel le programme ne peut s'exécuter faute d'une initialisation appropriée.

En cas d'échec de l'exécution sur la carte STM32-Discovery, vérfier que la ligne `-DSTM32F10X_LD_VL` est active.

1. L'argumentaire en faveur d'utiliser un environnement exécutif est développé à `www.freertos.org/FAQWhat.html#WhyUseRTOS`.
2. `https://github.com/libopencm3/libopencm3`

1.1 Premier exemple

Le premier exemple reproduit le résultat du programme en C ci-dessus, mais dans le contexte de FreeRTOS, afin d'apprendre à créer une tâche et d'y associer des actions sur GPIO et communication sur port série asynchrone (RS232).

La principale fonction à noter est la création de tâches, `xTaskCreate()` [2, p.6], qui prend en argument la fonction à appeler, le nom (jamais utilisé), la taille de la pile allouée à la tâche, les arguments, la priorité de la tâche, et un pointeur de retour de la structure représentant la tâche.

La fonction appelée par `xTaskCreate()` ne doit *jamais s'arrêter* (boucle infinie).

```
1  #include "FreeRTOS.h"
   #include "task.h"
3  #include "common.h"

5  void vLedsFloat(void* dummy)
   {while(1){
7    Led_Hi1();
     vTaskDelay(120/portTICK_RATE_MS);
9    Led_Lo1();
     vTaskDelay(120/portTICK_RATE_MS);
11  }
   }
13
   void vLedsFlash(void* dummy)
15 {while(1){
     Led_Hi2();
17   vTaskDelay(301/portTICK_RATE_MS);
     Led_Lo2();
19   vTaskDelay(301/portTICK_RATE_MS);
    }
21 }

23 /* Writes each 500 ms */
   void vPrintUart(void* dummy)
25 {portTickType last_wakeup_time;
    last_wakeup_time = xTaskGetTickCount();
27  // while (1) {}
    while(1){uart_puts("Hello World\r\n");
29     vTaskDelayUntil(&last_wakeup_time, 500/portTICK_RATE_MS);
     }
31 }

33 int main(void){
    volatile int i;
35  Usart1_Init(); // inits clock as well
    Led_Init();
37  Led_Hi1();

39  if (!(pdPASS == xTaskCreate( vLedsFloat, (signed char*) "LedFloat",64,NULL,1,NULL ))) goto hell;
    if (!(pdPASS == xTaskCreate( vLedsFlash, (signed char*) "LedFlash",64,NULL,2,NULL ))) goto hell;
41  if (!(pdPASS == xTaskCreate( vPrintUart, (signed char*) "Uart",    64,NULL,3,NULL ))) goto hell;

43  vTaskStartScheduler();
   hell:              // should never be reached
45   while(1);
      return 0;
47 }
```

tp_freertos/1basic/src/main.c

1. **Commenter la ligne 61 contenant le délai entre deux affichages et observer le résulat.**
2. **Ajouter l'affichage de la liste des tâches en cours d'exécution. Cette fonction est fournie par `vTaskList`, qui nécessite de modifier la configuration de FreeRTOS en conséquent, et de définir un tableau d'une quarantaine de caractères par tâche à afficher. On trouvera les options à ajouter à `FreeRTOSConfig.h` dans la documentation technique de FreeRTOS.**
   La sortie de ce dernier exemple sera de la forme

```
Hello World
Uart            R       4       301     3
```

```
IDLE            R       0       0       4
LedFlash        B       4       242     2
LedFloat        B       4       242     1
```

avec le statut de la tâche (R=Ready, B=Blocked, D=Deleted et S=Suspended=Blocked sans timeout) et la quantité de mémoire encore disponible sur la pile.

3. **Faire évoluer la taille de la pile de chaque tâche et observer la conséquence.**
4. **Observer le comportement pré-emptif de l'ordonnanceur qui peut interrompre une tâche qui a dépassé le temps qui lui est alloué (bloquer volontairement une tâche de clignotement de LED par une `while (1) {};` dans la tâche de priorité la plus faible.**
5. **Intervertir la priorité des deux tâches LEDs et constater que si la tâche bloquante a la plus forte priorité, les autres tâches ne pourront plus prendre la main.**
6. Le dépassement de pile est un problème courant dans la gestion de la mémoire d'un système embarqué aux ressouces réduites. FreeRTOS fournit un mécanisme, au travers de l'option `#define configCHECK_FOR_STACK_OVERFLOW 2` qui nécessite la définition du gestionnaire d'évènement `void vApplicationStackOverflowHook(TaskHandle_t xTask, signed char *pTaskName)`

   Le tâche d'attente *idle* se voit allouer une pile de `configMINIMAL_STACK_SIZE`. Activer la détection de dépassement de pile et conserver la définition de `configMINIMAL_STACK_SIZE` à 10 comme proposé par défaut. Constater le résultat.
7. **Ayant augmenté la taille de la pile par défaut, baisser la taille de la pile de la tâche de communication sur RS232 à 10. Constater le résultat.**
8. Finalement, FreeRTOS permet de connaître la taille restante de la pile par la fonction `uxTaskGetStackHighWaterMark(NULL);` (NULL signifiant que la tâche veut connaître son propre état, sinon il faut fournir le handler de la tâche consultée) si l'option `#define INCLUDE_uxTaskGetStackHighWaterMark 1` est active dans le fichier de configuration.

   Démontrer l'affichage de la pile restant dans la tâche `vPrintUart()`, et la cohérence en faisant évoluer la définition de la taille de la pile à la création de la tâche.

1.2 Échange de messages entre tâches

Des tâches ne sauraient vivre indépendamment les unes des autres dans le contexte d'un programme manipulant des données communes.

```
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#include "common.h"

void vLedsFloat(void* dummy);
void vLedsFlash(void* dummy);
void vPrintUart(void* dummy);

void vLedsFloat(void* dummy)
{while(1){
    Led_Hi1();
    vTaskDelay(120/portTICK_RATE_MS);
    Led_Lo1();
    vTaskDelay(120/portTICK_RATE_MS);
 }
}

void vLedsFlash(void* dummy)
{while(1){
    Led_Hi2();
    vTaskDelay(301/portTICK_RATE_MS);
    Led_Lo2();
    vTaskDelay(301/portTICK_RATE_MS);
  }
}

/* Writes each 500 ms */
void vPrintUart(void* dummy)
{portTickType last_wakeup_time;
 last_wakeup_time = xTaskGetTickCount();
```

```
 while(1){uart_puts("Hello World\r\n");
    vTaskDelayUntil(&last_wakeup_time, 500/portTICK_RATE_MS);
   }
}

xQueueHandle qh = 0;

void task_tx(void* p)
{int myInt = 0;
  while(1)
    {myInt++;
     if(!xQueueSend(qh, &myInt, 100))
        {uart_puts("Failed to send item to queue within 500ms");
        }
     vTaskDelay(1000);
    }
}

void task_rx(void* p)
{char c[10];
 int myInt = 0;
 while(1)
    {if(!xQueueReceive(qh, &myInt, 1000))
       {uart_puts("Failed to receive item within 1000 ms");
       }
     else {c[0]='0'+myInt;c[1]=0;
           uart_puts("Received: ");uart_puts(c);uart_puts("\r\n");
          }
    }
}

int main()
{Led_Init();
 Usart1_Init();

  qh = xQueueCreate(1, sizeof(int));

//  activer ces fonctions fait atteindre le timeout de transfert de donnees dans la queue
//    if (!(pdPASS == xTaskCreate( vLedsFloat, ( signed char * ) "LedFloat", 128, NULL, 2, NULL ))) goto hell;
//    if (!(pdPASS == xTaskCreate( vLedsFlash, ( signed char * ) "LedFlash", 128, NULL, 2, NULL ))) goto hell;
//    if (!(pdPASS == xTaskCreate( vPrintUart, ( signed char * ) "Uart",     128, NULL, 2, NULL ))) goto hell;

   xTaskCreate(task_tx, (signed char*)"t1", (128), 0, 2, 0);
   xTaskCreate(task_rx, (signed char*)"t2", (128), 0, 2, 0);
   vTaskStartScheduler();
hell: while(1) {};
    return 0;
}
```

tp_freertos/2message_passing/src/main.c

1. **ajouter les tâches commentées et observer la conséquence en terme de gestion du temps par l'ordonnanceur.**

1.3 Protéger les échanges de données

Le danger de partager des données entre tâches tient en la cohérence des informations. Si une première tâche est en train de manipuler les données requises par une seconde tâche, les valeurs contenues dans les variables peuvent devenir incohérente. Une méthode pour éviter ce problème est de garantir que *un seul processus* peut accéder à une variable à un instant donné : accès MUTually EXclusive ou méthode *mutex*. Chaque accès à une variable est encadrée par un *mutex* et l'exécution de la tâche n'est possible que si le *mutex* est débloqué.

```
#include "FreeRTOS.h"
#include "task.h"
#include "semphr.h"
#include "common.h"
#include "stdlib.h" // rand

int global=0;
```

```
// xSemaphoreHandle xMutex;

void vLedsFloat(void* dummy);
void vLedsFlash(void* dummy);
void vPrintUart(void* dummy);

void task_rx(void* p)
{   char aff[10];
    char *t=(char*)p;
    int myInt = 0;
    volatile int local;
    for (myInt=0;myInt<8;myInt++)
    {
//    xSemaphoreTake( xMutex, portMAX_DELAY );
        local=global;
        local++;
        uart_puts(t);
// cette operation nous fait perdre beaucoup de temps ... il y a toutes les
// chances pour que la seconde tache se lance pendant cet intervalle de temps ... mais la seconde tache
// va cherche global qui n'a pas encore ete incremente' et le resultat sera errone' puisque les 16 taches
// auront fait une somme de 8
        global=local;

//    xSemaphoreGive( xMutex );
    vTaskDelay( ( rand() & 0x5 ) );  // essayer de deplacer le delay sous cette '}' pour ne pas alterner
    }
    aff[0]=' ';aff[1]=global+'0';aff[2]=' ';aff[3]=0;uart_puts(aff);
    while (1) vTaskDelay( ( rand() & 0x5 ) ); // on n'a jamais le droit de finir toutes les taches
}

int main()
{
    Led_Init();
    Usart1_Init();
    srand( 567 );
//  xMutex = xSemaphoreCreateMutex();
    xTaskCreate(task_rx, (signed char*)"t1", 100, "1111111111111111111111111111111111111111111111\r\n\0", 1, 0);
    xTaskCreate(task_rx, (signed char*)"t2", 100, "2222222222222222222222222222222222222222222222\r\n\0", 1, 0);
    vTaskStartScheduler();

hell:
    while(1);
   return 0;
}
```

texte1/main_mutex.c

L'exemple ci-dessus considère une variable globale `globale` qui est manipulée par deux tâches. Chacune de ces tâches effectue un calcul long, illustré ici par un transfert sur le port série (une tâche très longue à l'échelle du temps de calcul : rappeler la durée d'un transfert de 45 caractères au débit de 115200 bauds selon un encodage 8N1 sur bus RS232).

Afin de pallier aux déficiences observées, nous utiliserons un *mutex*.

1. **Décommenter les lignes contenant la déclaration des mutex, et constater la différence de comportement.**

Attention : nous devons informer FreeRTOS de notre intention d'utiliser les `mutex` et donc de charger les fonctionnalités associées. Nous devons pour cela ajouter la configuration `#define configUSE_MUTEXES 1` dans `src/include/FreeRTOSConfig.h`.

1.4 Séquencer les tâches par les sémaphores

Finalement, l'exécution des tâches peut être séquencée sur des conditions d'exécution, ou sémaphores. Si ce mécanisme n'est pas implémenté judicieusement, il peut donner lieu à des bloquages dans lequel un processus de forte priorité attend d'être débloqué par un processus de faible priorité qui n'est jamais appelé : un *deadlock* survient.

```
#include "FreeRTOS.h"
#include "task.h"
#include "semphr.h"
#include "common.h"

```

```
int globale=0;
xSemaphoreHandle event_signal;

void task1(void* p)
{
  while (1) {
//    if (xSemaphoreTake(event_signal,500/portTICK_RATE_MS)==pdFALSE)
//       uart_puts("not available\r\n\0");
//    else
//       uart_puts("sem take\r\n\0");

    while (globale==0) ; // vTaskDelay( 1/portTICK_RATE_MS );
    globale=0;
    uart_puts("sem take\r\n\0");
// ici on demontre que le semaphore est bien fait car il rend la main en cas de blocage,
// contrairement au cas de la variable globale qui, en l'absence de Delay, va bloquer sur
// la tache prioritaire qui ne rend pas la main
  }
}

void task2(void* p)
{
  while (1) {
//    xSemaphoreGive(event_signal); // debloque tache 1
    globale=1;
    uart_puts("sem give\r\n\0");
    vTaskDelay( 700/portTICK_RATE_MS );  // remplacer 400 par 700 !
  }
}

int main()
{
 Usart1_Init();
 uart_puts("depart\r\n\0");
//    vSemaphoreCreateBinary( event_signal );   // Create the semaphore
//    xSemaphoreTake(event_signal, 0);          // Take semaphore after creating it.
 xTaskCreate(task1, (signed char*)"t1", STACK_BYTES(2048), 0, 2, 0);
 xTaskCreate(task2, (signed char*)"t2", STACK_BYTES(2048), 0, 1, 0);
 vTaskStartScheduler();

hell:
    while(1) {};
    return 0;
}
```

texte1/semaphore.c

Cet exemple donne lieu à de nombreuses combinaisons possibles si on se contente d'essayer d'émuler manuellement un sémaphore par une variable globale :

1. Si

   ```
   xTaskCreate(task1, (signed char*)"t1", STACK_BYTES(2048), 0, 2, 0);
   xTaskCreate(task2, (signed char*)"t2", STACK_BYTES(2048), 0, 1, 0);
   ```

   et `while (globale==0) ;` alors on bloque toute exécution car le producteur ne peut pas s'exécuter (`task1`, de priorité forte 2, est prioritaire mais fait la boucle sans attente)

2. la tâche est débloquée en mettant une attente dans `task1` :
   `while (globale==0) vTaskDelay( 1/portTICK_RATE_MS );` qui parfois donne la main à `task2` (qui est de faible priorité)

3. si les priorités sont inversées par

   ```
   xTaskCreate(task1, (signed char*)"t1", STACK_BYTES(2048), 0, 1, 0);
   xTaskCreate(task2, (signed char*)"t2", STACK_BYTES(2048), 0, 2, 0);
   ```

   et `while (globale==0) ;`, seule `task2` (la plus prioritaire) peut s'exécuter (ou `uart_puts()` de `task1` se fait écraser par celui de `task2`?)

Donc pour résumer, dans le cas 1, le consommateur est le plus prioritaire et le producteur n'a jamais le temps de s'exécuter, donc de récuperer les données et les fournir (`task2` devrait par exemple charger des données sur ADC, et `task1` devrait afficher). Dans le cas 2, tout se passe bien car le consommateur rend la main au producteur en s'endormant. Dans le cas 3, le producteur est trop prioritaire donc la donnée n'est jamais consommée.

En passant de la variable globale au sémaphore, l'exécution consommateur/producteur est ordonnancée sans devoir mettre explicitement d'attente pour rendre la main. Il arrive des cas de *timeout* mais au moins toutes les tâches sont appelées.

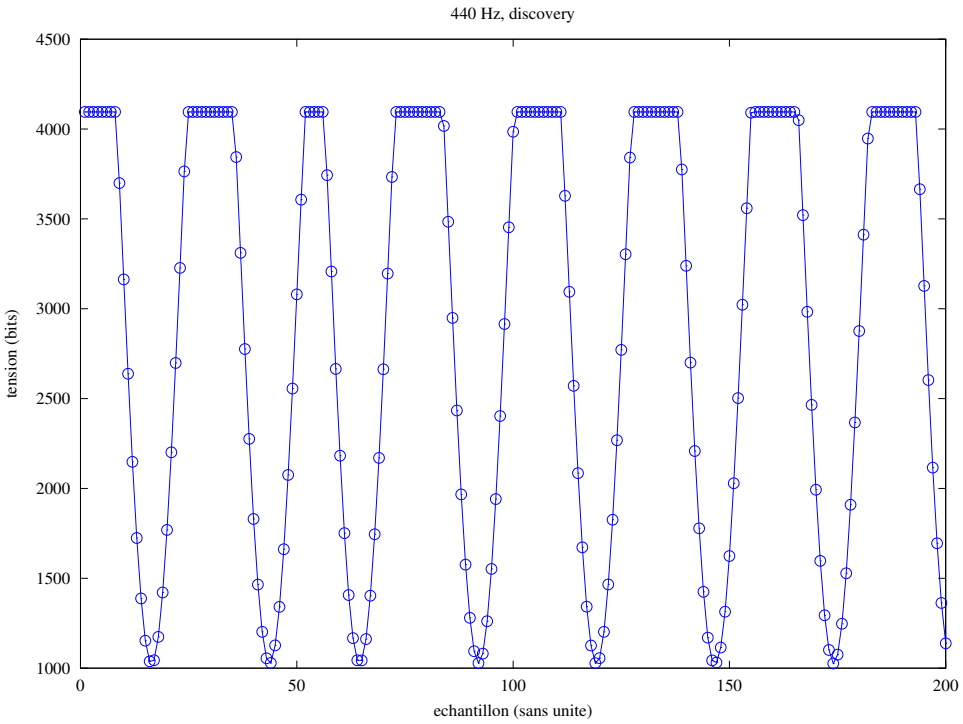
2 Mise en pratique

L'exemple ci-dessous est une simple lecture périodique de température :

```
#include "common.h"
#include <libopencm3/stm32/usart.h>


//#define avec_newlib
#ifdef avec_newlib
#include <stdio.h>
int _write(int file, char *ptr, int len);
#endif

int main(void)
{volatile int i, c = 0;
#ifdef avec_newlib
 volatile float f=2.0;
#endif
  short res;
  Usart1_Init();
  Led_Init();
//  adc_setup();

  while (1) {
//    if (c&0x01) {Led_Hi1();Led_Hi2();} else {Led_Lo1();Led_Lo2();}
//    c = (c == 9) ? 0 : c + 1; // cyclic increment c
    res=read_adc_naiive(1);
#ifndef avec_newlib
    uart_putc('0'); // USART: send byte
    uart_putc('x'); // USART: send byte

    affshort(res);
//    uart_putc(' ');
//    uart_putc(c + '0'); // USART: send byte
    uart_putc('\n'); uart_putc('\r');
    Led_Hi2();
    for (i = 0; i < 800000; i++) __asm__("NOP");
    Led_Lo2();
#else
    printf("%d %f\r\n", (int)res,sqrt(f));
#endif
  }
  return 0;
}
```

temperature/main.c

avec initialisation des ADCs, des horloges associées ainsi que des broches.
La carte son se branche sur l'ADC canal 1 : remplacer la mesure de température par une mesure de tension.
Remplacer la lecture lente de température par une lecture de tension la plus rapide possible.



Pour afficher une séquence de données stockées en format hexadécimal dans un fichier ASCII, on utilisera sous GNU/Octave :

```
f=fopen("fichier");d=fscanf(f,"%x",inf); plot(d);
```

Proposer une lecture périodique de température dans le contexte de FreeRTOS.

3 Problème des philosophes qui dînent

Le problème des philosophes qui dînent est un exemple classique de partage de ressources. Cinq philosophes sont assis autour d'une table ronde dans un restaurant asiatique et disposent de 5 baguettes, une entre chaque philosophe. Afin de pouvoir manger son repas, chaque philosophe a besoin de deux baguettes : celle à sa droite, et celle à sa gauche. Lorsqu'un philosophe peut accéder simultanément aux deux baguettes qui l'entourent, il s'en saisit et mange. Sinon, le philosophe réfléchit, et tentera de manger plus tard.

Comment représenter chaque philosophe dans un programme s'exécutant sous FreeRTOS ?

Comment exprimer le fait qu'une baguette ne peut être utilisée que par un philosophe à la fois ?

```
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#include "semphr.h"
#include "croutine.h"

#define NB_PHILO 5

xSemaphoreHandle xMutex[NB_PHILO];
int mange[NB_PHILO];

int main(void);
void Led_Init(void);
void Usart1_Init(void);
void uart_putc(char c);

void func(void* p)
{ int numero= *(int*) p;
  while (mange[numero]!=1)
    {uart_putc('a'+numero);vTaskDelay(500 / portTICK_RATE_MS);
     if (xSemaphoreTake(xMutex[numero],500/portTICK_RATE_MS)==pdFALSE)
```

```
        {uart_puts("not available\r\n\0");
23       xSemaphoreGive( xMutex[numero] );
        }
25    xSemaphoreTake( xMutex[(numero+1)%NB_PHILO], portMAX_DELAY );
      uart_putc('A'+numero);vTaskDelay(500 / portTICK_RATE_MS);
27    xSemaphoreGive( xMutex[numero] );
      xSemaphoreGive( xMutex[(numero+1)%NB_PHILO] );
29    uart_putc('0'+numero);
      mange[numero]=1;
31   }
  while (1) { vTaskDelay(100 / portTICK_RATE_MS); }; // on n'a jamais le droit de finir toutes les taches
33 }

35 int main()
{
37 //www.freertos.org/FreeRTOS_Support_Forum_Archive/February_2007/freertos_Problems_with_passing_parameters_to_task_1666309.
     html
  static int p[5]={0,1,2,3,4};
39  static char * taskNames[5] = {"P0","P1","P2","P3","P4"};

41  int i;
  Led_Init();
43  Usart1_Init();
  for (i=0;i<NB_PHILO;i++) xMutex[i] = xSemaphoreCreateMutex();
45  for (i=0;i<NB_PHILO;i++)
      {xTaskCreate(func, (const signed char const*)taskNames[i], STACK_BYTES(256), (void*)&p[i],1,0);}
47  vTaskStartScheduler();
  while(1);
49  return 0;
}
```

philosophes.c

Ce programme renvoie la sortie “juste” `eabcdEBnot available14DA03C2` puisque les 5 philosophes pensent, 1 et 4 mangent puis reposent leurs baguettes alors que 3 se fait rejeter sa demande de nourriture, puis 3 et 0 mangent pour finalement laisser 2 s’alimenter

Le problème de *deadlock* apparaît si une lattence existe entre la saisie de deux baguettes. Dans ce cas, chaque philosophe peut avoir pris la baguette à sa gauche et interdire à son voisin d’engager le repas : les ressources sont réservées et la condition de déblocage ne peut être résolue. Cet exemple est illustré en décommentant la constante `deadlock` dans l’exemple de *phthread* : la fonction `usleep()` autorise l’ordonnanceur à passer la main à une autre tâche alors que seule une baguette a été prise, se traduisant par un cas de blocage. Le solution consiste à ce qu’au bout d’un certain temps, le philosophe repose la baguette qu’il a saisie et se remette à penser, pour ré-essayer de finir son repas plus tard. C’est le sens de la séquence

```
if (xSemaphoreTake(xMutex[numero],500/portTICK_RATE_MS)==pdFALSE)
  {uart_puts("not available\r\n\0");
   xSemaphoreGive( xMutex[numero] );
```

qui n’était pas obligatoire si on suppose que les deux requêtes de mutex se font “simultanément” et que l’ordonnanceur de peut interrompre la tâche entre les deux requêtes.

Références

[1] *RM0008 – Reference manual, STM32F101xx, STM32F102xx, STM32F103xx, STM32F105xx and STM32F107xx advanced ARM-based 32-bit MCUs*, (May 2011), Doc ID 13902 Rev 13

[2] R. Barry, Using the FreeRTOS real time kernel – A practical guide, (2009)

[3] J.J. Labrosse, *MicroC OS II : The Real Time Kernel – 2nd Ed.*, CRC Press (2002)

A Initialisation des périphériques

L'initialisation de périphériques et USART par `libopencm3` s'obtient par

```
//#include <libopencm3/stm32/memorymap.h>
#include <libopencm3/stm32/rcc.h>
#include <libopencm3/stm32/gpio.h>
#include <libopencm3/stm32/usart.h>
#include <stdint.h>
#include "common.h"
#ifndef STM32F1
#include "stm32f4_initialisation.h"
#endif

#define usart1        // comment for STM32F4Discovery
//#define avec_newlib

#ifdef avec_newlib
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
int _write(int file, char *ptr, int len);
#endif


void clock_setup(void)
{
#ifdef STM32F1
#ifdef STM32F10X_LD_VL
 rcc_clock_setup_in_hse_8mhz_out_24mhz();  // STM32F100 discovery
#else
 rcc_clock_setup_in_hse_8mhz_out_72mhz();  // STM32F103
#endif
#else  // F4
 // rcc_clock_setup_hse_3v3(&rcc_hse_8mhz_3v3[RCC_CLOCK_3V3_168MHZ]);
  core_clock_setup();
#endif
  rcc_periph_clock_enable(RCC_GPIOC); // Enable GPIOC clock
  rcc_periph_clock_enable(RCC_GPIOD); // Enable GPIOD clock for F4 (LEDs)
  rcc_periph_clock_enable(RCC_GPIOA); // Enable GPIOA clock
#ifdef usart1
  rcc_periph_clock_enable(RCC_USART1);
#else
  rcc_periph_clock_enable(RCC_USART2);
#endif
  rcc_periph_clock_enable(RCC_ADC1);  // exemple ADC
}


void Usart1_Init(void)
{ // Setup GPIO pin GPIO_USART1_TX/GPIO9 on GPIO port A for transmit. */
  clock_setup();
#ifdef STM32F1
#ifdef usart1
  gpio_set_mode(GPIOA, GPIO_MODE_OUTPUT_50_MHZ,
      GPIO_CNF_OUTPUT_ALTFN_PUSHPULL, GPIO_USART1_TX);
#else
  gpio_set_mode(GPIOA, GPIO_MODE_OUTPUT_50_MHZ,
      GPIO_CNF_OUTPUT_ALTFN_PUSHPULL, GPIO_USART2_TX);
#endif
#else
  gpio_mode_setup (GPIOA, GPIO_MODE_AF, GPIO_PUPD_NONE, GPIO9);  //GPA9 : Tx send from STM32 to ext
  gpio_mode_setup (GPIOA, GPIO_MODE_AF, GPIO_PUPD_NONE, GPIO10); //GPD10: Rx recieve from ext to STM32
  gpio_set_af (GPIOA, GPIO_AF7, GPIO9);
  gpio_set_af (GPIOA, GPIO_AF7, GPIO10);
#endif

#ifdef usart1
  usart_set_baudrate(USART1, 115200);
  usart_set_databits(USART1, 8);
```

```
  usart_set_stopbits(USART1, USART_STOPBITS_1);
  usart_set_mode(USART1, USART_MODE_TX);
  usart_set_parity(USART1, USART_PARITY_NONE);
  usart_set_flow_control(USART1, USART_FLOWCONTROL_NONE);
  usart_enable(USART1);  // PA9 & PA10 for USART1
#else
  usart_set_baudrate(USART2, 115200);
  usart_set_databits(USART2, 8);
  usart_set_stopbits(USART2, USART_STOPBITS_1);
  usart_set_mode(USART2, USART_MODE_TX);
  usart_set_parity(USART2, USART_PARITY_NONE);
  usart_set_flow_control(USART2, USART_FLOWCONTROL_NONE);
  usart_enable(USART2);
#endif
}

void Led_Init(void)
{
#ifdef STM32F1
 gpio_set_mode(GPIOC,GPIO_MODE_OUTPUT_2_MHZ,GPIO_CNF_OUTPUT_PUSHPULL,GPIO8|GPIO9|GPIO1|GPIO2|GPIO12);
#else
// gpio_mode_setup(GPIOD, GPIO_MODE_OUTPUT,GPIO_PUPD_NONE, GPIO12|GPIO13|GPIO14|GPIO15);
 gpio_mode_setup(GPIOA, GPIO_MODE_OUTPUT,GPIO_PUPD_NONE, GPIO11|GPIO12|GPIO13);
#endif
}

#ifdef STM32F1
void Led_Hi1(void) {gpio_set  (GPIOC, GPIO9);gpio_set  (GPIOC, GPIO2);gpio_set  (GPIOC, GPIO12);}
void Led_Lo1(void) {gpio_clear(GPIOC, GPIO9);gpio_clear(GPIOC, GPIO2);gpio_clear(GPIOC, GPIO12);}
void Led_Hi2(void) {gpio_set  (GPIOC, GPIO8);gpio_set  (GPIOC, GPIO1);}
void Led_Lo2(void) {gpio_clear(GPIOC, GPIO8);gpio_clear(GPIOC, GPIO1);}
#else
//void Led_Hi1(void) {gpio_set  (GPIOD, GPIO12);}
//void Led_Lo1(void) {gpio_clear(GPIOD, GPIO12);}
//void Led_Hi2(void) {gpio_set  (GPIOD, GPIO13);}
//void Led_Lo2(void) {gpio_clear(GPIOD, GPIO13);}
void Led_Hi1(void) {gpio_set  (GPIOA, GPIO12);}
void Led_Lo1(void) {gpio_clear(GPIOA, GPIO12);}
void Led_Hi2(void) {gpio_set  (GPIOA, GPIO13);}
void Led_Lo2(void) {gpio_clear(GPIOA, GPIO13);}
#endif

// define newlib stub
#ifdef avec_newlib
int _write(int file, char *ptr, int len)
{ int i;
  if (file == STDOUT_FILENO || file == STDERR_FILENO) {
     for (i = 0; i < len; i++) {
         if (ptr[i] == '\n')
#ifdef usart1
              usart_send_blocking(USART1, '\r');
#else
              usart_send_blocking(USART2, '\r');
#endif
#ifdef usart1
         usart_send_blocking(USART1, ptr[i]);
#else
         usart_send_blocking(USART2, ptr[i]);
#endif
       }
      return i;
  }
  errno = EIO;
  return -1;
}
#endif

void uart_putc(char c) {
#ifdef usart1
usart_send_blocking(USART1, c); // USART1: send byte
```

```
#else
usart_send_blocking(USART2, c); // USART2: send byte
#endif
}

/* Writes a zero teminated string over the serial line*/
void uart_puts(char *c) {while(*c!=0) uart_putc(*(c++));}
```

Listing 1 – Bibliothèque `libopencm3`