FreeRTOS : application à la réalisation d'un analyseur de réseau numérique sur STM32

Q. Macé, J.-M Friedt,
Master microsystèmes, instrumentation et robotique (MIR), Univ. de Franche-Comté, Besançon,
21 juin 2017

FreeRTOS est un environnement exécutif à très faible empreinte mémoire (<10 KB) fournissant un ordonnanceur et les mécanismes associés pour le partage de ressources entre tâches. En utilisant une bibliothèque libre supportant les cœurs ARM Cortex, en particulier la gamme STM32, nous rendons FreeRTOS utilisable sur tout microcontrôleur muni d'un tel processeur. Nous appréhendons cet environnement de travail, favorable aux développements collaboratifs, dans le contexte de la réalisation d'un instrument de caractérisation de dispositifs radiofréquences.

1 Introduction

Le développement de sysèmes embarqués a pour vocation de minimiser les ressources requises pour atteindre un objectif donné. Dans ce contexte, la programmation de systèmes embarqués numériques s'accomode mal d'environnements inefficaces qui gaspillent les ressources [1] mises à disposition par le microcontrôleur : le langage C apparaît ainsi comme un compromis optimal entre la qualité du code généré et un niveau d'abstraction suffisamment élevé pour exprimer des algorithmes complexes. En particulier, l'utilisation des pointeurs (`*(type*)adresse=valeur;`) permet de s'approcher au plus près du matériel en manipulant aisément les bus d'adresse, de données et de contrôle qui relient l'unité arithmétique et logique aux périphériques qui l'entourent. Cependant, le C se prête relativement mal aux projets complexes nécessitant la cohabitation harmonieuse de plusieurs développeurs : même s'il est envisageable de définir des interfaces entre les fonctions écrites par plusieurs développeurs et garantir la qualité du code par les tests unitaires associés, une séparation du travail sous forme de tâches communiquant entre elles et échangeant des informations semble plus naturelle. La question porte donc sur les ressources requises pour mettre en place un tel environnement, la garantie de la cohérence des accès concurrents aux ressources induite par la multiplicité des tâches, ainsi que la portabilité du code résultant. Ces arguments sont développés à `www.freertos.org/FAQWhat.html#WhyUseRTOS`. FreeRTOS (`www.freertos.org`) propose une hiérarchie de priorités des tâches et donc la garantie que les tâches de priorités les plus élevées sont pré-emptées avec une latence minimale. Tous les appels bloquants prennent un argument de délai maximum au-delà duquel ils peuvent être débloqués : FreeRTOS respecte donc les préceptes du temps réel de borner les latences et ne jamais bloquer l'exécution d'un programme. Nous verrons cependant que la disponiblité des mécanismes ne garantit pas leur bonne utilisation, et il nous sera possible, par un choix maladroit des priorités de tâches, de bloquer un programme. Nous verrons que le déploiement de FreeRTOS nécessite moins de 8 KB sur la plateforme cible, respectant nos ambitions de systèmes fortement embarqués à ressources réduites.

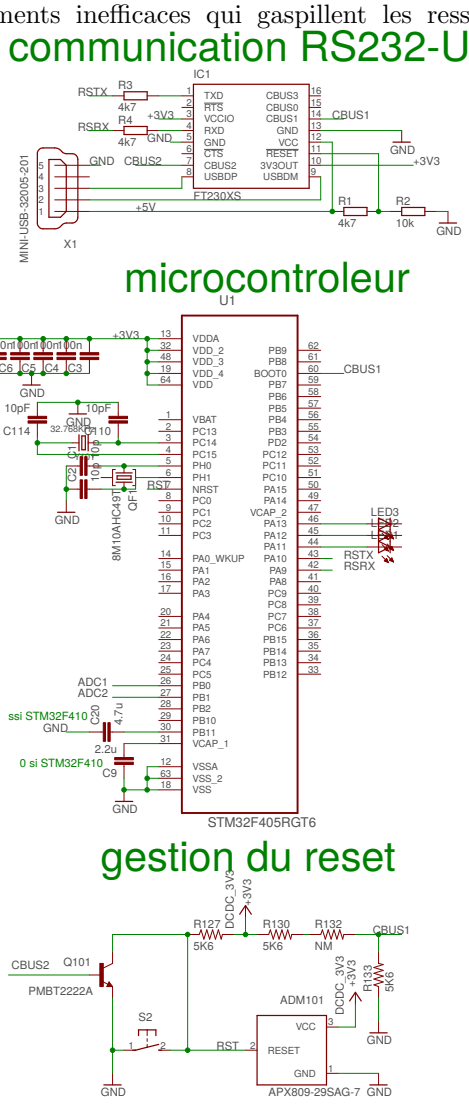

Figure 1: Schéma du circuit

Nous appréhendons ce problème, et la solution que nous lui avons trouvé au travers de l'environnement exécutif FreeRTOS [4], dans le contexte de la réalisation d'un instrument de caractérisation de dispositifs radiofréquences : un analyseur de réseau numérique. Cet instrument classique permet de caractériser la réponse d'un système en fonction de la fréquence, en amplitude et en phase. Nous allons nous intéresser aux gammes radiofréquences qui vont nous permettre de caractériser filtres et résonateurs classiquement utilisés dans le trai-

tement du signal radiofréquence, que ce soit pour filtrer une bande de fréquence donnée (e.g. les diverses bandes de fréquences fournissant les divers services accessibles sur un téléphone portable) ou cadencer des dispositifs synchrones (e.g. oscillateur d'un système numérique). Nous nous focalisons sur un objectif de coût minimum, qui va imposer le choix d'un microcontrôleur qui n'est pas supporté par FreeRTOS, fournissant donc une opportunité de comprendre l'architecture de l'environnement de travail : nous découvrirons que seuls 6 fichiers sont nécessaires pour accéder aux fonctions de FreeRTOS, garantie d'une excellente portabilité puisque toutes les fonctions spécifiques à une architecture donnée sont regroupées dans un unique fichier. Cet objectif de coût réduit quelque peu les ambitions de gamme de fréquence accessible, puisque le synthétiseur de signaux radiofréquences que nous sélectionnons (Analog Devices AD9834) ne sera cadencé qu'à 70 MHz, donnant accès aux gammes de fréquences entre le sub-MHz et une vingtaine de MHz (il est classiquement admis qu'une synthèse numérique de fréquence travaille entre DC et le tiers de sa fréquence de cadencement pour éviter la pollution par des raies parasites). Nous verrons qu'avec un peu d'astuce, ce montage nous permettra néanmoins d'émettre dans la bande FM (88-108 MHz).

La présentation porte donc sur trois objectifs :

1. porter FreeRTOS sur une architecture non-supportée en comprenant quels sont les fichiers nécessaires et leurs rôles, en particulier en nous liant à une bibliothèque libre fournissant les accès au matériel que nous avons sélectionné,
2. démontrer la portabilité du code à diverses plateformes de la même famille en jouant sur le script décrivant l'organisation de la mémoire (*linker script*) et quelques drapeaux de compilation,
3. démontrer les fonctionnalités apportées par FreeRTOS sur une plateforme matérielle mais aussi, pour le lecteur désireux de se familiariser avec cet environnement de développement sans avoir accès au matériel, avec l'émulateur `qemu`.

2 Présentation de FreeRTOS

Notre premier objectif tient donc dans la découverte de FreeRTOS. Rappelons qu'un environnement exécutif a pour vocation de donner au développeur le sentiment de travailler sur un système d'exploitation, ici avec la notion de tâches apparemment exécutées en parallèle et cadencées par un séquenceur. Cependant, l'environnement exécutif n'autorisera par le chargement dynamique de bibliothèques ou d'applications : le binaire monolithique généré à l'issue de la compilation occupe une quantité de ressources connues à la compilation qui permet de garantir la capacité de la plateforme de travail à exécuter ce code. Nous verrons que FreeRTOS offre une multitude de fonctionnalités pour appréhender des problèmes de gestion de la pile, d'ordonnancement des tâches, et surtout pour garantir la cohérence des accès aux ressources en en protégeant l'accès par les mécanismes classiques de sémaphore et leur version binaire que sont les mutex (*MUtually EXclusive*). Ces mécanismes doivent absolument être implémentés aux côtés de l'ordonnanceur pour garantir l'atomicité de leur manipulation et interdire la préemption de la tâche en train de réserver une ressource, au risque de voir le mécanisme de protection échouer.

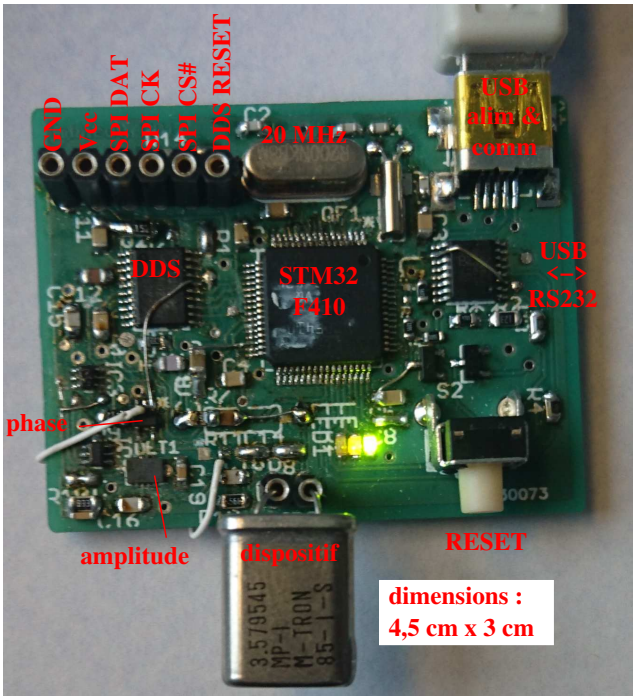

Figure 2: Photographie du circuit assemblé, comprenant un microcontrôleur STM32F410 cadencé à 20 MHz (multiplié en interne à 140 MHz) et les divers périphériques pour réaliser un analyseur de réseau

Nous choisissons de travailler sur microcontrôleur de la gamme STM32 (Figs. 1, 2), une architecture ARM Cortex M3 ou M4 selon la puissance de calcul requise, par soucis de pérennité de l'investissement intellectuel d'appréhender une nouvelle classe de microcontrôleurs. En effet, de nombreux fondeurs choisissent actuellement de décliner ce cœur de processeur, et même si ST abandonne sa fabrication, de nombreuses autres solutions resteront disponibles autour de la même architecture (Atmel SAM3 et SAM4, NXP LPC13xx, TI LM et TM, EFM32 Gecko, Freescale etc ...). Toujours par soucis de portabilité et de liberté, nous désirons ne pas nous lier à une bibliothèque issue d'un fondeur en particulier, même si la disponibilité des codes sources pourrait répondre à nos attentes, mais de nous lier à la bibliothèque libre implémentant un certain nombre de fonctionnalités pour Cortex, `libopencm3` (`libopencm3.org/`).

La question est donc : quel est le travail pour porter FreeRTOS à un nouveau microcontrôleur supporté par une nouvelle bibliothèque ?

Après avoir installé la chaîne de compilation pour ARM Cortex M3 et M4, tel que par exemple décrit à `github.com/jmfriedt/summon-arm-toolchain.git` qui placera `arm-none-eabi-gcc` et ses dépendances dans `$HOME/sat`, nous commençons par télécharger les sources de FreeRTOS, version 9.0.0 à la date de rédaction de cette prose, à `www.freertos.org/a00104.html`. Le contenu de cette archive peut paraître impressionnant au premier abord, jusqu'à ce que nous en comprenions l'organisation et que tout FreeRTOS est contenu dans les 6 fichiers de `FreeRTOS/Source` : `croutine.c`, `event_groups.c`, `list.c`, `queue.c`, `tasks.c`, et `timers.c`. Il nous faudra néanmoins un $7^{ème}$ fichier implémentant la gestion de la mémoire, mécanisme spécifique à chaque architecture de microcontrôleur : nous trouverons notre bonheur dans `FreeRTOS/Source/portable/GCC/ARM_CM*`. FreeRTOS ne fournit aucune abstraction du matériel ou fonctionnalité spécifique à une architecture donnée : nous devrons fournir de telles capacités par l'exploitation de `libopencm3`, et en particulier l'initialisation des périphériques et des horloges, ainsi que les outils de communication.

Les divers exemples proposés dans ce document sont disponibles à `github.com/jmfriedt/tp_freertos/`. Nous faisons l'hypothèse que le lecteur a par ailleurs téléchargé les sources de FreeRTOSv9.0.0 et les a placées au même niveau de l'arborescence que les exemples, tel que résumé dans le schéma de l'arborescence de la Fig. 3. Les divers `Makefile` de nos exemples sont des liens symboliques vers `Makefile.f1_opencm3` (ARM Cortex M3 de type STM32F1) ou `Makefile.f4_opencm3` (ARM Cortex M4 de type STM32F4). Les fichiers `Makefile.f1_libstm32` sont un résidu de support pour la bibliothèque fournie par ST-Microelectronics `libstm32` qui n'a d'intérêt que pour supporter des périphériques exotiques qui ne sont pas supportés par `libopencm3` (par exemple le compteur haute fréquence HRTIM qui équipe certains STM32F3). Le répertoire `common` contient les fonctions de base qui ne sont pas portables et donc pas implémentées par FreeRTOS : initialisation des horloges, initialisation des périphériques, communication ou acquisition de données (conversion analogique-numérique).

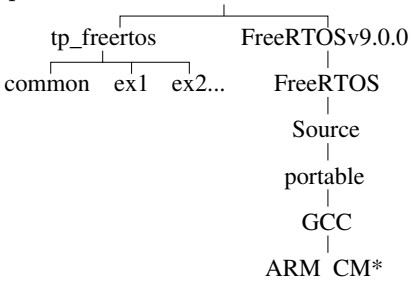

Figure 3: Arborescence du projet

Il devient à cette étape de l'étude nécessaire de définir la déclinaison exacte du microcontrôleur utilisé. Afin de démontrer la compacité de FreeRTOS et sa très faible empreinte mémoire, nous choisissons dans le bas de la gamme de STM32. Par soucis de réduction de coût, nous prenons la solution la moins chère disponible chez Farnell parmi les STM32F4 : le STM32F410 [2] semble un composant de choix, à 3 euros/pièce. Ses 32 KB de RAM et 128 KB de mémoire flash seront plus que suffisant pour l'application que nous envisageons. Le choix de ce microcontrôleur achève de déterminer les fichiers de l'archive de FreeRTOS sur lesquels nous lierons notre application :
`FreeRTOSv9.0.0/FreeRTOS/Source/portable/MemMang/heap_2.c` gèrera le tas dans la mémoire volatile et
`FreeRTOSv9.0.0/FreeRTOS/Source/portable/GCC/ARM_CM4F/port.c` fournit les fonctions spécifiques au Cortex-M4 (avec unité de calcul en virgule flottante).

3 `libopencm3` pour le STM32F4

Notre premier effort consiste à valider la capacité à exécuter des fonctions simples sur ce nouveau microcontrôleur afin de vérifier l'initialisation des périphériques et horloges associées. Ces fonctions seront ensuite utilisées depuis FreeRTOS sous forme de bibliothèque : nous prenons donc soin de séparer les fonctions exploitables par la suite de celles permettant ce test rapide. Ainsi, la fonction principale, qui sera ultérieurement remplacée par les tâches de FreeRTOS séquencées par l'ordonnanceur, ressemble à

```
#include "common.h"

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
    uart_putc(c + '0');          // USART1: send byte
    uart_puts("\r\n\0");
    for (i = 0; i < 800000; i++) __asm__("NOP");
  }
  return 0;
}
```

Listing 1 – Exemple en C d'utilisation de la bibliothèque de fonctions incluse dans le répertoire `common`.

tandis que la bibliothèque de fonctions se résume aux bases telles que les initialisation d'horloge, outils de communication (initialisation du port, mise en forme des variables sous forme de trames ASCII) ou manipulation des ports d'entrée-sortie généralistes (GPIO). Ces fonctions sont regroupées dans `github.com/jmfriedt/tp_freertos/blob/master/common/usart_opencm3.c`, avec une utilisation peut être abusive des conditions de compilation (`#ifdef`) pour tenir compte des diverses architectures supportées.

Nous supportons en effet de cette façon, en fonction des drapeaux passés lors de la compilation, trois déclinaisons du STM32 : le STM32F100 [3] qui équipe la carte STM32VL-Discovery (drapeaux `-DSTM32F1 -DSTM32F10X_LD_VL` pour la carte décrite à `www.st.com/en/evaluation-tools/stm32vldiscovery.html`), le STM32F103 qui équipe par exemple l'Olimex STM32-P103 décrite à `www.olimex.com/Products/ARM/ST/STM32-P103/` (drapeaux `-DSTM32F1 -DSTM32F10X_MD`) et qui est émulé dans le port d'André Beckus de `qemu` à cette plateforme (`beckus.github.io/qemu_stm32/`), et le STM32F410 (drapeaux `-DSTM32F4`) qui va nous intéresser dans la suite de cette présentation (Fig. 1). En plus des drapeaux (choix de la déclinaison de la bibliothèque `libopencm3` et de la nature du processeur, `cortex-m3` ou `cortex-m4`), le script définissant l'organisation de la mémoire achève de définir la cible. Ce script, d'extension `.ld`, est à ajuster pour chaque processeur : les scripts supportés sont dans le répertoire `ld`, fournissant l'argument de l'option `-T` de l'éditeur de lien au moment de rassembler les objets et les bibliothèques pour former un exécutable. Ces divers drapeaux sont visibles dans l'exemple de `Makefile` proposé à `github.com/jmfriedt/tp_freertos/blob/master/0no_freertos/Makefile.common`.

La raison pour proposer le support du STM32F103 est de permettre au lecteur qui ne possède pas de plateforme matérielle d'exécuter le code sur l'émulateur `qemu` tel qu'adapté par André Beckus. Dans le cas de cet exemple, nous vérifions le bon fonctionnement de la compilation pour STM32F103 – `-DSTM32F1 -DSTM32F10X_MD` – par l'exécution sous `qemu` par `arm-softmmu/qemu-system-arm -M stm32-p103 -serial stdio -kernel usart_cm3.bin`.

En activant GPIOC12 comme broche connectée à la LED pour être en accord avec la configuration de la plateforme émulée par André Beckus (voir `hw/arm/stm32_p103.c` dans ses sources), nous obtenons en sortie

```
LED On
0
LED Off
1
LED On
2
```

en accord avec nos attentes sur les fonctionnalités du programme.

Nous avons choisi de cadencer le STM32F410 par un quartz de 20 MHz. Ce choix, arbitraire, rompt avec l'habitude de cadencer les microcontrôleurs de la gamme STM32 avec un quartz de fréquence de résonance multiple de 8 MHz tel que supporté par défaut par `libopencm3`. La conséquence est de devoir manipuler à la main les paramètres de la boucle à verrouillage de phase (PLL) qui alimente les divers périphériques du microcontrôleur à partir de cet oscillateur de référence. Un ensemble de relations entre coefficients de multiplication et de division de la PLL, compte tenu des fréquences maximales acceptables par les divers périphériques, est résumé dans un document Excel fourni par ST-Microelectronics dans sa note d'application AN3988 et le logiciel associé STSW-STM32091. Dans notre cas, cela se résume par

```
#include <libopencm3/stm32/rcc.h>
#include <libopencm3/stm32/f4/memorymap.h>
#include <libopencm3/stm32/flash.h> // definitions du timer

// APB2 max=84 MHz but when the APB prescaler is NOT 1, the interface clock is fed
// twice the frequency => Sysclk = 140 MHz, APB2=2 but Timers are driven at twice that is 140.
const struct rcc_clock_scale rcc_hse_20mhz_3v3 = {
  .pllm = 20,                           // 20/20=1 MHz
  .plln = 280,                          // 1*280/2=140 MHz
  .pllp = 2,                            //       ^
  .pllq = 6,
  .hpre = RCC_CFGR_HPRE_DIV_NONE,
  .ppre1 = RCC_CFGR_PPRE_DIV_4,
  .ppre2 = RCC_CFGR_PPRE_DIV_2,
  .flash_config = FLASH_ACR_ICE | FLASH_ACR_DCE | FLASH_ACR_LATENCY_4WS,
  .ahb_frequency  = 140000000,
  .apb1_frequency = 35000000,
  .apb2_frequency = 70000000,
};

void core_clock_setup(void) {rcc_clock_setup_hse_3v3(&rcc_hse_20mhz_3v3);} // custom version
```

Le microcontrôleur est ainsi légèrement sur-cadencé, mais nous n'avons pas observé de dysfonctionnement au cours de nos expérimentations, et nous pourrons ainsi générer le signal d'horloge à 70 MHz nécessaire au bon fonctionnement du synthétiseur numérique de fréquence.

4 Organisation d'un programme FreeRTOS

Un programme exploitant les fonctionnalités de FreeRTOS doit déclarer les options dont il a besoin. Il est classique de placer ce fichier de configuration, nommé `FreeRTOSConfig.h` – documenté en détail à `www.freertos.org/a00110.html` – aux cotés de `main.c` dans le répertoire `src` de travail, en dehors de l'arborescence de FreeRTOS. Ce fichier contient des informations telles que la fréquence du processeur, la fréquence à laquelle l'ordonnanceur considère quelle tâche exécuter, la taille de la pile allouée à l'ordonnanceur (tâche idle), le nombre de niveaux de priorités des tâches et nous verrons plus loin certaines fonctionnalités spécifiques telles que le support des mutex. Bien entendu augmenter le nombre de fonctionnalités augmente les ressources requises : ce fichier est donc le levier pour adpater FreeRTOS à son application et aux ressources mises à disposition par le microcontrôleur.

La principale difficulté identifiée lors de l'utilisation de la bibliothèque `libopencm3` tient au gestionnaire d'interruption *timer* qui cadence l'ordonnanceur. FreeRTOS s'attend à ce que des fonctions aux noms bien précis soient appelées par le gestionnaire d'interruption. Par exemple, `FreeRTOSv9.0.0/FreeRTOS/Demo/CORTEX_M4F_STM32F407ZG-SK/FreeRTOSConfig.h` propose la solution pour la bibliothèque `libstm32` fournie par ST Microelectronics sous forme de

```
#define vPortSVCHandler SVC_Handler
#define xPortPendSVHandler PendSV_Handler
#define xPortSysTickHandler SysTick_Handler
```

que nous adaptons pour `libopencm3` selon les conseils de `www.jiggerjuice.info/electronics/projects/arm/freertos-stm32f103-port.html` pour obtenir

```
void sv_call_handler(void)  {vPortSVCHandler();}
void pend_sv_handler(void)  {xPortPendSVHandler();}
void sys_tick_handler(void) {xPortSysTickHandler();}
```

Une fois ces modificiations apportées, nous sommes prêts à tester notre premier exemple de programme FreeRTOS

```c
#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)
  {Led_Hi2(); vTaskDelay(301/portTICK_RATE_MS);
   Led_Lo2(); vTaskDelay(301/portTICK_RATE_MS);
  }
}

void vPrintUart(void* dummy)
{portTickType last_wakeup_time;
 last_wakeup_time = xTaskGetTickCount();
 // while (1) {} /* tester ce qui se passe avec boucle infinie ! */
 while(1)
   {uart_puts("Hello World\r\n");
    vTaskDelayUntil(&last_wakeup_time, 500/portTICK_RATE_MS);
   }
}

int main(void){
 Usart1_Init(); // inits clock as well
 Led_Init();

 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;
 if (!(pdPASS == xTaskCreate( vPrintUart, (signed char*) "Uart",    64,NULL,3,NULL ))) goto hell;
 vTaskStartScheduler();
 hell:  while(1); // should never be reached
 return 0;
}
```

Listing 2 – Premier programme FreeRTOS

Ce programme nous informe de l'organisation générale d'un programme FreeRTOS : nos fonctions d'accès bas niveau initialisent les périphériques et les horloges, les tâches sont enregistrées auprès de l'ordonnanceur qui est ensuite exécuté. L'ordonnanceur ne doit jamais arriver en famine d'activité (toujours au moins une tâche en attente) et donc la fin du programme n'est jamais atteinte (équivalent de la boucle infinie principale dans un programme embarqué sans système d'exploitation). Ici encore, ce programme se teste dans `qemu` par
`arm-softmmu/qemu-system-arm -M stm32-p103 -serial stdio -kernel output/main.bin` pour donner

```
LED Off
LED On
Hello World
LED Off
LED On
LED Off
LED On
Hello World
```

Toutes nos expériences ont exploité la version d'André Beckus de `qemu`, facile à modifier et à adapter à nos besoins puisque épurée au maximum pour supporter l'ARM Cortex-M3. Cependant, un projet s'est scindé et des développeurs de l'environnement intégré de développement Eclipse proposent une version bien plus riche de `qemu` disponible à `github.com/gnuarmeclipse/qemu/releases/tag/gae-2.8.0-20161227`. Cette version reste fonctionnelle en ligne de commande, tel que décrit à `gnuarmeclipse.github.io/qemu/options/`. Pour nos exemples, exécuter

```
qemu-system-gnuarmeclipse --verbose --verbose -M STM32-P103 \
-nographic -d unimp,guest_errors -kernel $FREERTOS_HOME/output/main.bin
```

sur un exemple qui fait clignoter deux LEDs se traduit par

```
GNU ARM Eclipse 64-bits QEMU v2.8.0 (qemu-system-gnuarmeclipse).
Board: 'STM32-P103' (Olimex Prototype Board for STM32F103RBT6).
Device file: '[...]/qemu/devices/STM32F103xx-qemu.json'.
Device: 'STM32F103RB' (Cortex-M3 r0p1, MPU, 4 NVIC prio bits, 43 IRQs), Flash: 128 kB, RAM: 20 kB.
Image: '[...]/0no_freertos/usart_cm3.elf'.
Command line: (none).
Load   1784 bytes at 0x08000000-0x080006F7.
Load     12 bytes at 0x080006F8-0x08000703.
Cortex-M3 r0p1 core initialised.
'/machine/mcu/stm32/RCC', address: 0x40021000, size: 0x0400
'/machine/mcu/stm32/FLASH', address: 0x40022000, size: 0x0400
'/machine/mcu/stm32/PWR', address: 0x40007000, size: 0x0400
'/machine/mcu/stm32/AFIO', address: 0x40010000, size: 0x0400
'/machine/mcu/stm32/EXTI', address: 0x40010400, size: 0x0400
'/machine/mcu/stm32/GPIOA', address: 0x40010800, size: 0x0400
'/machine/mcu/stm32/GPIOB', address: 0x40010C00, size: 0x0400
'/machine/mcu/stm32/GPIOC', address: 0x40011000, size: 0x0400
'/machine/mcu/stm32/GPIOD', address: 0x40011400, size: 0x0400
'/machine/mcu/stm32/GPIOE', address: 0x40011800, size: 0x0400
'/peripheral/led:red' 12*10 @(331,362) active low '/machine/mcu/stm32/GPIOC',12
QEMU 2.8.0 monitor - type 'help' for more information^M
(qemu) Cortex-M3 r0p1 core reset.

NVIC: SCR and CCR unimplemented
[led:red off]
[led:red on]
[led:red off]
[led:red on]
```

qui est bien le comportement attendu. Cependant, nous n'avons pas réussi à convaincre le port série d'afficher un message sur la sortie standard ou un client `telnet`, même en n'utilisant que le port 2, le seul supporté par la carte Olimex STM32-P103.

Les tâches sont enregistrées par `xTaskCreate()` [4, p.6], qui prend en argument la fonction à appeler, le nom (libre) qui permet d'identifier la tâche, 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 – le dernier paramètre étant souvent NULL si le descripteur de la tâche n'est pas utilisé dans le programme (par exemple pour influer sur l'ordonnanceur). Sur architecture ARM, la taille de la pile est automatiquement multipliée par quatre pour garantir l'alignement sur les adresses de 32 bits si la valeur est passée sous forme de `STACK_BYTES(val)`. La priorité de tâche est d'autant plus élevée que l'avant dernier argument est élevé (convention opposée à celle des priorités sous unix). On pourra s'en convaincre en retirant dans la tâche d'affichage `vPrintUart` la fonction `vTaskDelayUntil()` qui propose à l'ordonnanceur de pré-empter la tâche : dans ces conditions, les LEDs ne clignotent plus. Au contraire si, tout en retirant la fonction permettant la pré-emption, nous inversons la priorité des trois tâches – en passant `vPrintUart()` en 1 et `vLedsFloat` en 3, on constatera que les LEDs se remettent à clignoter. En effet dans ce dernier cas, l'ordonnanceur s'autorise à pré-empter périodiquement la tâche de priorité la plus faible pour proposer aux tâches de priorités plus élevées de prendre la main. De la même façon, bloquer par un `while (1) {}` une tâche de faible priorité n'empêchera pas une tâche de forte priorité d'être pré-emptée, mais une telle boucle infinie dans une tâche de priorité la plus élevée se traduit par un blocage de l'exécution.

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

Le lecteur qui désire tenter d'exécuter ce code sous GNU/Linux pourra tenter d'exploiter l'émulateur FreeRTOS sur threads POSIX : la version 9.0.0 de FreeRTOS est supportée par `github.com/megakilo/FreeRTOS-Sim.git`. L'exemple ci-dessus finit par fonctionner, mais nécessite d'initialiser un certain nombre de fonctionnalités qui n'ont pas été nécessaires pour faire tourner FreeRTOS sur microcontroleur. En particulier, toutes les initialisations de `vCreateAbortDelayTasks()` à `vStartTaskNotifyTask();` dans `Project/main.c` doivent être maintenues pour initialiser les structures de données émulant le matériel. Nous obtenons bien dans un terminal à l'issue de la compilation un exécutable fonctionnel sous GNU/Linux qui nous informe du séquencement des tâches :

```
Running as PID: 6651
Led_Hi1
Timer Resolution for Run TimeStats is 100 ticks per second.
Hello World
Led_Hi2
Led_Hi1
Led_Lo1
Led_Hi1
Led_Lo2
Led_Lo1
Led_Hi1
Hello World
Led_Lo1
```

Ce premier exemple nous permet déjà d'appréhender l'importance des priorités des tâches. Imaginons que nous introduisions maladroitement dans la tâche `vPrintUart()` une boucle infinie vide `while (1) {}` juste avant la boucle proposée dans cet exemple. Dans la configuration qui est proposée, où `vPrintUart()` est de priorité la plus élevée, cette boucle infinie ne peut pas être interrompue par l'ordonnanceur, et le programme est bloqué, aucun message n'apparaît sur le port série et aucune LED ne clignote. Si au contraire nous inversons les priorités de `vPrintUart()` et `vLedsFloat()`, passant le clignotement des LEDs prioritaire devant l'affichage du message, alors la boucle infinie interdit toujours l'affichage d'un message sur le port série (la tâche `vPrintUart()` ne sort jamais de sa boucle infinie), mais au moins les diodes clignotent puisque l'ordonnanceur pré-empte les tâches de priorité les plus élevées.

4.1 Informations sur l'utilisation de la pile et des tâches

Pour le moment, nous n'avons que créé trois tâches qui s'exécutent apparemment simultanément, deux qui font clignoter une LED et la dernière qui communique. Le gain de FreeRTOS est pour le moment douteux. Que peut nous amener l'environnement exécutif ? Le premier gain significatif qui permet d'appréhender un problème classique dans le développement de systèmes embarqués est la gestion de la pile. La pile est la zone temporaire où une tâche stocke les informations qui lui sont relatives au cours de son exécution (variables locales, emplacement du pointeur de programme lors des sauts aux fonctions) : chaque tâche contient sa propre pile, et corrompre la pile est la garantie de crash du programme. La pile (*stack*), généralement placée à la fin de la RAM puisqu'empiler une variable se traduit par un décrément du pointeur de pile, se distingue du tas (*heap*) qui se trouve généralement en début de RAM, et dont le contenu a la durée de vie de l'exécution du programme (contrairement à la pile qui n'est valable que pendant la durée d'exécution de la tâche). FreeRTOS fournit plusieurs mécanismes pour informer le programmeur du statut de la pile et de son risque de corruption. L'option `#define configCHECK_FOR_STACK_OVERFLOW 2` dans `FreeRTOSConfig.h`, qui nécessite la définition du gestionnaire d'événement `void vApplicationStackOverflowHook(TaskHandle_t xTask, signed char *pTaskName)`, permet d'appeler ledit gestionnaire d'évènements si une corruption de pile survient. Par exemple en définissant

```
void vApplicationStackOverflowHook(TaskHandle_t xTask, signed char *pTaskName)
{while(1) {uart_puts("\r\nStack: ");uart_puts(pTaskName);vTaskDelay(301/portTICK_RATE_MS);}
}
```

nous constatons que réduire la taille de la pile allouée à la tâche `vPrintUart()` à deux mots de 32 bits se traduit par

```
Stack: Uart
Stack: Uart
```

qui indique que la fonction d'affichage d'un message sur le port série a dépassé la taille de pile qui lui a été allouée.

De la même façon, une taille insuffisante par défaut de pile `configMINIMAL_STACK_SIZE`, qui définit en particulier l'allocation de mémoire de la tâche qui ne fait rien (en attente de tâches à activer), se traduit par

```
Stack: IDLE
Stack: IDLE
```

avec ici encore une tâche qui a dépassé la taille de pile qui lui a été allouée. Bien entendu, un programme fonctionnel peut activer ce mode de déverminage et doit ne jamais voir la fonction `vApplicationStackOverflowHook()` appelée dans la condition nominale de fonctionnement où chaque tâche confine ses allocations en mémoire à la pile qui lui a été allouée.

Dans la même veine, une tâche peut essayer de savoir combien de place il lui reste à allouer sur sa pile. Pour ce faire, l'option de configuration `#define INCLUDE_uxTaskGetStackHighWaterMark 1` est activée et la fonction `uxTaskGetStackHighWaterMark(NULL);` (NULL signifiant que la tâche veut connaître son propre état, sinon il faut fournir le handler de la tâche consultée) renvoie l'information recherchée.

Enfin, nous pouvons afficher la liste des tâches enregistrées auprès de l'ordonnanceur, leur état et la quantité de pile restante. Ici encore ces options s'activent dans le fichier de configuration, cette fois par

```
#define configUSE_TRACE_FACILITY          1
#define configUSE_STATS_FORMATTING_FUNCTIONS     1
```

et en allouant un tableau d'au moins 40 caractères par tâche dont l'état doit être affiché, pointeur qui sera fourni en argument à la fonction `vTaskList` qui remplit le tableau de caractères de la chaîne à afficher. Ainsi,

```
void vPrintUart(void* dummy)
{char c[256];
 portTickType last_wakeup_time;
 last_wakeup_time = xTaskGetTickCount();
 while(1){uart_puts("\nHello World\r\n");
          vTaskList(c); uart_puts(c);
          vTaskDelayUntil(&last_wakeup_time, 500/portTICK_RATE_MS);
        }
}
```

affiche toutes les 500 ms l'état des tâches enregistrées auprès de l'ordonnanceur. Bien entendu, la taille de la pile associée à l'initialisation de `vPrintUart` doit être plus grande que le tableau qui y est créé.

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 dans les colonnes successives le nom que nous avons arbitrairement attribué à la tâche (second argument de `xTaskCreate()`, le statut de la tâche (R=Ready, B=Blocked, D=Deleted et S=Suspended=Blocked sans timeout), la priorité de la tâche et la quantité de mémoire encore disponible sur la pile (plus cette valeur se rapproche de 0, plus le risque de corruption de la pile devient important), puis finalement la position dans la file de l'ordonnanceur de la tâche. Dans tous ces exemples, on pourra s'entraîner à faire varier la taille de la pile allouée lors de la déclaration de la tâche `vPrintUart` et constater la cohérence des informations fournies par FreeRTOS. Notons que seuls quelques KB de mémoire ont été nécessaires pour faire tourner FreeRTOS : les programmes fournis dans ce document fonctionnent tous sur le STM32F100 de la STM32VL-Discovery board (8 KB de RAM) sous réserve de ne pas dilapider les ressources, par exemple en se liant à `newlib`.

4.2 Échange de données par files d'attente

Nous savons donc créer des tâches, garantir l'intégrité de leur pile, et créer des variables locales à chaque tâche. Il est courant que plusieurs tâches aient à échanger des données pour y appliquer des traitements successifs. Nous ne savons cependant pas quel est l'état de chaque tâche au moment de la production ou la consommation de données, et il nous faut un mécanisme permettant à deux tâches d'échanger des données sans faire d'hypothèse sur le comportement de l'ordonnanceur. Un tel mécanisme est fourni par les queues, qui sont des FIFO (*First In, First Out*) entre deux tâches.

```
#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)
[...]

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
//   xTaskCreate( vLedsFloat, ( signed char * ) "LedFloat", 128, NULL, 2, NULL );
//   xTaskCreate( vLedsFlash, ( signed char * ) "LedFlash", 128, NULL, 2, NULL );
//   xTaskCreate( vPrintUart, ( signed char * ) "Uart",     128, NULL, 2, NULL );
   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;
}
```

Dans cet exemple, deux fonctions `task_tx()` et `task_rx()` échangent des données avec une condition de *timeout* qui informe d'un délai excessif entre les transactions. On se convaincra du bon fonctionnement de ce mécanisme en ajoutant les fonctions de commutation des LEDs et de communication – identiques à celles vues auparavant – qui introduisent des latences additionnelles et font atteindre la condition de délai dépassé, en fonction des priorités des diverses tâches.

4.3 Protection de l'accès concurrent aux ressources communes à plusieurs tâches

Le danger de partager des structures de données globales – en particulier tableaux servant de tampons lors de l'acquisition et de traitements de 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

incohérentes. Une méthode pour éviter ce problème est de garantir qu'*un seul processus* peut accéder à une variable à un instant donné : accès MUTuellement EXclusif ou *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 : une tâche qui tente de s'approprier un mutex déjà pris est mise en attente jusqu'à ce que ce mutex soit relaché, et réussir à prendre un mutex garantit qu'aucune autre tâche n'accède aux structures de données protégées jusqu'à ce que ce mutex soit rendu. En pratique, la mise en œuvre peut ressembler à

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

// #define avec_mutex

volatile int global=0; // la variable incrementee par les 2 taches *8
#ifdef avec_mutex
SemaphoreHandle_t xMutex; // pour FreeRTOSv9
#endif

void task_rx(void* p)
{   char aff[10];
    char *t=(char*)p;
    int myInt = 0;
    volatile int local;
    for (myInt=0;myInt<8;myInt++)
     {
#ifdef avec_mutex
      xSemaphoreTake( xMutex, portMAX_DELAY );
#endif
      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;

#ifdef avec_mutex
    xSemaphoreGive( xMutex );
#endif
    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 );
#ifdef avec_mutex
  xMutex = xSemaphoreCreateMutex();
#endif
  xTaskCreate(task_rx, (signed char*)"t1", 128, "1111111111111111111111111111111111111111\r\n\0", 1,
     0);
  xTaskCreate(task_rx, (signed char*)"t2", 128, "2222222222222222222222222222222222222222\r\n\0", 1,
     0);
  vTaskStartScheduler();
  while(1);
  return 0;
}
```

L'exemple ci-dessus considère une variable globale `global` 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 :

un transfert de 45 caractères au débit de 115200 bauds selon un encodage 8N1 sur bus RS232 prend près de 4 ms). Si nous omettons de protéger l'accès à `global`, les deux tâches risquent d'en stocker dans `local` une valeur à un certain temps, de toutes deux effectuer une addition sur cette même valeur initiale, et la première tâche stocke son résultat suivie de la seconde tâche qui stocke le *même résultat*, induisant une erreur de calcul puisque l'addition aurait du s'effectuer deux fois au lieu d'une. La conséquence de la pré-emption d'une tâche par l'ordonnanceur alors qu'un affichage est en cours est donc que d'une part les chaînes de caractères ne sont pas affichées dans leur intégralité mais sont interlacées, et la somme de `global` n'est pas nécessairement correcte à l'issue du calcul :

```
22222222222222222222222111111111111111111111112222222222222222222
1111111111111111111
222222222222222222222222211111111111111111111111111222222222222222222
...
 :  :
```

Les deux symboles " :" à la fin du calcul (code ASCII de "0" plus 12) indiquent que la somme de `global` n'est pas 16 comme escompté (deux tâches qui incrémentent huit fois) mais moindre à cause du conflit d'utilisation de la ressource commune qu'est la variable globale.

Le palliatif consiste à décommenter l'option de précompilation `#define avec_mutex` qui active la protection de l'accès à `global` puis `local` par une unique tâche à un instant donné. Ce résultat est confirmé par chaque chaîne de caractère passée en argument aux deux instances de la tâche qui s'affichent entièrement et sans être interrompues par l'autre tâche que l'ordonnanceur s'autorise à préempter compte tenu de la durée excessive de la communication par RS232.

```
...
22222222222222222222222222222222222222222222
11111111111111111111111111111111111111111111
22222222222222222222222222222222222222222222
11111111111111111111111111111111111111111111
22222222222222222222 ? 22222222222222222222
 @
```

avec le " ?" et "@" qui sont respectivement le code ASCII de "0" plus 15 et 16, soit le résultat attendu de la dernière exécution de chacune des tâches qui incrémente chacune `global` en se garantissant la ressource tant que l'affichage n'est pas achevé.

Compiler ces programmes nécessite d'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 `FreeRTOSConfig.h`.

5 Application à l'analyseur de réseau

Nous avons complété ce rapide tour d'horizon de FreeRTOS, reste maintenant à mettre ces concepts en œuvre dans un cas concret. Nous nous intéressons à la réalisation d'un analyseur de réseau embarqué (Fig. 4), un instrument susceptible de caractériser la fonction de transfert, en amplitude et en phase, d'un dispositif radiofréquence [5]. Une description détaillée des choix de conception de l'instrument est proposée à `jmfriedt.free.fr/network_analyzer.pdf` et dépasse le cadre de cette présentation, qui se limite à l'implémentation du logiciel sur FreeRTOS. Nous avons donc, en plus du microcontrôleur, besoin de trois composants :

1. une source radiofréquence. Le synthétiseur numérique direct (*Direct Digital Synthesizer – DDS*) Analog Devices AD9834 fournit les fonctionnalités nécessaires : un composant totalement numérique ne nécessitant que peu de composants passifs annexes, un pas de fréquence fin programmable depuis le microcontrôleur, et une puissance de sortie faible mais suffisante pour notre mesure. Un DDS est cadencé par une horloge externe, qui dans notre cas sera fournie par une sortie *timer* du microcontrôleur : cadencé à 140 MHz, le STM32F410 peut fournir une horloge à $f_{CK} = 70$ MHz sur une sortie *timer*, dans la plage acceptable par l'AD9834. La fréquence de sortie f_o est liée à f_{CK} et au mot w programmé par le microcontrôleur dans le registre adéquat par

$$f_o = \frac{w}{2^{28}} \times f_{CK}$$

avec 2^{28} la taille de l'accumulateur interne au DDS (28 bits). Le pas de fréquence est donc $f_{CK}/2^{28} \simeq 0,3$ Hz, suffisant pour caractériser des dispositifs de fort facteur de qualité (donc de faible encombrement spectral),

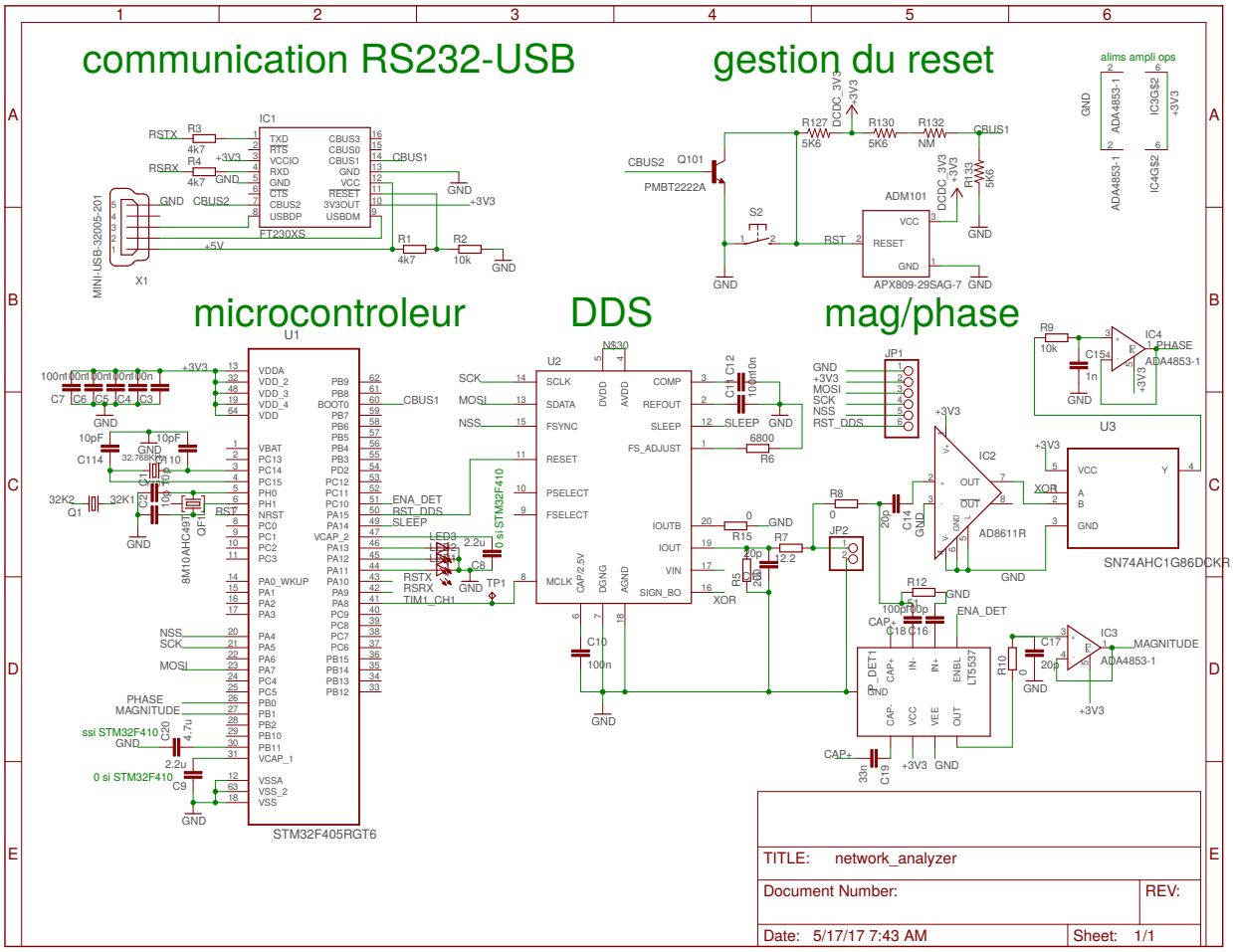

FIGURE 4 – Schéma de l'analyseur de réseau complet. L'ensemble des composants présents sur ce circuit – microcontrôleur, synthèse de fréquence, détecteur de puissance et de phase, coûte moins de 30 euros.

2. d'un détecteur de puissance. Une puissance – ou plus simplement une tension – s'obtient classiquement par un redressement du signal suivi d'un filtrage passe-bas pour obtenir l'amplitude moyenne du signal. Au lieu de nous battre à polariser convenablement une diode radiofréquence pour concevoir un redresseur fonctionnel dans une large plage de fréquences et de puissances d'entrée, nous nous contentons d'utiliser un détecteur de puissance commercialement disponible chez Linear Technology, le LT5537. Ce composant, fonctionnel de quelques kHz au GHz et pour des puissances aussi basses que -75 dBm, propose par ailleurs une mise en veille pour une économie d'énergie s'il n'est pas utilisé,
3. finalement, un détecteur de phase (Fig. 5). Nous choisissons ici la voie du numérique, avec une implémentation classique de mélangeur par porte logique XOR (OU eXclusif). En effet dans une telle porte, deux signaux en phase donnent toujours une sortie nulle (0 XOR 0=1 XOR 1=0) tandis que deux signaux en opposition de phase donnent toujours une sortie égale à la tension d'alimentation (0 XOR 1=1). Entre les deux, un filtre passe-bas en sortie du XOR donne une tension continue proportionnelle au déphasage entre les deux signaux d'entrée, signal de référence issu du DDS et signal qui a sondé le dispositif en cours de caractérisation (DUT). Alors que le DDS fournit une sortie numérique de fréquence égale à celle générée sur la sortie analogique, le niveau du signal analogique qui a sondé le DUT est insuffisant pour attaquer la seconde entrée du composant numérique qu'est la porte XOR. Un comparateur est donc nécessaire pour saturer le signal en sortie du DUT et permettre une mesure de phase.

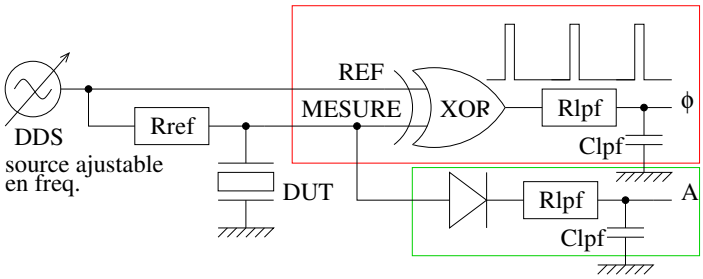

Figure 5: Principe de la mesure d'amplitude (vert) et de phase par porte XOR (rouge).

Une fois ces concepts mis en œuvre sur un circuit fonctionnel, il nous reste à programmer le microcontrôleur pour balayer séquentiellement les fréquences dans la plage qui nous intéresse, et pour chacune de ces fréquences lire sur deux voies de conversion analogique-numérique du microcontrôleurs les valeurs de phase et d'amplitude pour tracer la caractéristique du dispositif analysé. L'utilisateur voulant pouvoir programmer la plage de fréquences et le pas de fréquence du balayage, le programme FreeRTOS se compose de trois tâches :

1. une tâche indiquant que l'instrument est sous tension et en attente d'ordres de l'utilisateur, allumant et éteignant des diodes électroluminescentes (LEDs) dans un motif de chenillard,
2. une tâche de communication chargée de recevoir les paramètres de l'utilisateur : fréquence de début, fréquence de fin, pas de fréquence et attente entre deux mesures,
3. une tâche de mesure chargée de programmer le DDS et pour chaque fréquence, de mesurer la phase et l'amplitude et transmettre ces informations à l'utilisateur.

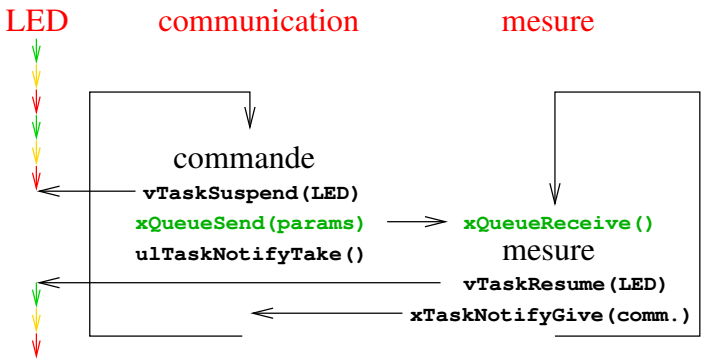

Figure 6: Séquencement des tâches – en rouge en haut du schéma – de l'application : la communication d'une séquence de caractères depuis le PC débloque la tâche *communication* qui bloque la tâche *LED* et lance la *mesure*. Une fois la *mesure* achevée, les *LED* se remettent à clignoter et la *communication* est en attente d'un nouvel ordre de mesure. Le temps s'écoule de haut en bas.

Ces trois tâches doivent donc interagir (Fig. 6) : la tâche LED doit cesser de faire clignoter les diodes au cours de la mesure car l'appel de courant fait varier la tension de référence des convertisseurs analogique-numériques mais doit se remettre en marche à l'issue de la mesure, la tâche mesure doit être bloquée tant que les paramètres n'ont pas été acquis et la tâche de réception des paramètres doit communiquer les fréquences à la tâche de mesure. L'échange des paramètres se fait par les queues que nous avons vu auparavant. Puisqu'un unique pointeur est fourni comme argument du passage de paramètres par queue, une structure de donnée comprenant tous les paramètres est créée dont le pointeur sera passé dans la queue entre les tâches de communication et de mesure. Par contre pour ce qui est de séquencer les opérations en bloquant et débloquant les diverses tâches, nous utilisons des mécanismes optimisés par FreeRTOS au lieu des classiques mutex pour garantir qu'une seule tâche accède à une ressource donnée à un instant donné. Pour ce faire, les mécanismes `vTaskSuspend()` et `vTaskResume()` permettent à une tâche d'interrompre ou reprendre l'exécution d'une autre tâche dont le descripteur (pointeur fourni comme dernier argument de `xTaskCreate()`) est fourni en argument. Par ailleurs, `xTaskNotifyGive()` permet à une tâche de débloquer une autre tâche qui est en attente par `ulTaskNotifyTake()`. La première fonction prend en argument le descripteur de la tâche à débloquer, tandis que la seconde fonction se content d'un temps maximum au-delà duquel la tâche se réveillera – éventuellement infini. Enfin, l'échange d'informations par queues, `xQueueSend()` et `xQueueReceive()`, est aussi utilisé pour séquencer les opérations, toujours avec possibilité de placer un temps maximum (*timeout*) sur la réception si la réception de message ne doit pas être une condition bloquante. Si la réception de message n'a pas abouti avant le *timeout*, `xQueueReceive()` renverra `pdFalse` (`www.freertos.org/a00118.html`), permettant de connaître quelle condition a débloqué la tâche.

Une application de ces concepts, dont le code source complet est à `github.com/jmfriedt/tp_freertos/tree/master/FreeRTOS-Network-Analyzer-on-STM32`, est proposée en Fig. 7 : un résonateur à onde de volume, tel que classiquement utilisé pour cadencer les microcontrôleurs, est caractérisé par cet instrument. Le paramètre clé d'un résonateur, en plus de sa fréquence de résonance qui détermine la fréquence du signal issu du circuit d'entretien de l'oscillation (oscillateur), est son facteur de qualité qui détermine sa capacité à emmagasiner de l'énergie et donc d'être insensible aux perturbations extérieures, formant ainsi un oscillateur stable. Le facteur de qualité, au-delà de ces considérations énergétiques, détermine la largeur de la raie spectrale caractérisant le résonateur : un résonateur à onde de volume en quartz présente classiquement un facteur de qualité Q de l'ordre de 10^5 pour des fréquences de fonctionnement f de quelques MHz : la largeur de la résonance est alors $\Delta f = f/Q$ de l'ordre de quelques dizaines à quelques centaines de Hz (la valeur choisie ici pour Q est volontairement sur-estimée pour choisir Δf suffisamment petit pour correctement caractériser la réponse du dispositif). Le pas de fréquence lors de la programmation du DDS doit donc être petit devant

Δf : si par exemple la résonance doit se caractériser sur N points, on choisira un pas de fréquence lors de la programmation du DDS de $\Delta f/N$. Dans notre cas, nous choisissons un pas de 5 Hz sur une plage de ± 500 Hz autour de la résonance, soit 1000/5=200 points de mesure qui prennent quelques dizaines de secondes à être transférées par liaison RS232. Le balayage ne doit pas être trop rapide, faute de quoi l'énergie emmagasinée à la résonance dans le résonateur n'est pas totalement dissipée lors de la mesure suivante, déformant la fonction de transfert : nous devons attendre quelques constantes de temps $Q/(\pi f) \simeq 6$ ms (à 5 MHz) entre deux mesures pour garantir l'indépendance des mesures successives. Ce temps d'attente pourra être mis à profit pour communiquer les données au PC, la liaison RS232 étant lente.

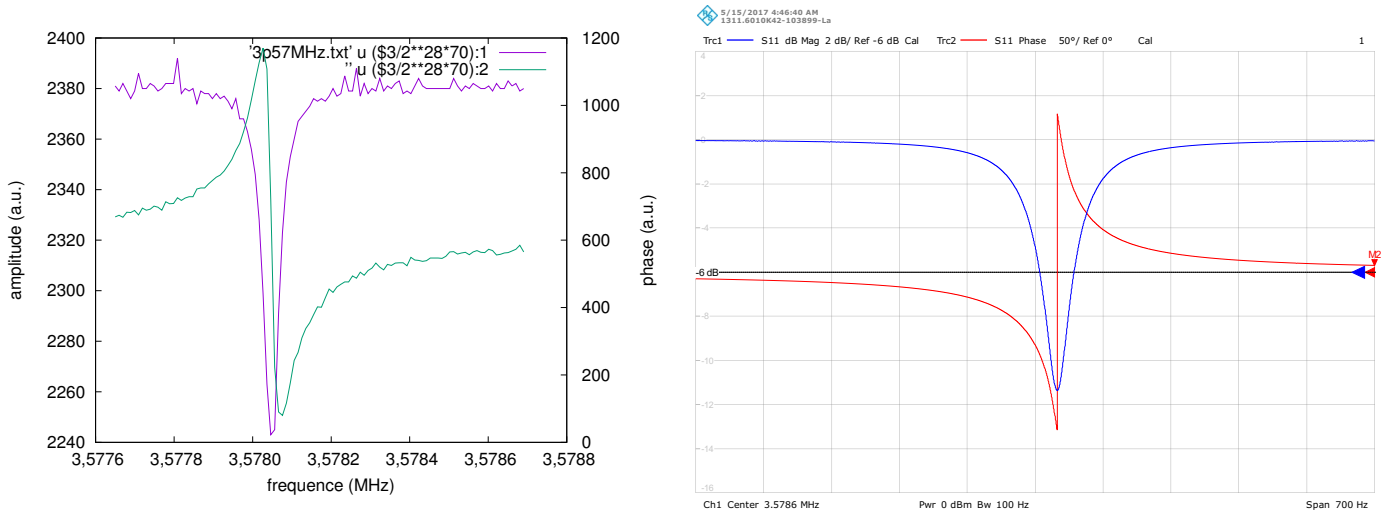

FIGURE 7 – Gauche : mesure d'un résonateur à onde de volume par le circuit comprenant un microcontrôleur, un DDS, un détecteur de puissance et un détecteur de phase. La légende en haut à droite du graphique contient les commandes `gnuplot` permettant l'affichage des courbes avec un axe des abscisses gradué en MHz au lieu des mots représentant la fréquence dans le DDS. Droite : caractérisation du même dispositif avec un analyseur de réseau commercial.

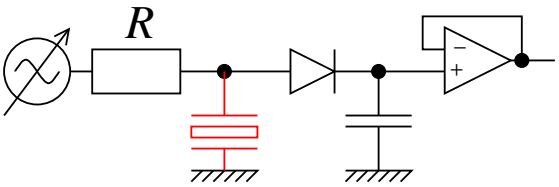

Figure 8: Mesure du coefficient de réflexion du dispositif : la fraction de la puissance qui atteint la détecteur de puissance est celle qui n'est pas transmise par le dispositif vers la masse. L'impédance de référence est R, le DUT est en rouge.

Dans la configuration de la mesure que nous proposons, dans laquelle le DUT est inséré dans un pont diviseur de tension vers la masse (Fig. 8), la résonance s'observe sous forme d'une chute de puissance arrivant au détecteur : hors résonance l'intégralité du signal fourni par le DDS arrive au détecteur, et une fraction de cette puissance est dirigée vers la masse lorsque la résonance est atteinte. La fraction de la puissance dirigée vers la masse est donnée par la relation classique du pont diviseur de tension entre une impédance de référence – classiquement 50 Ω, ici la résistance en sortie du DDS – et le DUT – et la conservation de l'énergie nous indique la fraction résultante qui est dirigée vers le détecteur de puissance. Fig. 7 présente bien une chute de puissance lorsque la résonance est atteinte, associée à une rotation de phase.

Le lecteur désireux de reproduire ces mesures pourra profiter d'une interface graphique qui simplifie la prise en main des paramètres de configuration, tel que présenté sur la Fig. 9.

6 Conclusion

Nous avons conçu un circuit autour du STM32F410 en vue de réaliser un analyseur de réseau radiofréquence pour la caractérisation des dispositifs fonctionnant dans la gamme de quelques centaines de kHz à quelques dizaines de MHz. Ce développement a été l'opportunité d'explorer les fonctionnalités de FreeRTOS, porté à ce microcontrôleur grâce à l'utilisation de la bibliothèque `libopencm3`. Nous avons vu que la simplicité de l'architecture de FreeRTOS le rend facilement exploitable sur n'importe quel microcontrôleur muni de quelques kB de mémoire.

Les applications de ce circuit sont multiples, au-delà de la caractérisation des dispositifs radiofréquences : dans le cas particulier des résonateurs à onde de volume décapsulés, leur sensibilité à leur environnement en fait d'excellents capteurs. En particulier, la variation de la fréquence de résonance avec l'épaisseur d'une couche adsorbée en surface du substrat piézoélectrique (chargé de convertir le signal électromagnétique en onde élastique) en fait des capteurs chimiques à la mode, dont l'électronique présentée dans ce document permet de mesurer finement la réponse en observant l'évolution de

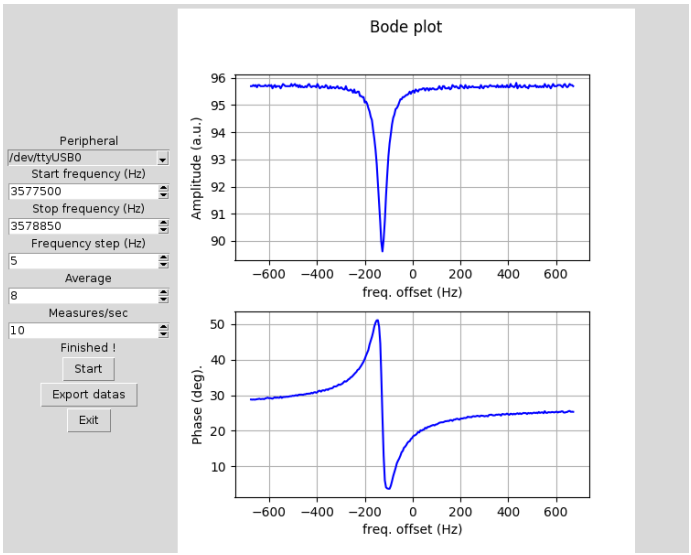

FIGURE 9 – Interface graphique pour la configuration de l'instrument et la restitution des courbes – le code source est disponible à `github.com/jmfriedt/tp_freertos/blob/master/FreeRTOS-Spectrum-Analyzer-on-STM32/Interface.py` pour l'échange des informations sur port série et l'affichage des informations acquises par les convertisseurs analogique-numériques.

la phase du résonateur à la résonance, dans une application abusivement nommée de "microbalance à quartz". De façon générale, les capteurs à sortie de fréquence fournissent une excellente sensibilité de par la capacité à mesurer avec une très grande précision les variations de fréquences, ici uniquement limitée par la stabilité de l'oscillateur qui cadence le microcontrôleur (et donc le DDS au travers de la sortie *timer*).

Le choix d'un synthétiseur numérique de fréquence cadencé à 70 MHz pourrait laisser penser que ce circuit ne peut fonctionner que jusqu'à une vingtaine de MHz dans les conditions nominales de filtrage des raies parasites générées par ce composant numérique. Il n'en est rien : lorsque le composant cadencé à f_{CK} est programmé pour générer un signal à f_o, les raies "parasites" sont déterministes et situées à f_{CK}, $f_{CK}-f_o$ et $f_{CK}+f_o$. En programmant $f_o = 25$ MHz, cette dernière raie se trouve à $70+25 = 95$ MHz, en plein dans la bande FM commerciale et donc détectable par tout récepteur FM, par exemple ceux équipant les téléphones portables. Le lecteur est donc encouragé à poursuivre l'exploration dans cette voie de transmission d'informations sur porteuse radiofréquence, la majorité des concepts de l'émission radiofréquence étant accessibles par ce circuit, avec la souplesse de la configuration logicielle du DDS par le microcontrôleur.

Références

[1] É. Carry. J.-M. Friedt, *L'environnement Arduino est-il approprié pour enseigner l'électronique embarquée ?*, conférence CETSIS (2017), disponible à `jmfriedt.free.fr/cetsis2017_arduino.pdf`

[2] *RM0401 – Reference manual, STM32F410 advanced ARM-based 32-bit MCUs*, (Octobre 2015), Doc ID 027812 Rev 2

[3] STM32F100 ARM Cortex-M3 FreeRTOS Demo à `www.freertos.org/FreeRTOS-for-Cortex-M3-STM32-STM32F100-Discov` `html`

[4] R. Barry, *Using the FreeRTOS real time kernel – A practical guide*, (2009) et sa mise à jour R. Barry, *Mastering the FreeRTOS Real Time Kernel* (2016) à `www.openrtos.net/Documentation/161204_Mastering_the_FreeRTOS_Real_Time_Kernel-A_Hands-On_Tutorial_Guide.pdf`

[5] D. Rabus, J.-M. Friedt, S. Ballandras, G. Martin, É. Carry, V. Blondeau-Patissier, *High-sensitivity open-loop electronics for gravimetric acoustic-wave-based sensors*, IEEE Trans. Ultrason. Ferroelectr. & Freq. Control. **60** (6), 1219—1226 (2013)