OS temps-réel sur STM32

J.-M Friedt, 2 février 2018

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 (stm32 pour la bibliothèque fournie par ST Microelectronics, cm3 pour le support `libopencm3` [2]). Nous travaillerons sur l'archive de FreeRTOS version 9 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 Premiers pas avec le STM32

Le STM32 est un processeur ARM de la classe Cortex-M3 [1]. Ce cœur se décline de la gamme faible coût aux performances modestes, jusqu'au haut de gamme comportant des interfaces ethernet et USB.

Bien que FreeRTOS [2] fournisse un certain nombre de niveaux d'abstraction [3] pour fournir des méthodes s'apparentant au multitâche, les fonctions de bas niveau telles que la communication asynchrone (RS232) ou synchrone (SPI/I^2C) restent à la charge du développeur, quitte à utiliser les fonctionnalités fournies par une bibliothèque annexe (`libstm32` ou `libopencm3`).

Nous nous engageons donc dans un premier temps sur un petit exercice de communication sur bus asynchrone afin de nous familiariser avec le microcontrôleur et en particulier une de ses subtilités qui tient dans le routage des horloges sans lesquelles un périphérique ne peut pas fonctionner. L'architecture de la carte est résumée dans la Fig. 1 : on y trouvera la référence du port de communicatin asynchrone (USART), LEDs et convertisseur analogique-numérique (ADC) accessibles.

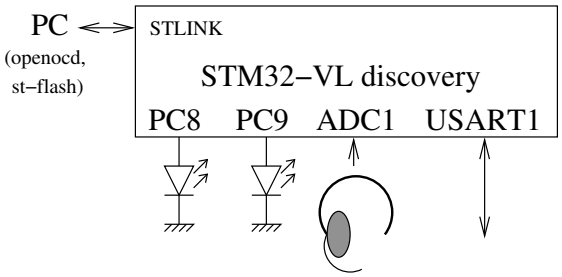


FIGURE 1 – Connections du STM32VL Discovery vers le monde extérieur.

Ces diverses fonctions – initialisation des horloges, des périphériques, accès bas niveau – sont placé dans un répertoire contenant les diverses bibliothèques que nous utiliserons dans nos projets FreeRTOS. Comme diverses cartes de développement proposent des ports de communications différents ou des boutons poussoirs/LEDs sur des ports différents, nous abstrai-

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`

rons l'allumage/l'extinction de LED ou la communication dans ce même répertoire en fournissant des méthodes génériques (`common.h`) dont l'implémentation dépend de chaque plateforme de travail.

2 Premiers pas : compiler et flasher

L'exemple que nous nous proposons dans un premier temps d'expérimenter est le classique clignotement de diode et envoi de trames de texte sur le port série. En particulier, l'objectif est d'illustrer l'initialisation des ports (entrée/sortie, fonction GPIO ou étendue) et surtout la sélection des horloges cadençant chaque périphérique (code 1). Dans cet exemple, nous utilisons les deux bibliothèques simplifiant l'accès aux ressources du STM32 (en évitant dans un premier temps de consulter les 1093 pages du manuel) : `libopencm3`[3] et `libstm32`. Bien que l'exemple sur le site de FreeRTOS[4] se lie avec bibliothèque de ST Microelectronics `libstm32`, nous avons traduit le code pour se lier à `libopencm3` et ainsi obtenir automatiquement le support pour STM32F4.

Afin de réutiliser un maximum de code entre le programme C autonome et FreeRTOS, nous exploitons la compilation séparée pour le code d'initialisation (réutilisable) et la charge utile (spécifique à chaque application).

Les codes d'initialisation sont fournis en annexe A.

⚠ L'initialisation des liaisons asynchrones en autre chose que N (ie O ou E) nécessite de définir des mots de *9 bits* et non de 8!

Une fois ces initialisations effectuées, envoyer un message sur USART et faire clignoter une diode s'obtient en `libopencm3` par

```
#include "common.h"

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

int main(void)
{ int i, c = 0;
  Usart1_Init();
  Led_Init();
  while (1) {
    if (c&0x01) {Led_Hi1();Led_Hi2();} else {Led_Lo1();Led_Lo2();}
    c = (c == 9) ? 0 : c + 1; // cyclic increment c
#ifndef avec_newlib
    uart_putc(c + '0'); // USART: send byte
    uart_puts("\r\n\0");
#else
    printf("%d\r\n", (int)c);
#endif
    for (i = 0; i < 800000; i++) __asm__("NOP");
  }
  return 0;
}
```

Listing 1 – Exemple en C exploitant `libopencm3`

ou par `libstm32` au moyen de

```
#include "common.h"

// Function Declarations
int main(void)
{ int i, c = 0,k=0;
//  clock_setup();
  Led_Init();
  Usart1_Init(); //  usart_setup();
  while (1) {
    if (k==1) Led_Hi1(); else Led_Lo1();
```

3. `http://libopencm3.org/wiki/Main_Page` mais le plus simple est de sélectionner cette bibliothèque lors de la compilation de sa *toolchain* par `https://github.com/vedderb/summon-arm-toolchain/`
4. `http://www.freertos.org/FreeRTOS-for-Cortex-M3-STM32-STM32F100-Discovery.html`

```
    k=1-k;
    c = (c == 9) ? 0 : c + 1; // cyclic increment c
    uart_putc(c + '0'); // USART1: send byte
    uart_putc('\r');
    uart_putc('\n');
    for (i = 0; i < 80000; i++) __asm__("NOP");
  }
  return 0;
}
```

Listing 2 – Exemple en C exploitant `libstm32`

Ayant validé la compilation de ces programmes de base et le lien avec les bibliothèques appropriées au moyen du `Makefile`

```
all: usart_cm3.bin usart_stm32.bin pwm_cm3.bin

#libopencm3 examples
main_cm3.o: main_cm3.c
  arm-none-eabi-gcc -Wall -I../common -fno-common -mthumb -mcpu=cortex-m3 \
            -msoft-float -MD -DSTM32F1 -DSTM32F10X_MD -c main_cm3.c

usart_cm3.o: ../common/usart_opencm3.c
  arm-none-eabi-gcc -Wall -I../common -fno-common -mthumb -mcpu=cortex-m3 \
            -msoft-float -MD -DSTM32F1 -DSTM32F10X_MD -o usart_cm3.o -c ../common/usart_opencm3.c

usart_cm3.bin: main_cm3.o usart_cm3.o
  arm-none-eabi-gcc -o usart_cm3.elf usart_cm3.o main_cm3.o  \
      --static -T../ld/stm32vl-discovery.ld -nostartfiles -Wl,--gc-sections -mthumb \
            -mcpu=cortex-m3 -msoft-float -mfix-cortex-m3-ldrd -lopencm3_stm32f1
  arm-none-eabi-objcopy -O binary usart_cm3.elf usart_cm3.bin

pwm_cm3.o: pwm_cm3.c
  arm-none-eabi-gcc -Wall -I../common \
            -fno-common -mthumb -mcpu=cortex-m3 -msoft-float -MD -DSTM32F1 \
            -o pwm_cm3.o -c pwm_cm3.c

pwm_cm3.bin: pwm_cm3.o usart_cm3.o
  arm-none-eabi-gcc -o pwm_cm3.elf pwm_cm3.o usart_cm3.o -lopencm3_stm32f1 --static \
            -T../ld/stm32vl-discovery.ld -nostartfiles -Wl,--gc-sections -mthumb \
            -mcpu=cortex-m3 -msoft-float -mfix-cortex-m3-ldrd
  arm-none-eabi-objcopy -O binary pwm_cm3.elf pwm_cm3.bin

#LDSCRIPT= ../ld/stm32_flash.ld
LDSCRIPT= ../ld/stm32vl-discovery.ld

usart_stm32.o: ../common/usart_stm32.c
  arm-none-eabi-gcc -DSTM32F10X_MD -I./ -c -fno-common -O0 -g3 -I../common \
            -I/home/jmfriedt/sat/arm-none-eabi/include/stm32f1/ -I/home/jmfriedt/sat/arm-none-eabi/include/ \
            -Wall -mcpu=cortex-m3 -mthumb -fno-common -mthumb -msoft-float -c -o usart_stm32.o ../common/usart_stm32.c

main_stm32.o: main_stm32.c
  arm-none-eabi-gcc -DSTM32F10X_MD -I./ -c -fno-common -O0 -g3 -I../common \
            -I/home/jmfriedt/sat/arm-none-eabi/include/stm32f1/ -I/home/jmfriedt/sat/arm-none-eabi/include/ \
            -Wall -mcpu=cortex-m3 -mthumb -fno-common -mthumb -msoft-float main_stm32.c

usart_stm32.bin: main_stm32.o usart_stm32.o
  arm-none-eabi-gcc -fno-common -O0 -g3 -Wall -mcpu=cortex-m3 -mthumb -msoft-float \
            -Wl,--gc-sections -T$(LDSCRIPT) -nostartfiles -o usart_stm32.elf \
            main_stm32.o usart_stm32.o -lstm32 -lopencm3_stm32f1
  arm-none-eabi-objcopy -O binary usart_stm32.elf usart_stm32.bin

clean:
  rm *.o *.bin *.d *.elf ../common/*.o

flash: usart_cm3.bin
  stm32flash.sh -w usart_cm3.bin /dev/ttyUSB0
```

Listing 3 – Makefile associé pour `libopencm3` et `libstm32`

Techniquement, il est possible de directement charger et exécuter le logiciel depuis le gestionnaire de lien JTAG[5] `openocd` au moyen de

```
openocd -s /usr/local/share/openocd/scripts/ -f board/stm32vldiscovery.cfg \
-c "init" -c "reset init" -c "stm32f1x mass_erase 0" \
-c "flash write_image ‘pwd‘/output/main.elf" -c "reset"
```

Cependant, cette approche a le mauvais goût de faire planter le *bootloader* après cette transaction et de nécessiter de débrancher/rebrancher le câble USB afin de relancer le Discovery Board.

Une alternative consiste à passer par `gdb`, qui en plus nous donne accès aux outils de déverminage associés (affichage de la pile, des valeurs des variables ...). Dans un premier terminal nous lançons
`openocd -s /usr/local/share/openocd/scripts/ -f board/stm32vldiscovery.cfg` qui lance le serveur. Dans un second terminal, le client `gdb` pour la cible appropriée est lancé avec pour argument le programme (au format ELF, idéalement avec les symboles de déverminage compilés par `-g`) et le programme est chargé par
`arm-none-eabi-gdb output/main.elf`

Une fois `gdb` lancé, nous le connectons au client et chargeons le programme par

```
target remote localhost:3333
load
continue
```

En particulier, l'intérêt de `gdb` est de permettre d'interrompre l'exécution du programme pour observer l'état de la pile (`bt`), afficher une variable (`print c`) ou placer des points d'arrêt (`break fonction`, par exemple `break uart_putc`).

En cas d'erreur de liaison, vérifier que la carte de développement n'est pas considérée comme un périphérique de stockage de masse : `dmesg` doit indiquer `usb-storage : device ignored`. Dans le cas contraire, ajouter une condition au chargement du module `usb-storage` en créant le fichier `/etc/modprobe.d/usb-storage.conf` contenant `options usb-storage quirks=483:3744:i`

1. **Faire clignoter alternativement les deux LEDs (PC8 et PC9). Varier la vitesse de clignotement.**
2. **Afficher la chaîne de caractères** `Hello World` **au moyen d'une fonction prenant en entrée un tableau de caractères.**

Nous profitons de cet exemple pour sensibiliser sur l'impact d'utiliser inconsidérément une bibliothèque. Avec l'utilisation de `newlib` pour une fonctionnalité telle que `printf`, la taille du code résultant est
`-rwxr-xr-x 1 jmfriedt jmfriedt  29136 Aug 10 13:00 usart.bin`

Nous obtenons exactement le même résultat en écrivant à la main les fonctions de gestion de l'affichage de chaînes de caractères, et en évitant `newlib` nous réduisons la taille à
`-rwxr-xr-x 1 jmfriedt jmfriedt  1620 Aug 10 13:00 usart.bin`

L'utilisation de FreeRTOS n'affranchit donc pas de lire une *datasheet* et s'approprier les méthodes d'accès aux périphériques. Les exemples fournis dans `https://github.com/libopencm3/libopencm3-examples` permettent d'aborder sereinement cette architecture.

3 Outils d'émulation et de déverminage

Nous avons vu que `gdb` permet de sonder les ressources du microcontrôleur pendant l'exécution du programme. Afin de tester du logiciel sur du matériel inexistant ou en cours de développement, une alternative aux cartes de développement consiste à exécuter le programme sur un émulateur. Celle fonctionnalité est par exemple fournie par `qemu`. Par défaut, `qemu` supporte l'architecture ARM mais pas le STM32. Une configuration de `qemu` pour STM32 est proposée à `https://github.com/beckus/qemu_stm32` qui se compile par `./configure --enable-debug --target-list="arm-softmmu" && make`
La configuration ainsi proposée de `qemu` ne supporte qu'un port de communication série asynchrone (UART), et il s'agit de UART2. Nous ajoutons donc le support pour UART1 en modifiant le fichier `hw/arm/stm32_p103.c` pour y ajouter

```
DeviceState *uart1 = DEVICE(object_resolve_path("/machine/stm32/uart[1]", NULL));
assert(uart1);
stm32_uart_connect((Stm32Uart *)uart1,serial_hds[0],STM32_USART1_NO_REMAP);
```

5. vérifier que la carte Discovery n'est pas reconnue comme un périphérique de stockage de mass – `dmesg` doit indiquer `usb-storage 1-1:1.0: device ignored`. Il est envisageable que `openocd` nécessite les droits d'administration pour accéder au port USB. Ne pas hésiter à re-essayer plusieurs fois la connexion en cas d'échec.

Dans sa version la plus simple, `qemu` exécute le programme. Nous pouvons demander à ce que les *deux* ports série soient redirigés vers la sortie standard :

```
qemu-system-arm -M stm32-p103 -serial stdio -serial stdio -kernel main.bin
```

mais plus subtil, `qemu` peut se comporter comme serveur `gdb`. Avec l'option `-s`, `qemu` attend une connexion `gdb` sur le port 1234 :

```
qemu-system-arm -M stm32-p103 -s -S -serial stdio -kernel main.bin
```

suivi dans un second terminal du lancement de `arm-none-eabi-gdb main.elf` qui se configure par

```
target remote localhost:1234
continue
```

4 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 de `FreeRTOSv9.0.0/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.

4.1 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).

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

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){
```

```
16    Led_Hi2();
      vTaskDelay(301/portTICK_RATE_MS);
18    Led_Lo2();
      vTaskDelay(301/portTICK_RATE_MS);
20   }
  }

22
  /* Writes each 500 ms */
24 void vPrintUart(void* dummy)
  {portTickType last_wakeup_time;
26  last_wakeup_time = xTaskGetTickCount();
   // while (1) {}
28  while(1){uart_puts("Hello World\r\n");
       vTaskDelayUntil(&last_wakeup_time, 500/portTICK_RATE_MS);
30    }
  }

32
  int main(void){
34  volatile int i;
   Usart1_Init(); // inits clock as well
36  Led_Init();
   Led_Hi1();

38
   if (!(pdPASS == xTaskCreate( vLedsFloat, (signed char*) "LedFloat",64,NULL,1,NULL ))) goto hell;
40 if (!(pdPASS == xTaskCreate( vLedsFlash, (signed char*) "LedFlash",64,NULL,2,NULL ))) goto hell;
   if (!(pdPASS == xTaskCreate( vPrintUart, (signed char*) "Uart",    64,NULL,3,NULL ))) goto hell;
42
   vTaskStartScheduler();
44 hell:              // should never be reached
    while(1);
46    return 0;
  }
```

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.

4.2 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"
#include <stm32/gpio.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");
       }
```

```
58      else {c[0]='0'+myInt;c[1]=0;
              uart_puts("Received: ");uart_puts(c);uart_puts("\r\n");
60          }
        }
62  }

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

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

70  //  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;
72  //    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;
74
      xTaskCreate(task_tx, (signed char*)"t1", (128), 0, 2, 0);
76    xTaskCreate(task_rx, (signed char*)"t2", (128), 0, 2, 0);
      vTaskStartScheduler();
78  hell: while(1) {};
       return 0;
80  }
```

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.**

4.3 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.

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

7   int global=0;
    // xSemaphoreHandle xMutex;
9
    void vLedsFloat(void* dummy);
11  void vLedsFlash(void* dummy);
    void vPrintUart(void* dummy);
13
    void task_rx(void* p)
15  {   char aff[10];
        char *t=(char*)p;
17      int myInt = 0;
        volatile int local;
19      for (myInt=0;myInt<8;myInt++)
        {
21  //    xSemaphoreTake( xMutex, portMAX_DELAY );
            local=global;
23          local++;
            uart_puts(t);
25  // 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
27  // 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
29          global=local;

31  //    xSemaphoreGive( xMutex );
        vTaskDelay( ( rand() & 0x5 ) );  // essayer de deplacer le delay sous cette '}' pour ne pas alterner
```

```
33      }
        aff[0]=' ';aff[1]=global+'0';aff[2]=' ';aff[3]=0;uart_puts(aff);
35      while (1) vTaskDelay( ( rand() & 0x5 ) ); // on n'a jamais le droit de finir toutes les taches
    }
37
    int main()
39  {
        Led_Init();
41      Usart1_Init();
        srand( 567 );
43  //  xMutex = xSemaphoreCreateMutex();
        xTaskCreate(task_rx, (signed char*)"t1", 100, "1111111111111111111111111111111111111111\r\n\0", 1, 0);
45      xTaskCreate(task_rx, (signed char*)"t2", 100, "2222222222222222222222222222222222222222\r\n\0", 1, 0);
        vTaskStartScheduler();
47
    hell:
49      while(1);
       return 0;
51  }
```

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`.

4.4 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.

```
1   #include "FreeRTOS.h"
    #include "task.h"
3   #include "semphr.h"
    #include "common.h"
5
    int globale=0;
7   xSemaphoreHandle event_signal;

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

17      while (globale==0) ; // vTaskDelay( 1/portTICK_RATE_MS );
        globale=0;
19      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,
21  // 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
23    }
    }
25
    void task2(void* p)
27  {
      while (1) {
29  //    xSemaphoreGive(event_signal); // debloque tache 1
        globale=1;
```

```
31      uart_puts("sem give\r\n\0");
        vTaskDelay( 700/portTICK_RATE_MS );  // remplacer 400 par 700 !
33    }
  }
35
  int main()
37  {
   Usart1_Init();
39  uart_puts("depart\r\n\0");
  //    vSemaphoreCreateBinary( event_signal );   // Create the semaphore
41  //    xSemaphoreTake(event_signal, 0);          // Take semaphore after creating it.
   xTaskCreate(task1, (signed char*)"t1", STACK_BYTES(2048), 0, 2, 0);
43  xTaskCreate(task2, (signed char*)"t2", STACK_BYTES(2048), 0, 1, 0);
   vTaskStartScheduler();
45
  hell:
47      while(1) {};
        return 0;
49  }
```

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.

5 Mise en pratique

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

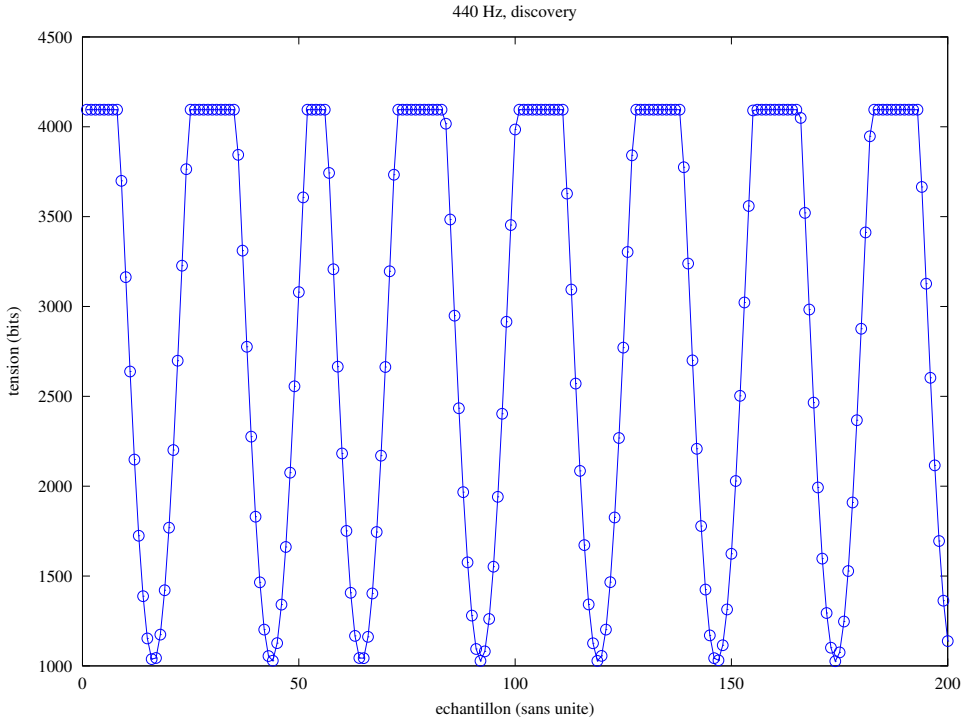
```
1  #include "stm32/gpio.h"
   #include "stm32/usart.h"
3  #include <stm32/rcc.h>
   #include <cmsis/stm32.h>
5  #include <stm32/flash.h>
   #include <stm32/misc.h>
7  #include <stm32/adc.h>
   #include <cmsis/stm32.h>
9  #include <stdlib.h>
   #include <string.h>
11
```

```
uint16_t readADC(void);
void initADC(void);

ADC_InitTypeDef ADC_InitStructure;

unsigned short readADC1(unsigned char channel)                    // jmf convert ADC1
{ADC_RegularChannelConfig(ADC1,channel,1,ADC_SampleTime_239Cycles5);
 ADC_SoftwareStartConvCmd(ADC1,ENABLE);
 while (ADC_GetFlagStatus(ADC1,ADC_FLAG_EOC)==RESET); // EOC is set @ end of cv
 return(ADC_GetConversionValue(ADC1));
}

void writeHEXc(unsigned char ptr)
{unsigned char b;
 b=((ptr&0xf0)>>4);
 if (b<10) uart_putc(b+48); else uart_putc(b+55);
 b=((ptr&0x0f));
 if (b<10) uart_putc(b+48); else uart_putc(b+55);
}

void writeHEXi(uint16_t val)
{writeHEXc(val>>8);
 writeHEXc(val&0xff);
}

int main(void)
{int i=0;
 unsigned short tempe;
  GPIO_InitTypeDef GPIO_InitStructure;

  Led_Init();
  Usart1_Init();
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_AFIO | RCC_APB2Periph_ADC1 ,ENABLE);

  /* Configure USART1 RX (PA.10) as input floating */
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_10;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
  GPIO_Init(GPIOA, &GPIO_InitStructure);

  // Configure ADC1
  ADC_DeInit(ADC1);

  ADC_InitStructure.ADC_Mode=ADC_Mode_Independent;
  ADC_InitStructure.ADC_ScanConvMode=DISABLE;
  ADC_InitStructure.ADC_ContinuousConvMode=DISABLE;
  ADC_InitStructure.ADC_ExternalTrigConv=ADC_ExternalTrigConv_None;
  ADC_InitStructure.ADC_DataAlign=ADC_DataAlign_Right;
  ADC_InitStructure.ADC_NbrOfChannel=1;

  ADC_Init(ADC1, &ADC_InitStructure);
  ADC_Cmd(ADC1, ENABLE);

  ADC_TempSensorVrefintCmd(ENABLE);
  ADC_ResetCalibration(ADC1);
  while (ADC_GetResetCalibrationStatus(ADC1));
  ADC_StartCalibration(ADC1);
  while (ADC_GetCalibrationStatus(ADC1));

  while (1)
  {
   tempe=readADC1(ADC_Channel_16);
   writeHEXi(tempe); uart_putc('\n');
   for (i = 0; i < 80000; i++) __asm__("NOP");
  }
}
```

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.

6 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)
```

```
20      {uart_putc('a'+numero);vTaskDelay(500 / portTICK_RATE_MS);
         if (xSemaphoreTake(xMutex[numero],500/portTICK_RATE_MS)==pdFALSE)
22          {uart_puts("not available\r\n\0");
             xSemaphoreGive( xMutex[numero] );
24          }
         xSemaphoreTake( xMutex[(numero+1)%NB_PHILO], portMAX_DELAY );
26       uart_putc('A'+numero);vTaskDelay(500 / portTICK_RATE_MS);
         xSemaphoreGive( xMutex[numero] );
28       xSemaphoreGive( xMutex[(numero+1)%NB_PHILO] );
         uart_putc('0'+numero);
30       mange[numero]=1;
        }
32    while (1) { vTaskDelay(100 / portTICK_RATE_MS); }; // on n'a jamais le droit de finir toutes les taches
}
34
int main()
36 {
//www.freertos.org/FreeRTOS_Support_Forum_Archive/February_2007/freertos_Problems_with_passing_parameters_to_task_1666309.
    html
38    static int p[5]={0,1,2,3,4};
      static char * taskNames[5] = {"P0","P1","P2","P3","P4"};
40
      int i;
42    Led_Init();
      Usart1_Init();
44    for (i=0;i<NB_PHILO;i++) xMutex[i] = xSemaphoreCreateMutex();
      for (i=0;i<NB_PHILO;i++)
46        {xTaskCreate(func, (const signed char const*)taskNames[i], STACK_BYTES(256), (void*)&p[i],1,0);}
      vTaskStartScheduler();
48    while(1);
      return 0;
50 }
```

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"
void clock_setup(void);

//#define usart1
//#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
 rcc_clock_setup_hse_3v3(&rcc_hse_8mhz_3v3[RCC_CLOCK_3V3_168MHZ]);
#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);
#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);}
#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 4 – Bibliothèque `libopencm3`

tandis que les mêmes fonctions s'obtiennent par `libstm32` au moyen de

```
/* Include STM32 Firmware Lib Headers */
#include "common.h"
#include "stm32/usart.h"
#include "stm32/gpio.h"
#include "stm32/rcc.h"

/* Inits USART1 (serial line) */
void Usart1_Init(void)
{USART_InitTypeDef        usart_i;
 USART_ClockInitTypeDef   usart_c;
 GPIO_InitTypeDef       gpio_i;

 RCC_APB2PeriphClockCmd( RCC_APB2Periph_AFIO,    ENABLE);
 GPIO_PinRemapConfig(GPIO_Remap_USART1,DISABLE);
  /* Enable needed clocks */
 RCC_APB2PeriphClockCmd( RCC_APB2Periph_USART1, ENABLE);
 RCC_APB2PeriphClockCmd( RCC_APB2Periph_GPIOA,  ENABLE);

  /* TX - GPIO*/
 gpio_i.GPIO_Pin    = GPIO_Pin_9;
 gpio_i.GPIO_Speed = GPIO_Speed_50MHz;
 gpio_i.GPIO_Mode   = GPIO_Mode_AF_PP;
 GPIO_Init( GPIOA, &gpio_i);

  /* RX - GPIO*/
 gpio_i.GPIO_Pin    = GPIO_Pin_10;
 gpio_i.GPIO_Speed = GPIO_Speed_50MHz;
 gpio_i.GPIO_Mode   = GPIO_Mode_IN_FLOATING;
 GPIO_Init( GPIOA, &gpio_i);

  /* Configure UART 115200-8N1 */
 usart_i.USART_BaudRate                = 115200;
 usart_i.USART_HardwareFlowControl     = USART_HardwareFlowControl_None;
 usart_i.USART_Mode          = USART_Mode_Rx | USART_Mode_Tx;
 usart_i.USART_Parity              = USART_Parity_No;
 usart_i.USART_StopBits              = USART_StopBits_1;
 usart_i.USART_WordLength              = USART_WordLength_8b;

 usart_c.USART_Clock       = USART_Clock_Enable;
 usart_c.USART_CPHA      = USART_CPHA_1Edge;
 usart_c.USART_CPOL      = USART_CPOL_Low;
 usart_c.USART_LastBit       = USART_LastBit_Disable;

  /* Write configuration to registers */
 USART_ClockInit(USART1, &usart_c);
 USART_Init(USART1, &usart_i);
 USART_Cmd(USART1,ENABLE);
}

void Led_Hi1(void) {GPIO_SetBits   (GPIOC, GPIO_Pin_8);}
void Led_Lo1(void) {GPIO_ResetBits (GPIOC, GPIO_Pin_8);}
void Led_Hi2(void) {GPIO_SetBits   (GPIOC, GPIO_Pin_9);}
void Led_Lo2(void) {GPIO_ResetBits (GPIOC, GPIO_Pin_9);}

/* Configure the LED GPIOs*/
void Led_Init(void)
{GPIO_InitTypeDef GPIO_InitStructure;

  /* init Clocks */
 RCC_APB2PeriphClockCmd(RCC_APB2Periph_AFIO, ENABLE);
 RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOC, ENABLE);

    /* Configure PC1-PC2 as output push-pull (LED) */
 GPIO_WriteBit(GPIOC,0x00000000,Bit_SET);
 GPIO_InitStructure.GPIO_Pin =  GPIO_Pin_1|GPIO_Pin_2 |GPIO_Pin_8|GPIO_Pin_9;
```

```
 GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
 GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
 GPIO_Init(GPIOC, &GPIO_InitStructure);
}

/* Writes one character over the serial line*/
void uart_putc(char c)
{while (USART_GetFlagStatus(USART1, USART_FLAG_TXE) == RESET) {};
 USART_SendData(USART1, c);
}

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

Listing 5 – Bibliothèque `libstm32`