Programmation en C sur STM32

J.-M Friedt, 1er septembre 2019

Objectifs de ce document :
— compiler un programme pour microcontrôleur (STM32) sous GNU/Linux : faire clignoter une diode, exemples en C
— communication RS232, communication de données en ASCII
— timer, processus périodiques, interruptions
— conversion analogique-numérique et numérique-analogique de données, échantillonnage
— exploiter les fonctions de traitement du signal de GNU/Octave pour concevoir un filtre
— implémenter ce filtre dans un microcontrôleur avec des calculs sur des entiers uniquement
— valider expérimentalement le fonctionnement du filtre

1 Rappel de commandes Unix : recherches dans l'arborescence

Comment trouver le fichier d'entête qui contienne les définitions de types?
```
grep -r uint32 ~/sat/arm-none-eabi/include
```
Comment trouver l'emplacement d'un fichier requis lors de la compilation?
```
find ~/sat/arm-none-eabi/include -name *gpio.h -print
```
Comment recherche les occurences de la macro `GPIOC` dans les fichiers de la forme `gpio.h`?
```
find . -name *gpio.h -exec grep PORT_C '{}' \;
```

2 Premier pas sur STM32

FIGURE 1 – La carte STM32VL-discovery

Le STM32 [1], est une classe de microcontrôleurs proposés par ST autour du cœur ARM 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. La plate-forme utilisée est disponible pour un peu moins de 12 euros chez Farnell (référence 1824325) et ne nécessite aucun matériel additionnel pour fonctionner. Le microcontrôleur équipant cette carte est le bas de gamme de la série STM32 mais le cœur ARM Cortex-M3 est compatible avec toute la gamme de ce fabriquant. En particulier, les caractéristiques de cette carte sont :

— processeur STM32F100RB, cœur Cortex-M3 cadencé à 24 MHz, 128 KB flash, 8 KB RAM,
— 7 canaux DMA, 3 USARTs, 2 SPIs, 2 I2C, ADC, DAC, RTC, horloges, deux chiens de garde,
— interface ST-Link par USB pour déverminer/programmer,
— résonateur externe rapide (HSE) 8 MHz,
— résonateur externe lent (LSE) 32768 Hz.
— deux LEDs (vert, bleu) connectées au **port C** (broches **8 et 9**).

Un ouvrage conséquent d'exemples est proposé en [2]. Cependant, contrairement aux exemples de cet ouvrage, nous utiliserons une bibliothèque libre indépendante d'un vendeur et applicable à la majorité des processeurs architecturés autour des cœurs Cortex M3 et M4 : `libopencm3`). Le rôle de la bibliothèque est de cacher au néophyte les détails du fonctionnement du matériel pour ne donner accès qu'à des fonctions plus explicites du type `gpio_set()` ou `gpio_clear()`.

Nous nous engageons donc dans un premier temps sur un petit exercice de manipulation des ports numériques 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. 2 : on y trouvera la référence du port de communicatin asynchrone (USART), LEDs et convertisseur analogique-numérique (ADC) accessibles.

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 abstrairons 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.

1. `www.st.com/stm32`, et en particulier la série STM32F1xx [1] `www.st.com/web/en/resource/technical/document/reference_manual/CD00171190.pdf`

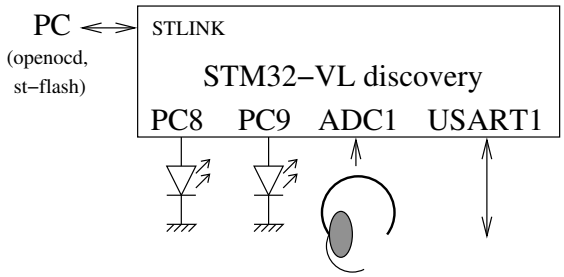


FIGURE 2 – Connections du STM32VL Discovery vers le monde extérieur.

Les exemples sont inspirés de [2] et des exemples de `libopencm3` distribués à `https://github.com/libopencm3/libopencm3-exa`

3 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). Bien que historiquement ST MicroElectronic propose une bibliothèque `libstm32`, nous favoriserons la bibliothèque libre `libopencm3` [2] pour ne pas devenir dépendants d'un fondeur tout en simplifiant l'accès aux ressources du STM32 (en évitant dans un premier temps de consulter les 1093 pages du manuel).

3.1 Clignotement de diode

L'exemple ci-dessous fait clignoter une diode :

```
#include <libopencm3/stm32/rcc.h> // ne PAS charger f1/rcc.h
#include <libopencm3/stm32/gpio.h>

static void gpio_setup(void)
{rcc_clock_setup_in_hse_8mhz_out_24mhz();
 rcc_periph_clock_enable(RCC_GPIOC);
 gpio_set_mode(GPIOC, GPIO_MODE_OUTPUT_2_MHZ, GPIO_CNF_OUTPUT_PUSHPULL, GPIO8|GPIO9);
}

int main(void)
{int i;
 gpio_setup();
 gpio_set(GPIOC,GPIO8);
 gpio_clear(GPIOC,GPIO9);

 while (1) {
   gpio_toggle(GPIOC, GPIO8);
   gpio_toggle(GPIOC, GPIO9);
   for (i=0; i<800000; i++) __asm__("nop");
 }
 return 0;
}
```

Listing 1 – Exemple de base initialisant les périphériques et faisant clignoter deux LEDs

Ce premier programme se contente de faire clignoter les deux diodes mais donne la séquence de base d'initialisation du microcontrôleur :

1. activer la source d'horloge du périphérique (GPIO)
2. définir la direction d'utilisation du périphérique (entrée ou sortie)
3. définir l'état initial du périphérique
4. une boucle infinie avec changement d'état des diodes et attente.

2. `http://libopencm3.org/wiki/Main_Page`

3.2 Compilation

Le programme se compile en complétant la commande de `gcc` par un certain nombre d'options :

```
arm-none-eabi-gcc -I~/sat/arm-none-eabi/include -fno-common -mthumb -mcpu=cortex-m3 -msoft-float -DSTM32F1

arm-none-eabi-gcc -o usart.elf fichier.o -L~/sat/arm-none-eabi/lib -lopencm3_stm32f1 --static \
  -Wl,--start-group -lc -lnosys -Wl,--end-group -T./stm32vl-discovery.ld -nostartfiles \
  -Wl,--gc-sections -mthumb -mcpu=cortex-m3 -msoft-float -mfix-cortex-m3-ldrd
```

Noter en particulier l'appel aux bibliothèques c et opencm3 par les options `-lc` et `-lopencm3_stm32f1` – `-lc` n'est utile que si nous faisons appel à `newlib`. Par ailleurs, `-I` renseigne la liste des répertoires contenant les fichiers d'entête (`.h`) et `-L` la liste des répertoires contenant les implémentations des bibliothèques. L'option `-T` fournit la carte mémoire indiquant au compilateur la taille de la RAM et de la mémoire flash non-volatile.

Cette commande se résume par le `Makefile` suivant :

```
#NAME=usart
NAME=adc-dac-timer
#NAME=adc-dac-printf
#NAME=filtre
#NAME=miniblink
#NAME=timer_jmf
all: $(NAME).bin

$(NAME).bin: $(NAME).elf
	arm-none-eabi-objcopy -Obinary $(NAME).elf $(NAME).bin

$(NAME).o: $(NAME).c
	arm-none-eabi-gcc -I$(HOME)/sat/arm-none-eabi/include \
	  -fno-common -mthumb -mcpu=cortex-m3 -msoft-float -DSTM32F1 -c $(NAME).c

$(NAME).elf: $(NAME).o
	arm-none-eabi-gcc -o $(NAME).elf $(NAME).o -lopencm3_stm32f1 --static -Wl,--start-group \
	      -lc -lnosys -Wl,--end-group -L$(HOME)/sat/arm-none-eabi/lib \
	      -T./stm32vl-discovery.ld -nostartfiles -Wl,--gc-sections \
	      -mthumb -mcpu=cortex-m3 -msoft-float -mfix-cortex-m3-ldrd

flash: $(NAME).bin
	st-flash write $(NAME).bin 0x8000000
	# st-flash write vl $(NAME).bin 0x8000000
clean:
	rm *.elf *.o
```

Lors du passage d'une famille de STM32 à l'autre, les 3 points auxquels il est bon de veiller sont

1. la fréquence de cadencement du cœur, 24 MHz pour le processeurs bas de gamme (VL), 72 MHz sinon,
2. la taille de la mémoire renseignée dans le *linker script* appelé par l'option `-T` (ici, `-T./stm32vl-discovery.ld`)
3. penser à préciser la famille du modèle de STM32 utilisé (par exemple `-DSTM32F10X_MD` pour la famille *Medium Density* du STM32F103).

Le listing correspondant se génère par :

```
arm-none-eabi-objdump -dSt usart.elf > usart.list
```

et la liste d'instructions aux divers formats attendus par les outils de programmation (format hexadécimal Intel ou Motorola, image binaire) à partir du fichier ELF :

```
arm-none-eabi-objcopy -Obinary usart.elf usart.bin
arm-none-eabi-objcopy -Oihex usart.elf usart.hex
arm-none-eabi-objcopy -Osrec usart.elf usart.srec
```

Rappel : `gcc -S` arrête la compilation à la génération de l'assembleur (fichier `.s`), `gcc -o` arrête la compilation à la génération des objets (avant le linker).

3.3 Transfert au microcontrôleur

Une fois le programme compilé, il est transféré au micocontrôleur par l'interface JTAG, ici émulée par le port série virtuel [3]. La commande pour transférer le programme `usart.bin` est :
`st-flash write usart.bin 0x8000000` [4]

On notera par ailleurs que l'outil pour transférer un programme au format binaire depuis le PC vers un processeur STM32F1 en dehors de son installation sur une carte Discovery est disponible sous la forme de `stm32flash`. Alternativement, il est possible de directement charger et exécuter le logiciel depuis le gestionnaire de lien JTAG `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 dernière solution 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`. 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.

Il arrive en effet qu'un programme soit difficile à déverminer (*debug*) et nécessite de sonder l'état de la mémoire du microcontrôleur en cours d'éxécution. `gdb` (GNU Debugger) fournit une telle fonctionnalité : l'exécution du programme est interrompue afin de permettre à l'utilisateur de sonder des informations telles que valeur d'une variable, état de la pile ou point d'arrêt.

Sur le circuit qui nous intéresse ici, l'interface de programmation communique avec `gdb` au moyen de `openocd`. Cet outil, classiquement utilisé pour contrôler les sondes JTAG, supporte le protcole de communication des cartes de démonstration STM32VL-Discovery.

`gdb` nécessite d'une part une interface de communication vers le microcontrôleur capable d'exécuter le code qui lui est destiné, et d'autre part le fichier original qui a été transmis vers la cible afin d'associer les points d'arrêt (*breakpoint*) et les variables à ds symboles compréhensibles par le développeur.

Nous lançons d'une part
`openocd -s /usr/local/bin/openocd-bin/share/openocd/scripts/ -f board/stm32vldiscovery.cfg \`
`-f interface/stlink-v1.cfg` pour créer le lien de communication entre `gdb` et le microcontrôleur, et d'autre part
`arm-none-eabi-gdb programme.elf` afin d'interfacer `gdb` (pour architecture ARM) avec le microcontrôleur en chargeant le fichier et la table des symboles associée `programme.elf`. La communication à proprement dit s'initialise par `target remote localhost:3333` et le programme est chargé en mémoire du microcontrôleur par la commande `load`. Une fois le programme en cours d'éxécution, il est possible d'en interrompre la séquence d'opération, définir des points d'arrêt (`breakpoint`), afficher la liste des instructions (`list`) ou afficher l'état de la pile (`backtrace`) ou de variables (`print variable`).

Cette méthode de travail est aussi applicable sur PC si l'on prend soin d'autoriser au préalable de l'exécution du programme qui crashe la génération d'un fichier image du contenu de la mémoire au moment du crash (`core dump`). La génération de ce fichier est désactivée par défaut sous GNU/Linux à cause de l'espace important requis par ces fichiers. Pour ce faire, `ulimit -c unlimited` et on exécute le programme trivialememt erroné ci-dessous

3. Le code source du programme `st-flash` est disponible à `https://github.com/texane/stlink`
4. avec la version pré-2017 de `st-flash`, on utilisera `st-flash write v1 usart.bin 0x8000000`.

```c
#include <stdio.h>

void assigne(char* c,int i)
{c[i]=i;}

int main()
{char c[4];
 int i;
 for (i=0;i<40960;i++) {
    assigne(c,i);
    printf("%d\n",i);}
}
```

qui se traduit par une violation d'accès à un segment mémoire qui n'est pas alloué à ce processus et la génération du fichier `core.pid`. gdb est alors invoqué par `gdb ./programme ./core.pid`. Afin d'accéder aux symboles, on aura pris soin de compiler avec option de déverminage `-g` et la commande `print i` nous informe de l'étape à laquelle le programme a crashé.

```
Program terminated with signal 11, Segmentation fault.
#0  assigne (i=8660, c=0xbf8d9e2c "") at core.c:4
4       {c[i]=i;}
(gdb) print i
$1 = 8660
(gdb) bt
#0  assigne (i=8660, c=0xbf8d9e2c "") at core.c:4
#1  main () at core.c:9
```

Le programme peut aussi être exécuté (`run`), depuis gdb, éventuellement pas à pas (`start` puis `step`), avec assignation de points d'arrêt (`tb ligne` pour définir le *breakpoint*). **Exercices :**

1. **compiler avec deux options d'optimisation, `-O2` et `-O0`. Quel constat sur la fréquence de clignotement des diodes ?**
2. **Faire clignoter alternativement les deux LEDs (PC8 et PC9). Varier la vitesse de clignotement.**

3.4 Compilation conditionnelle

Aussi étrange que cela puisse paraître, les fonctions d'initialisation n'ont pas le même nom selon que la cible soit le STM32F1 ou le STM32F4. Nous pourrons donc profiter de la compilation conditionnelle (instruction au préprocesseur `#ifdef`) pour sélectionner une ou l'autre architecture en fonction des drapeaux fournis à `gcc`. Une constante peut être soit définie dans les sources par `#define constante valeur`, soit lors de la compilation par le drapeau `-Dconstante=valeur` de `gcc`. Nous avons utilisé cette approche pour proposer les mêmes codes sources pour initialiser les deux familles de microcontrôleurs dans codes repris en annexe A.

4 Horloge et interruption

Afin de s'affranchir de la dépendance aux options de compilation, il *faut* cadencer les fonctions critiques en termes de latences non par du logiciel mais par le matériel dédié à ces fonctions : les horloges (*timer*). Ainsi, le même résultat que précédemment s'obtient par

```c
#include <libopencm3/stm32/rcc.h>
#include <libopencm3/stm32/gpio.h>
#include <libopencm3/stm32/timer.h>
#include <libopencm3/cm3/nvic.h>
#include <libopencm3/stm32/exti.h>

static void clock_setup(void) {rcc_clock_setup_in_hse_8mhz_out_24mhz();} // max 24 MHz

static void gpio_setup(void)
{ rcc_periph_clock_enable(RCC_GPIOC);    // LED
  gpio_set_mode(GPIOC, GPIO_MODE_OUTPUT_50_MHZ, GPIO_CNF_OUTPUT_PUSHPULL, GPIO8|GPIO9); // LED output
  gpio_set(GPIOC, GPIO9);                                          // initial state
  gpio_clear(GPIOC, GPIO8);
```

```
}

static void tim_setup(void)
{ rcc_periph_clock_enable(RCC_TIM2);                              // TIM2 clock
  nvic_enable_irq(NVIC_TIM2_IRQ);                                 // TIM2 interrupt
  rcc_periph_reset_pulse(RST_TIM2);     // timer_reset(TIM2);     // Reset TIM2 -- old version
// voir https://github.com/libopencm3/libopencm3-examples/commits/master/examples/stm32/f1/lisa-m-2/adc_injec_timtrig
// correction 01/09/2019
  timer_set_mode(TIM2, TIM_CR1_CKD_CK_INT, TIM_CR1_CMS_EDGE, TIM_CR1_DIR_UP);
  timer_set_prescaler(TIM2, 90);          // prescaler value.
  timer_disable_preload(TIM2);            // Enable preload
  timer_continuous_mode(TIM2);            // Continous mode
  timer_set_period(TIM2, 65535);          // Period
  timer_disable_preload(TIM2);            // ARR reload enable
  timer_enable_counter(TIM2);             // Counter enable.
  timer_enable_irq(TIM2, TIM_DIER_CC1IE); // Enable commutation interrupt.
}

void tim2_isr(void)   // isr = Interrupt Service Routine, fonction de gestion de l'interruption
{ if (timer_get_flag(TIM2, TIM_SR_CC1IF)) {
     timer_clear_flag(TIM2, TIM_SR_CC1IF);
     gpio_toggle(GPIOC, GPIO8 | GPIO9);
    }
}

int main(void)
{ clock_setup();
  gpio_setup();
  tim_setup();
  while (1) __asm("nop"); // ce programme ne fait ... rien !
  return 0;
}
```

Le *timer* se configure en divisant l'horloge cadençant le périphérique (*prescaler*) puis en définissant la période sous forme de valeur à atteindre par le compteur pour réinitialiser le décompte et déclencher une interruption.

Rechercher dans la datasheet la section concernant les *timers* et justifier de la fréquence de clignotement de la LED.

Quel paramètre modifier afin de changer cette période? le vérifier.

Noter que le programme principal ne fait rien! La commutation de l'état des diodes se fait dans le gestionnaire d'interruption qui n'est pas explicitement appelé mais déclenché par un évènement matériel.

Trouver la liste des gestionnaires d'interruptions du STM32.

5 Communiquer par RS232

Le STM32 est muni d'au moins deux interfaces de communication asynchrone (USART). L'interface connectée au convertisseur RS232-USB de la carte de développement est l'USART1. L'exemple qui suit aborde deux aspects :

- communication sur port asynchrone (RS232) avec initialisation des périphériques, initialisation des horloges, et initialisation du port de communication (protocole d'échange des bits). La fonction principale se charge alors de communiquer des caractères sur le port série,
- exploitation des bibliothèques standard du C (`stdio` et `stdlib`) et, en particulier dans cet exemple `printf()` qui nécessite le point d'entrée (*stub*) `_write()`.

Côté PC, la communication par port série se fait au moyen de `minicom` : `CTRL A-O` pour définir le port (`/dev/ttyUSB0`) et le contrôle de flux (ni matériel, ni logiciel). `CTRL A-P` pour définir la vitesse (ici 115200 bauds, N81).

```
#include <libopencm3/stm32/rcc.h>
#include <libopencm3/stm32/gpio.h>
#include <libopencm3/stm32/usart.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

static void clock_setup(void)
{ rcc_clock_setup_in_hse_8mhz_out_24mhz();
  rcc_periph_clock_enable(RCC_GPIOC); // Enable GPIOC clock
  rcc_periph_clock_enable(RCC_GPIOA); // Enable GPIOA clock
  rcc_periph_clock_enable(RCC_USART1);
}

static void usart_setup(void)
{ // Setup GPIO pin GPIO_USART1_TX/GPIO9 on GPIO port A for transmit. */
  gpio_set_mode(GPIOA, GPIO_MODE_OUTPUT_50_MHZ,
      GPIO_CNF_OUTPUT_ALTFN_PUSHPULL, GPIO_USART1_TX);

  usart_set_baudrate(USART1, 115200);   // USART_BRR(USART1) = (uint16_t)((24000000<<4)/(38400*16));
  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);
}

static void gpio_setup(void)
{ gpio_set_mode(GPIOC, GPIO_MODE_OUTPUT_2_MHZ, GPIO_CNF_OUTPUT_PUSHPULL, GPIO9|GPIO12);
}

int main(void)
{ int i, c = 0;
  clock_setup();
  gpio_setup();
  usart_setup();
  while (1) {
    gpio_toggle(GPIOC, GPIO9);
    gpio_toggle(GPIOC, GPIO12);
    c = (c == 9) ? 0 : c + 1; // cyclic increment c
#ifndef avec_newlib
    usart_send_blocking(USART1, c + '0'); // USART1: send byte
    usart_send_blocking(USART1, '\r');
    usart_send_blocking(USART1, '\n');
#else
    printf("%d\r\n", (int)c);
#endif
    for (i = 0; i < 800000; i++) __asm__("NOP");
  }
  return 0;
}

#ifdef avec_newlib
int _write(int file, char *ptr, int len)
{int i;
 for (i = 0; i < len; i++) usart_send_blocking(USART1, ptr[i]);
 return i;
}
#endif
```

Nous avons vu que l'utilisation de la bibliothèque d'entrée-sortie standard (`stdio`), et en particulier son implémentation pour systèmes embarqués, est gourmande en ressources. Pour s'en convaincre, compiler le programme avec (`#define avec_newlib`) et sans (commenter cette ligne). Nous constatons une augmentation significative du temps de programmation du microcontrôleur associée à une croissance significative de la taille du code – de 1580 octets à 29032 – qui inclut maintenant toutes ces fonctions.

Exercice :

1. constater le bon fonctionnement du programme en observant les trames communiquées sous `minicom`,
2. que se passe-t-il si un mauvais débit de communication est sélectionné?
3. activer ou désactiver les fonctions standard du C implémentées par `newlib`. Quelle conséquence? analyser en particulier ce que fait la fonction `_write()`, un des *stubs* de newlib.

4. envoyer une valeur hexadécimale (par exemple 0x55 et 0xAA) en ASCII, *i.e.* définir deux variables avec ces valeurs et écrire la fonction qui permette d'en afficher le contenu sous `minicom`,
5. envoyer une valeur en décimal (par exemple 250) en ASCII
6. sachant qu'une chaîne de caractères est représentée en C par un tableau de caractères se terminant par le caractère de valeur 0, ajouter une fonction d'envoi de chaîne de caractères.
7. démontrer le bon fonctionnement de cette fonction en affichant le message `Hello World` au moyen d'une fonction prenant en argument un tableau de caractères.

L'interface de communication RS232 est le mode de communication privilégié entre un système embarqué et l'utilisateur. À peu près tous les circuits, même aussi complexes qu'un disque NAS Lacie [5], un routeur ethernet [6] ou NeufBox [7], exploitent ce genre d'interface en phase de développement.
⚠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!

Tous ces exemples seront repris dans le contexte de l'introduction à FreeRTOS en consultant les codes disponibles à `https://github.com/jmfriedt/tp_freertos/0no_freertos` et en particulier `main_cm3.c` et son Makefile associé `Makefile.f1`.

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
```

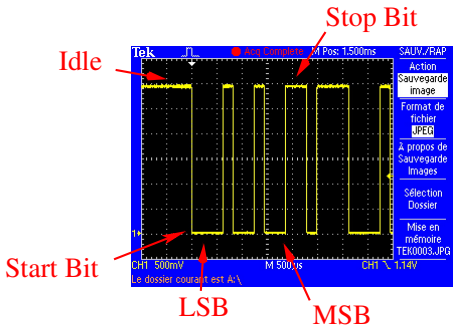


FIGURE 3 – Chronogramme de la liaison asynchrone.

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.

6 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);
```

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
```

5. http://lacie.nas-central.org/wiki/Serial_port_%282Big_2%29
6. http://maschke.de/wag54g/
7. http://www.neufbox4.org/wiki/index.php?title=Port_s%C3%A9rie

7 Lire une valeur analogique

Tous les programmes réalisés jusqu'ici vont nous permettre d'atteindre le but de ce TP : mesurer une grandeur physique à intervalles de temps réguliers et retranscrire ces informations pour les rendre exploitables par un utilisateur. L'exemple ci-dessous est une simple lecture périodique de température sur la sonde interne au microcontrôleur :

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

int main(void)
{volatile int i, c = 0;
  short res;
  Usart1_Init();
  Led_Init();
  adc_setup();

  while (1) {
    res=read_adc_naiive(1);
    uart_putc('0'); // USART: send byte
    uart_putc('x'); // USART: send byte

    affshort(res);
    uart_putc('\n'); uart_putc('\r');
    Led_Hi2();
    for (i = 0; i < 800000; i++) __asm__("NOP");
    Led_Lo2();
  }
  return 0;
}
```

avec initialisation des ADCs qu'il faudra compléter, des horloges associées ainsi que des broches. Le canal 16 est un cas particulier : il s'agit d'une sonde interne de température. Prendre soin cependant de l'activer pour l'utiliser.

```
//#include <stm32f1/stm32f10x.h>     // definition uint32_t
#include <libopencm3/cm3/common.h> // BEGIN_DECL
#include <errno.h>
#include <stdio.h>
#include <unistd.h>

#include <libopencm3/cm3/nvic.h>
#include <libopencm3/stm32/gpio.h>
#include <libopencm3/stm32/usart.h>
#include <libopencm3/stm32/f1/rcc.h>
#include <libopencm3/stm32/f1/adc.h>

int _write(int file, char *ptr, int len);

#define nbr 256

static void clock_setup(void)
{ rcc_clock_setup_in_hse_8mhz_out_24mhz();

  rcc_periph_clock_enable(RCC_GPIOC); // port C (diodes)
  rcc_periph_clock_enable(RCC_GPIOA); // port A (USART1)
  rcc_periph_clock_enable(RCC_USART1); // USART1
  rcc_periph_clock_enable(RCC_ADC1);
}

static void usart_setup(void)
{ gpio_set_mode(GPIOA, GPIO_MODE_OUTPUT_50_MHZ,
    GPIO_CNF_OUTPUT_ALTFN_PUSHPULL, GPIO_USART1_TX);

  [... voir auparavant]
}

int _write(int file, char *ptr, int len)
{
  [... voir auparavant]
}

```

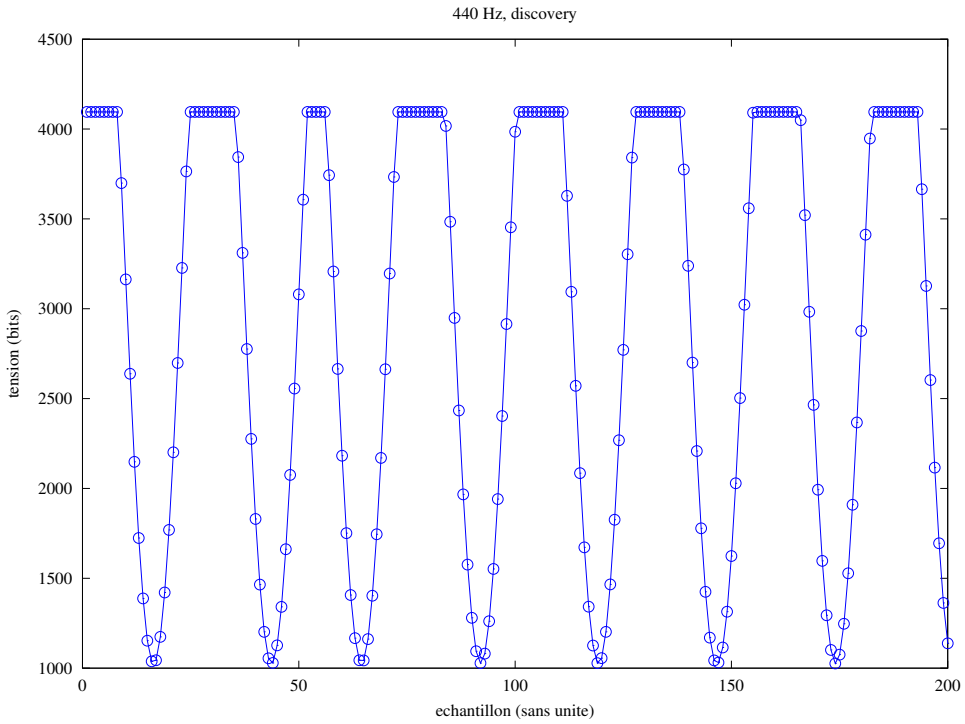
```
static void adc_setup(void)
{ int i;
  gpio_set_mode(GPIOA, GPIO_MODE_INPUT, GPIO_CNF_INPUT_ANALOG, GPIO0|GPIO1|GPIO2);
  adc_power_off(ADC1);
  adc_disable_scan_mode(ADC1);            // single conversion
  adc_set_single_conversion_mode(ADC1);
  adc_disable_external_trigger_regular(ADC1);
  adc_set_right_aligned(ADC1);
  adc_set_sample_time_on_all_channels(ADC1, ADC_SMPR_SMP_28DOT5CYC);
  adc_power_on(ADC1);
  for (i = 0; i < 800000; i++) __asm__("nop"); // demarrage ADC ...
  adc_reset_calibration(ADC1);
  adc_calibrate(ADC1);
}

static uint16_t read_adc_naiive(uint8_t channel)
{ uint8_t channel_array[16];
  channel_array[0] = channel;                 // choix du canal de conversion
  adc_set_regular_sequence(ADC1, 1, channel_array);
  adc_start_conversion_direct(ADC1);
  while (!adc_eoc(ADC1));                     // attend la fin de conversion
  return(adc_read_regular(ADC1));
}

int main(void)
{int i,tab[nbr];
 int j = 0;
 clock_setup();
 usart_setup();
 gpio_set_mode(GPIOC, GPIO_MODE_OUTPUT_2_MHZ, GPIO_CNF_OUTPUT_PUSHPULL, GPIO8);
 gpio_set(GPIOC, GPIO8);
 adc_setup();

 while (1) {
      for (i=0;i<nbr;i++) tab[i]= read_adc_naiive(1);
      for (i=0;i<nbr;i++) printf("%d\n", tab[i]);
      gpio_toggle(GPIOC, GPIO8); /* LED on/off */
      for (i = 0; i < 100000; i++) __asm__("NOP");
 }
 return 0;
}
```

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);`

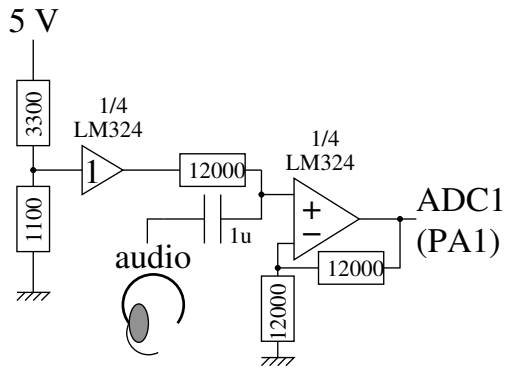


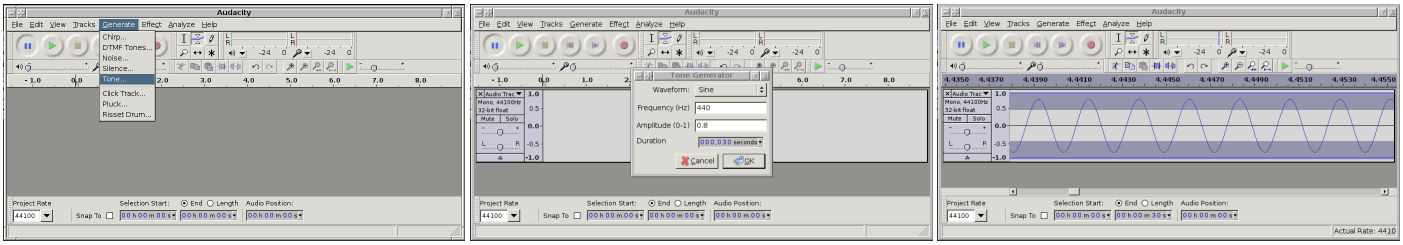
FIGURE 4 – Ajout d'un T de polarisation pour décaller la tension de valeur moyenne nulle en sortie de la carte son vers la demi-tension de référence du convertisseur analogique-numérique.

Afin de générer des signaux de l'ordre de quelques dizaines de Hz à quelques kilohertz, nous allons exploiter la carte son qui équipe la majorité des ordinateurs personnels. Un logiciel, `audacity`[8] propose de synthétiser un signal et de l'émettre sur la sortie de sa carte son (Fig. 4).

Sous GNU/Linux, un outil de configuration en ligne de commande, `play`, est fourni avec la boite à outils pour fichiers son `sox`. Pour générer un signal à fréquence fixe sur la carte son : `play -n synth sin 440 gain -5` (éviter un gain trop élevé pour ne pas saturer l'étage de sortie de la carte son, qui se configure par ailleurs au moyen de `alsamixer`).

Pour balayer la fréquence de sortie (fonction *chirp* de `audacity`) :
`while [ TRUE ]; do play -n synth 3 sin 440-660 gain -5;done`

8. disponible pour GNU/Linux, MacOS X et MS-Windows à `http://audacity.sourceforge.net/`



Nous nous contenterons pour le moment de générer une sinusoïde, pour n'exploiter que plus tard les formes d'ondes plus complexes (*chirp*).

Exercices :

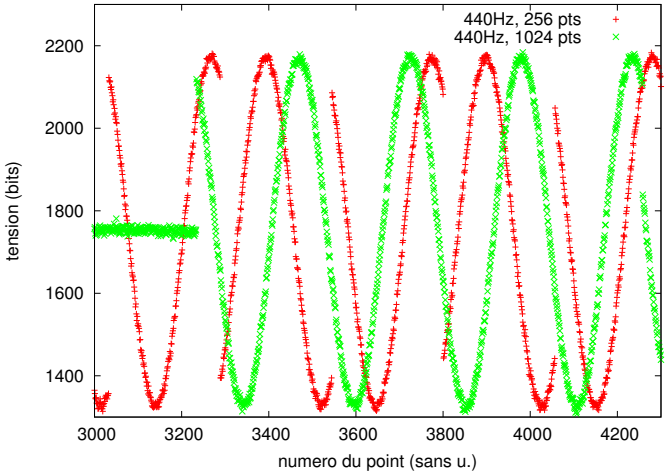
1. enregistrer une séquence issue de la carte son du PC
2. visualiser au moyen de GNU/Octave ou `gnuplot` le résultat de l'acquisition (commandes identiques à celles de Matlab, voir section 11 pour un rappel)
3. établir la fréquence d'échantillonnage.

Nous pouvons suivre deux stratégies pour **établir la fréquence d'échantillonnage** f_e : temporelle ou fréquentielle. Dans le premier cas nous comptons le nombre M d'échantillons dans N périodes et établissons, connaissant la fréquence f_s du signal mesuré, alors

$$f_e = \frac{M}{N} f_s$$

D'un point de vue fréquentiel, la transformée de Fourier sur M points d'un signal numérisé à f_e s'étend de $-f_e/2$ à $+f_e/2$. Connaissant la fréquence de ce signal numérisé qui se présente dans la transformée de Fourier comme une raie de puissance maximale en position N, nous déduisons que

$$N/M = f_s/f_e \Leftrightarrow f_e = \frac{M}{N} f_s$$



7.1 GNURadio et `gnuradio-companion`

Un outil de traitement numérique du signal est `gnuradio` accompagné de son interface graphique `gnuradio-companion`. Cet outil permet d'accéder à divers périphériques – dont la carte son du PC – pour acquérir et générér des signaux (éventuellement simultanément), ainsi que traiter ces signaux pour en extraire les signaux. Les modes de modulation les plus courants y sont fournis ainsi que divers outils classiques tels que l'analyse de Fourier, oscilloscope ... Cet outil *opensource* est particulièrement favorable à l'ajout de ses propres fonctions de traitement selon des modalités d'analyses inspirées des blocs existant (Fig. 5).

8 Déclencher une lecture de valeur analogique sur *timer*

Nous avons vu auparavant que l'inconvénient de cadencer une opération sur une attente en boucle vide est la dépendance de la vitesse d'exécution avec la nature et version du compilateur ou avec les options d'optimisation. Une utilisation naturelle des horloges internes est de cadencer la conversion analogique-numérique. Une telle fonctionnalité est cablée au niveau du matériel du STM32 et ne nécessite qu'une configuration appropriée pour garantir la fréquence d'échantillonage conditionnée par l'horloge interne au processeur.

```
#include <libopencm3/stm32/rcc.h>
#include <libopencm3/stm32/flash.h>
#include <libopencm3/stm32/gpio.h>
#include <libopencm3/stm32/adc.h>
#include <libopencm3/stm32/usart.h>
```

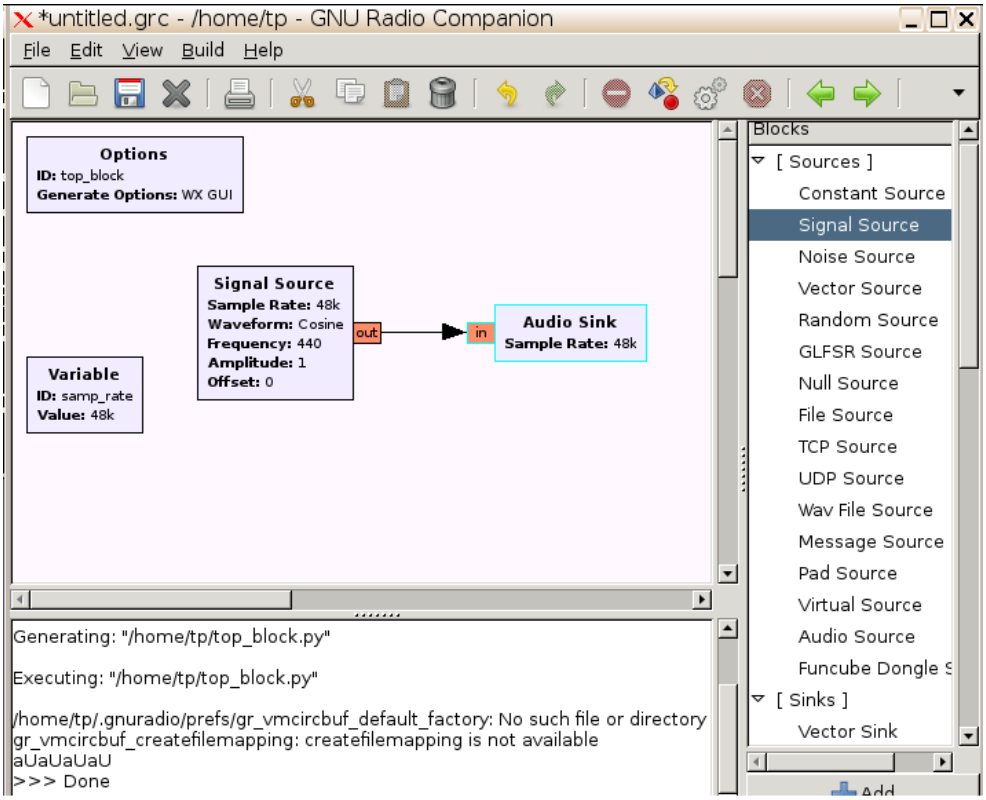


FIGURE 5 – Graphique `gnuradio-companion` pour générer un signal périodique sinusoïdal sur la sortie audio du PC.

```
#include <libopencm3/stm32/timer.h>
#include <libopencm3/cm3/nvic.h>

volatile uint16_t temperature[256] ;
volatile int jmf_index=0;

static void usart_setup(void)
{
  rcc_periph_clock_enable(RCC_USART1);
  /* Setup GPIO pin GPIO_USART1_TX/GPIO9 on GPIO port A for transmit. */
  gpio_set_mode(GPIOA, GPIO_MODE_OUTPUT_50_MHZ,
     GPIO_CNF_OUTPUT_ALTFN_PUSHPULL, GPIO_USART1_TX);

        /* Setup UART parameters. */
  usart_set_baudrate(USART1, 115200);
  usart_set_databits(USART1, 8);
  usart_set_stopbits(USART1, USART_STOPBITS_1);
  usart_set_mode(USART1, USART_MODE_TX_RX);
  usart_set_parity(USART1, USART_PARITY_NONE);
  usart_set_flow_control(USART1, USART_FLOWCONTROL_NONE);

  /* Finally enable the USART. */
  usart_enable(USART1);
}

static void gpio_setup(void)
{ rcc_periph_clock_enable(RCC_GPIOA);
  rcc_periph_clock_enable(RCC_GPIOC);
  gpio_set_mode(GPIOA, GPIO_MODE_INPUT, GPIO_CNF_INPUT_ANALOG, GPIO0);
  gpio_set_mode(GPIOA, GPIO_MODE_INPUT, GPIO_CNF_INPUT_ANALOG, GPIO1);
  gpio_set_mode(GPIOC, GPIO_MODE_OUTPUT_2_MHZ, GPIO_CNF_OUTPUT_PUSHPULL, GPIO8|GPIO9);
}

```

```
static void timer_setup(void)
{ uint32_t timer;
  uint32_t rcc_apb;
  timer = TIM2;
  rcc_periph_clock_enable(RCC_TIM2);
  gpio_set_mode(GPIOA, GPIO_MODE_INPUT, GPIO_CNF_INPUT_ANALOG, GPIO0);
  rcc_periph_reset_pulse(RST_TIM2);     // timer_reset(TIM2);     // Reset TIM2 -- old version
  timer_set_mode(timer, TIM_CR1_CKD_CK_INT, TIM_CR1_CMS_EDGE, TIM_CR1_DIR_UP);
  timer_set_period(timer, 0xF*5);
  timer_set_prescaler(timer, 0x8);
  timer_set_clock_division(timer, 0x0);
  timer_set_master_mode(timer, TIM_CR2_MMS_UPDATE); // Generate TRGO on every update
  timer_enable_counter(timer);
}

static void irq_setup(void)
{ nvic_set_priority(NVIC_ADC1_2_IRQ, 0);
  nvic_enable_irq(NVIC_ADC1_2_IRQ);
}

static void adc_setup(void)
{ int i;
  rcc_periph_clock_enable(RCC_GPIOA);
  rcc_periph_clock_enable(RCC_ADC1);
  gpio_set_mode(GPIOA, GPIO_MODE_INPUT, GPIO_CNF_INPUT_ANALOG, GPIO0);
  gpio_set_mode(GPIOA, GPIO_MODE_INPUT, GPIO_CNF_INPUT_ANALOG, GPIO1);
  adc_power_off(ADC1);       // Make sure the ADC doesn't run during config.
  adc_enable_scan_mode(ADC1);
  adc_set_single_conversion_mode(ADC1);
  adc_enable_external_trigger_injected(ADC1,ADC_CR2_JEXTSEL_TIM2_TRGO); // start with TIM2 TRGO
  adc_enable_eoc_interrupt_injected(ADC1);                              // Generate the ADC1_2_IRQ
  adc_set_right_aligned(ADC1);
//adc_enable_temperature_sensor(ADC1);
  adc_set_sample_time_on_all_channels(ADC1, ADC_SMPR_SMP_28DOT5CYC);
  adc_power_on(ADC1);

  for (i = 0; i < 800000; i++) /* Wait a bit. */
      __asm__("nop");          /* Wait for ADC starting up. */
  adc_calibrate(ADC1);
  while ((ADC_CR2(ADC1) & ADC_CR2_RSTCAL) != 0);
  adc_calibrate(ADC1);
  while ((ADC_CR2(ADC1) & ADC_CR2_CAL) != 0);
}

static void my_usart_print_int(uint32_t usart, int value)
{ int8_t i;
  uint8_t nr_digits = 0;
  char buffer[25];
  if (value < 0) {usart_send_blocking(usart, '-');value=-value;}
  while (value > 0) {buffer[nr_digits++] = "0123456789"[value % 10];value /= 10;}
  for (i = (nr_digits - 1); i >= 0; i--) {usart_send_blocking(usart, buffer[i]);}
  usart_send_blocking(usart, '\r');usart_send_blocking(usart, '\n');
}

int main(void)
{uint8_t channel_array[16];
 int k;
 rcc_clock_setup_in_hse_8mhz_out_24mhz();
 gpio_setup();
 usart_setup();
 timer_setup();
 irq_setup();
 adc_setup();

 gpio_set(GPIOC, GPIO8|GPIO9);
 channel_array[0] = 0; // channel number for conversion
 adc_set_injected_sequence(ADC1, 1, channel_array);
 while (1) {
   if (jmf_index>=255)
      {for (k=0;k<256;k++) my_usart_print_int(USART1, temperature[k]);
```

```
        jmf_index=0;
110     }
    gpio_toggle(GPIOC, GPIO8);
112  }
   return 0;
114 }

116 void adc1_2_isr(void)
  { ADC_SR(ADC1) &= ~ADC_SR_JEOC;
118   if (jmf_index<256) {temperature[jmf_index] = adc_read_injected(ADC1,1);jmf_index++;}
  }
```

9 Conversion numérique-analogique

Implémenter une loi de commande suppose être capable d'interagir avec le système observé de fọn continue, ou tout au moins avec plus d'états que la simple commande binaire. Un convertisseur numérique-analogique (DAC) fonctionnant sur N bits et configuré par un mot M génère une tension (sortie analogique) V comprise entre 0 V et une tension de référence V_{REF} selon

$$V = \frac{V_{REF} \cdot M}{2^N}$$

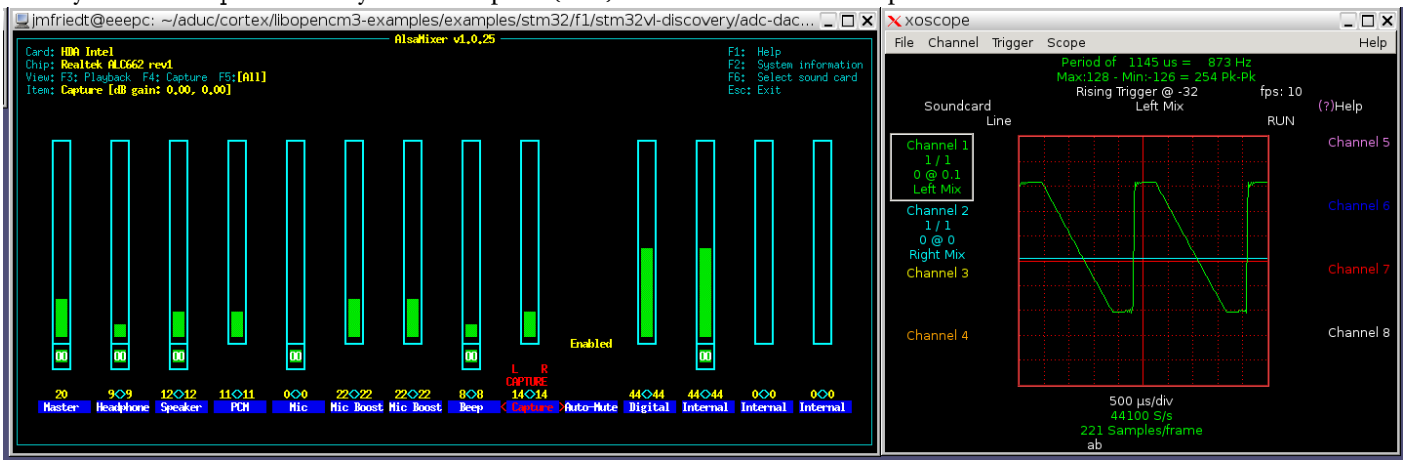
Le DAC s'initialise par

```
rcc_peripheral_enable_clock(&RCC_APB1ENR, RCC_APB1ENR_DACEN);
gpio_set_mode(GPIOA, GPIO_MODE_OUTPUT_50_MHZ, GPIO_CNF_OUTPUT_ALTFN_PUSHPULL, GPIO4);
dac_disable(CHANNEL_1);
dac_disable_waveform_generation(CHANNEL_1);
dac_enable(CHANNEL_1);
dac_set_trigger_source(DAC_CR_TSEL2_SW);
```

puis s'utilise au moyen de

```
int target=0;
dac_load_data_buffer_single(target, RIGHT12, CHANNEL_1);
dac_software_trigger(CHANNEL_1);
target+=3; if (target>3000) target=0;
```

Le résultat de la génération de rampes sur le convertisseur numérique-analogique s'observe sur l'entrée de la carte son soit au moyen de `xoscope` ou au moyen du bloc puits (*sink*) de GNURadio oscilloscope :



Cette figure illustre l'enregistrement par carte son (logiciel `xoscope` pour exploiter la carte son comme oscilloscope) lorsque le convertisseur numérique-analogique est configuré pour générer des signaux en dent de scie. La carte son inverse le signal en entrée et est configurée (activation de l'entrée, gain) au moyen de `alsamixer`.

10 Accès carte mémoire

Les microcontrôleurs ne fournissent pas une puissance de calcul importante (quoi que ...) mais sont capables d'effectuer un certain nombre d'opérations simples telles qu'asservissements et acquisitions de données. Dans tous les cas, par soucis de

contrôle *a posteriori* ou de dupliquer en local les données transmises en temps réel, les applications de ces microcontrôleurs sont étendues par la possibilité de stocker ces informations sur un support peu coûteux et consommant peu de courant en mode veille. Nous nous proposons d'étudier le stockage d'informations sur cartes mémoires de type MutliMediaCard (MMC) ou Secure Digital (SD) comme cas ludique d'application de la communication synchrone sur bus SPI.

Afin de gagner du temps, nous ne détaillerons pas le protocole d'initialisation et de communication d'une carte SD : nous utiliserons une bibliothèque mise à disposition par son auteur pour conserver et relire des données sur support de stockage de masse non volatile. Nous avons abordé en détail le protocole de communication sur bus synchrone SPI et les étapes d'initialisation de la cart mémoire dans [3].

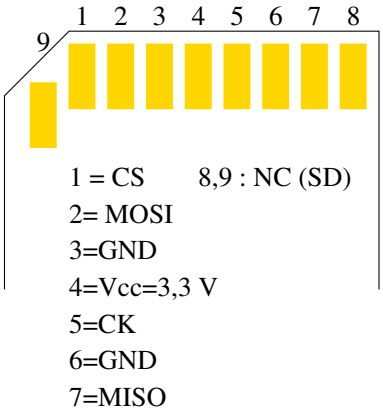


FIGURE 6 – Assignation des broches d'une carte SD. Les broches 8 et 9 n'existent pas dans les cartes MMC, qui fonctionnent sinon de la même façon.

Les cartes SD fonctionnent en deux mode : le mode natif, rapide et supportant l'encryption des données, et le mode SPI, plus lent mais facilement accessible depuis n'importe quel microcontrôleur. Nous utiliserons ce second mode, dans lequel seules 4 broches sont utiles (fig. 6). Le stockage sur carte SD en mode natif, bloc par bloc, est efficace du point de vue du microcontrôleur mais ne permet pas de relire les informations depuis un ordinateur personnel qui n'est pas informé du format de stockage. Afin de respecter une norme connue par les systèmes d'exploitation exécutés sur ordinateurs personnels, il est judicieux de recherche un format de stockage encore d'actualité tout en étant compatible avec les ressources de calcul d'un petit microcontrôleur : le format MS-DOS FAT répond à ces exigences. Il reste néanmoins lourd à implémenter et surtout à déverminer : nous profiterons donc des implémentations libres disponibles sur le web [3].

11 Afficher des données

Deux outils sont mis à disposition pour afficher des courbes : `gnuplot` et `octave` [9].
`gnuplot` affiche des courbes lues dans des fichiers :

```
plot "fichier"
```

on peut préciser les colonnes à afficher, l'axe des abscisses et ordonnées... :

```
plot "fichier" using 2:3 with lines
```

pour utiliser la seconde colonne comme abscisse et la troisième colonne comme ordonnée, les points étant reliés par des lignes. `help plot` pour toutes les options.

`octave` est une version opensource de Matlab et en reprend toutes les syntaxes (pour ce qui nous concerne ici, `load` et `plot`). Noter que contrairement à `gnuplot` qui peut extraire des colonnes de données d'à peu près m'importe que format de fichier ASCII (*i.e.* quelquesoit le nombre de colonnes par ligne), Matlab et GNU/Octave exigent de charger des fichiers contenant une matrice dont le nombre de colonnes est constant dans toutes les lignes, et sans chaînes de caractères autres que les commentaires commençant par %. Seule fonctionnalité nouvelle de GNU/Octave par rapport à Matlab qui nous sera utile ici : `octave` peut lire des données au format hexadécimal :

9. Pour une description plus exhaustive des fonctionnalités de `gnuplot` et `octave`, on pourra consulter `http://jmfriedt.free.fr/lm_octave.pdf`.

```
f=fopen("fichier","r");d=fscanf("%x");fclose(d);
```

La variable `d` contient alors les informations qui étaient stockées dans le fichier `fichier` au format hexadécimal. Si le fichier contenait `N` lignes, les données sont lues par colonne et les informations d'une colonne donnée d'obtiennent par `d(1:N:length(d))`.

12 Conception d'un filtre de réponse finie (FIR)

Le sujet de cette partie n'est pas de développer la théorie des filtres numériques mais uniquement de rappeler quelques uns des concepts utiles en pratique :

— un filtre de réponse impulsionnelle finie [5] (Finite Impulse Response, FIR) b_m est caractérisé par une sortie y_n à un instant donné n qui ne dépend que des valeurs actuelle et passées de la mesure x_m, $m \leq n$:

$$y_n = \sum_{k=0..m} b_k x_{n-k}$$

L'application du filtre s'obtient pas convolution des coefficients du filtre et des mesures.

— un filtre de réponse impulsionnelle infinie (IIR) a une sortie qui dépend non seulement des mesures mais aussi des valeurs passées du filtre.

— GNU/Octave (et Matlab) proposent des outils de synthèse des filtres FIR et IIR. Par soucis de stabilité numérique et simplicité d'implémentation, nous ne nous intéresserons qu'aux FIR. Dans ce cas, les coefficient de récursion sur les valeurs passées du filtre (nommées en général a_m dans la littérature) seront égaux à $a = 1$.

Le calcul d'un filtre dont la réponse spectrale ressemble à une forme imposée par l'utilisateur (suivant un critère des moindres carrés) est obtenu par la fonction `firls()`. Cette fonction prend en argument l'ordre du filtre (nombre de coefficients), la liste des fréquences définissant le gabarit du filtre et les magnitudes associées. Elle renvoie la liste des coefficients `b`.

L'application d'un FIR de coefficients `b` (avec `a=1`) ou de façon plus générale d'un IIR de coefficients `a` et `b` sur un vecteur de données expérimentales `x` s'obtient par `filter(b,a,x);`

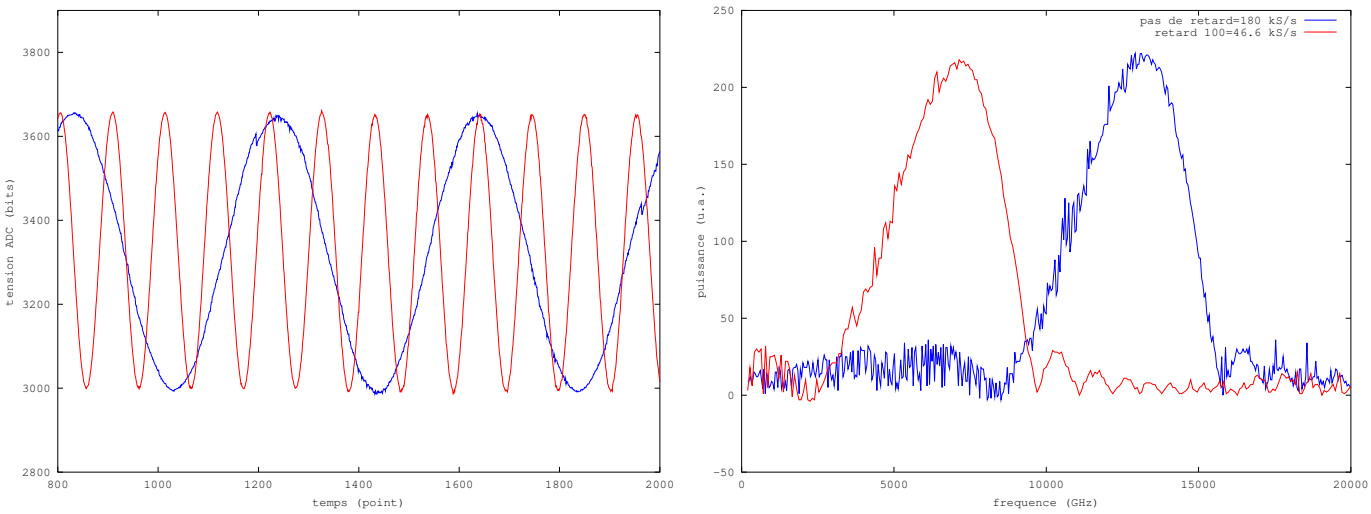


FIGURE 7 – Gauche : le même signal (sinusoïde à 440 Hz) échantilloné à deux fréquences différentes. Droite : les mêmes coefficients de filtres appliqués à un *chirp* balayant de 200 à 20000 Hz mais pour des fréquences d'échantillonage différentes.

Tout filtre se définit en terme de fréquence normalisée : la fréquence 1 correspond à la demi-fréquence d'échantillonnage (fréquence de Nyquist) lors de la mesure des données.

Exercices :

1. en supposant que nous échantillonnons les données sur un convertisseur analogique-numérique à 16000 Hz, identifier les coefficients d'un filtre passe-bande entre 700 et 900 Hz. Combien faut-il de coefficients pour obtenir un résultat satisfaisant ?
2. Quelle est la conséquence de choisir trop de coefficients ? Quelle est la conséquence de choisir trop peu de coefficients ?
3. ayant obtenu la réponse impulsionnelle du filtre, quelle est sa réponse spectrale ?

4. valider sur des données (boucle sur les fréquences) simulées la réponse spectrale du filtre

Au lieu de boucler sur des fréquences et observer la réponse du filtre appliqué à un signal monochromatique, le `chirp` est un signal dont la fréquence instantanée évolue avec le temps. Ainsi, la fonction
`chirp([0:1/16000:5],40,5,8000);`
génère un signal échantillonné à 16 kHz, pendant 5 secondes, balayant la gamme de fréquences allant de 40 Hz à 8 kHz.

Exercices :

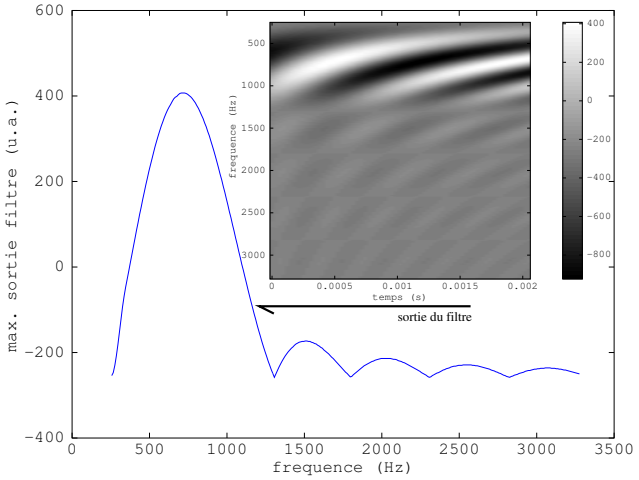
1. appliquer le filtre identifié auparavant au chirp, et visualiser le signal issu de cette transformation
2. que se passe-t-il si la fréquence de fin du chirp dépasse 8 kHz dans cet exemple ?

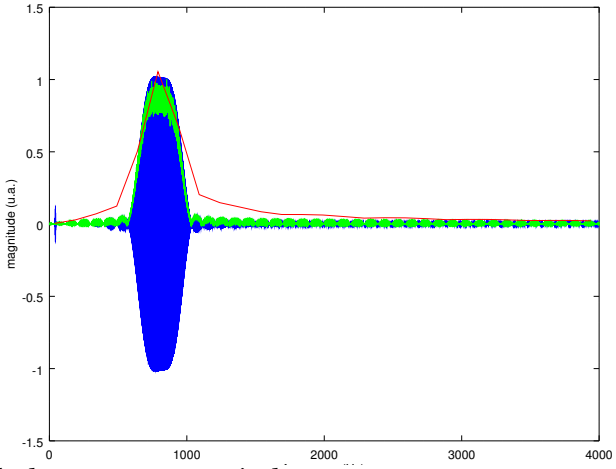
13 Application pratique du filtre

Exercices : Implémenter sur STM32

1. sachant que les acquisitions se font sur 12 bits, quelle est la plage de valeurs sur laquelle se font les mesures ?
2. comment exploiter les coefficients numériques du filtre pour un calcul sur des entiers sur 16 bits (short) ? sur 32 bits (long) ?

1. l'échantillonnage périodique de mesures analogiques acquises sur ADC1
2. l'affichage du résultat de ces mesures
3. le filtrage de ces mesures par un FIR synthétisé selon la méthode vue plus haut
4. l'affichage du résultat de ce filtrage
5. l'affichage de l'amplitude du signal résultant du filtrage
6. appliquer ce programme à une mesure sur un chirp généré par la carte son du PC au moyen de `audacity`
7. comparer l'expérience et la simulation.






```
fe=16000;
ffin=4000;
fdeb=40;
% excitation par un chirp
b=firls(160,[0 600 700 900 1000 fe/2]/fe*2,[0 0 1 1 0 0]);
x=chirp([0:1/fe:5],fdeb,5,ffin);
f=linspace(fdeb,ffin,length(x));
plot(f,filter(b,1,x));hold on
% excitation par un bruit blanc et autocorr
x=randn(1,length(f)*2);
y=filter(b,1,x);
rxy=xcorr(x,y);
Ryx=abs(fft(rxy));
Ryx=Ryx(1:length(f));
plot(linspace(0,ffin,length(f)),Ryx/max(Ryx),'g')
% balayage explicite de frequence
freq=[fdeb:150:ffin];
k=1;
for f=freq
  x=sin(2*pi*f*[0:1/fe:1]);
  y=filter(b,1,x);
  sortie(k)=max(y);
  k=k+1;
end
hold on;plot(freq,sortie,'r')
xlabel('frequence (Hz)')
ylabel('magnitude (u.a.)')
```

En haut à gauche : mesure expérimentale de la réponse d'un FIR passe bande conçu pour être passant entre 700 et 900 Hz. La mesure est effectuée au moyen d'un chirp de 40 à 4000 Hz en 30 secondes généré par une carte son d'ordinateur. En bas à gauche : modélisation de la réponse d'un filtre passe bande entre 700 et 900 Hz, en bleu par filtrage d'un chirp, en rouge par calcul de l'effet du filtre sur quelques signaux monochromatiques. Ci-dessus : programme GNU/Octave de la simulation.

Exemple de programme et résultats :

```
// #define pc
//#define debug

#ifndef pc
//#include <stm32f1/stm32f10x.h>      // definition uint32_t
#include <libopencm3/cm3/common.h> // BEGIN_DECL

#include <libopencm3/stm32/f1/rcc.h>
#include <libopencm3/stm32/f1/flash.h>
#include <libopencm3/stm32/f1/gpio.h>
#include <libopencm3/stm32/f1/adc.h>
#include <libopencm3/stm32/usart.h>
#include <libopencm3/stm32/timer.h>
#include <libopencm3/cm3/nvic.h>
#else
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#endif

#define NB 128
#define NBFIL 61

unsigned short temperature[NB] ;
volatile int jmf_index=0;

long filtre(unsigned short *,long *,long ,long ,long *);
```

```
#ifndef pc
static void usart_setup(void)
{
        rcc_peripheral_enable_clock(&RCC_APB2ENR, RCC_APB2ENR_USART1EN);

        gpio_set_mode(GPIOA, GPIO_MODE_OUTPUT_50_MHZ,
                GPIO_CNF_OUTPUT_ALTFN_PUSHPULL, GPIO_USART1_TX);
        usart_set_baudrate(USART1, 115200);
        usart_set_databits(USART1, 8);
        usart_set_stopbits(USART1, USART_STOPBITS_1);
        usart_set_mode(USART1, USART_MODE_TX_RX);
        usart_set_parity(USART1, USART_PARITY_NONE);
        usart_set_flow_control(USART1, USART_FLOWCONTROL_NONE);
        usart_enable(USART1);
}

static void gpio_setup(void)
{ rcc_peripheral_enable_clock(&RCC_APB2ENR, RCC_APB2ENR_IOPCEN);
  gpio_set_mode(GPIOA, GPIO_MODE_INPUT, GPIO_CNF_INPUT_ANALOG, GPIO0);
  gpio_set_mode(GPIOA, GPIO_MODE_INPUT, GPIO_CNF_INPUT_ANALOG, GPIO1);
  gpio_set_mode(GPIOC, GPIO_MODE_OUTPUT_2_MHZ, GPIO_CNF_OUTPUT_PUSHPULL, GPIO8|GPIO9);
}

static void timer_setup(void)
{ uint32_t timer;
  volatile uint32_t *rcc_apbenr;
  uint32_t rcc_apb;

  timer = TIM2;
  rcc_apbenr = &RCC_APB1ENR;
  rcc_apb = RCC_APB1ENR_TIM2EN;

  rcc_peripheral_enable_clock(rcc_apbenr, rcc_apb);
  timer_reset(timer);
  timer_set_mode(timer, TIM_CR1_CKD_CK_INT, TIM_CR1_CMS_EDGE, TIM_CR1_DIR_UP);
  timer_set_period(timer, 0xF*5);
  timer_set_prescaler(timer, 0x8);
  timer_set_clock_division(timer, 0x0);
  timer_set_master_mode(timer, TIM_CR2_MMS_UPDATE); // Generate TRGO on every update
  timer_enable_counter(timer);
}

static void irq_setup(void)
{ nvic_set_priority(NVIC_ADC1_2_IRQ, 0);
  nvic_enable_irq(NVIC_ADC1_2_IRQ);
}

static void adc_setup(void)
{ int i;
  rcc_peripheral_enable_clock(&RCC_APB2ENR, RCC_APB2ENR_ADC1EN);
  adc_off(ADC1);
  adc_enable_scan_mode(ADC1);
  adc_set_single_conversion_mode(ADC1);
  adc_enable_external_trigger_injected(ADC1,ADC_CR2_JEXTSEL_TIM2_TRGO); // start with TIM2 TRGO
  adc_enable_eoc_interrupt_injected(ADC1);                              // Generate the ADC1_2_IRQ
  adc_set_right_aligned(ADC1);
//adc_enable_temperature_sensor(ADC1);
  adc_set_sample_time_on_all_channels(ADC1, ADC_SMPR_SMP_28DOT5CYC);
  adc_power_on(ADC1);

  for (i = 0; i < 800000; i++) __asm__("nop");
  adc_reset_calibration(ADC1);
  while ((ADC_CR2(ADC1) & ADC_CR2_RSTCAL) != 0);
  adc_calibration(ADC1);
  while ((ADC_CR2(ADC1) & ADC_CR2_CAL) != 0);
}

void adc1_2_isr(void)
{ ADC_SR(ADC1) &= ~ADC_SR_JEOC;
  if (jmf_index<NB)
```

```
99      {temperature[jmf_index]=adc_read_injected(ADC1,1);jmf_index++;}
}
101 #else
  #define fe 32768.
103   #define USART1 0
   volatile float temps=0.;
105   volatile float freq=fe/(float)NB; // garantit 1 periode

107   void adc1_2_isr(void)
     {temperature[jmf_index]=(int)(1200.+1000.*cos(2*M_PI*freq*temps));jmf_index++;
109      temps+=(1/fe);
     }
111 #endif

113 static void my_usart_print_int(unsigned int usart, int value)
{
115 #ifndef pc
   int8_t i;
117   uint8_t nr_digits = 0;
   char buffer[25];
119   if (value < 0) {usart_send_blocking(usart, '-');value=-value;}
   while (value > 0) {buffer[nr_digits++] = "0123456789"[value % 10];value /= 10;}
121   for (i = (nr_digits - 1); i >= 0; i--) {usart_send_blocking(usart, buffer[i]);}
   usart_send_blocking(usart, '\r');usart_send_blocking(usart, '\n');
123 #else
   printf("%d ",value);
125 #endif
}
127

long filtre(unsigned short *in,long *fil,long nin,long nfil,long *sortie)
129 {unsigned short k,j;long s=0,max=-2500;
 if (nin>nfil)
131   {for (k=0;k<nin-nfil;k++)
     {s=0;
133      for (j=0;j<nfil;j++)
          s+=((((long)fil[j]*(long)in[nfil-1+k-j])))/256;
135      s=s/256;
      sortie[k]=s;
137      if (max<s) {max=s;}
     }
139   }
   return((long)max);
141 }

143 int main (void)
{
145  unsigned short k=0;
 long sortie[NB],max;
147  unsigned char channel_array[16];

149 // a=(round(firls(60,[0 500 700 900 1000 32768/16]/32768*16,[0 0 1 1 0 0])*65536)')
/*long fil200[NBFIL]={-284,-343,193,700,362,-457,-665,-167,108,8,390,1092,\
151      557,-1361,-2181,-294,2040,1797,-110,-628,-77,-1315,-3220,-821,\
     5539,7177,-1131,-10626,-8081,5196,12938,5196,-8081,-10626,-1131,7177,\
153     5539,-821,-3220,-1315,-77,-628,-110,1797,2040,-294,-2181,-1361,\
     557,1092,390,8,108,-167,-665,-457,362,700,193,-343,-284};
155 */

157 // a=(round(firls(60,[0 500 700 900 1000 32768/2]/32768*2,[0 0 1 1 0 0])*65536)')
long fil200[NBFIL]={-384,-543,-695,-836,-964,-1074,-1164,-1229,-1270,-1283, \
159  -1268,-1224,-1153,-1054,-930,-783,-616,-433,-237,-34,173,379,578,767,941,\
  1096,1228,1334,1411,1459,1475,1459,1411,1334,1228,1096,941,767,578,379,173,\
161  -34,-237,-433,-616,-783,-930,-1054,-1153,-1224,-1268,-1283,-1270,-1229,-1164,\
  -1074,-964,-836,-695,-543,-384};
163

#ifndef pc
165   rcc_clock_setup_in_hse_8mhz_out_24mhz();
   gpio_setup();
167   usart_setup();
   timer_setup();
```

```
169   irq_setup();
      adc_setup();
171
      gpio_set(GPIOC, GPIO8|GPIO9);
173   channel_array[0] = 1; // channel number for conversion
      adc_set_injected_sequence(ADC1, 1, channel_array);
175 #endif
      while (1) {
177 #ifdef pc
        adc1_2_isr();
179 #endif
        if (jmf_index>=NB)
181       {
    #ifdef debug
183       for (k=0;k<NB;k++) my_usart_print_int(USART1, temperature[k]);
    #endif
185       max=filtre(temperature,fil200,NB,NBFIL,sortie);
    #ifdef debug
187       for (k=0;k<NB-NBFIL;k++) my_usart_print_int(USART1, sortie[k]);
    #endif
189 #ifndef pc
          my_usart_print_int(USART1, max);
191 #else
          printf("\t%f\t%d\n", freq,max);
193       temps=0.;
          freq+=10.;
195       if (freq>=(fe/10.)) return(0);
    #endif
197       jmf_index=0;
        }
199 #ifndef pc
        gpio_toggle(GPIOC, GPIO8);
201 #endif
      }
203   return 0;
    }
```

On notera en particulier que ce programme est conçu soit pour s'exécuter sur PC (cas `#define pc`) soit sur STM32 (cas `#undef pc`) : la seule différence tient dans les fonctions d'initialisation et de communication, tandis que la partie arithmétique est commune aux deux implémentations. Il est en effet bien plus aisé de valider l'algorithme avec des valeurs synthétiques (appel explicite de la fonction de gestion d'interruption dans le `main` dans le cas de l'exécution sur PC – fonction qui renvoie dans ce cas des données idéales de sinisoïde à fréquence variable) que de déverminer le programme directement sur microcontrôleur.

14 Conception d'un filtre de réponse infinie (IIR)

Un IIR tient compte non seulement des dernières observations mais aussi des valeurs passées du filtre :

$$y_n = \sum_{k=0..m} b_k x_{n-k} - \sum_{k=1..m} a_k y_{n-k}$$

GNU/Octave propose les fonctions `cheb` et `butter` pour concevoir des filtres IIR. Une fois les coefficients b et a (cette fois a est un vecteur) identifiés, nous appliquons la relation précédente pour identifier la nouvelle valeur de la sortie filtrée. Un IIR présente moins de coefficients qu'un FIR pour une fonction de transfert donnée et par conséquent induit une lattence plus faible. Cependant, il peut être instable dans son implémentation numérique.

Exercice :
Exploiter ces fonctions pour concevoir un filtre aux performances comparables aux IIR proposés ci-dessus, et implémenter ces filtres dans le STM32 pour en valider le fonctionnement.

A Initialisation des périphériques

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

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

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

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


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


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

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

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

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

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

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

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

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

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

Listing 2 – Bibliothèque `libopencm3`

Références

[1] *RM0008 – Reference manual, STM32F101xx, STM32F102xx, STM32F103xx, STM32F105xx and STM32F107xx advanced ARM-based 32-bit MCUs*, (May 2011), Doc ID 13902 Rev 13, disponible à `http://www.st.com/web/en/resource/technical/document/reference_manual/CD00171190.pdf`

[2] G. Brown, *Discovering the STM32 Microcontroller*, version du 5 Juin 2016, disponible à `www.cs.indiana.edu/~geobrown/book.pdf`

[3] G. Goavec-Mérou, J.-M Friedt, *Le microcontroleur STM32 : un cœur ARM Cortex-M3*, GNU/Linux Magazine France n.148 (Avril 2012), disponible à `jmfriedt.free.fr`

[4] J.- M Friedt, G. Goavec-Mérou, *Traitement du signal sur système embarqué – application au RADAR à onde continue*, GNU/-Linux Magazine France n.149 (Mai 2012), disponible à `jmfriedt.free.fr`

[5] A.V. Oppenheim & R.W. Schafer, *Discrete-Time Signal Processing*, Prentice Hall (1989)

[6] W.H. Press, B.P. Flannery, S.A. Teukolsky, & W.T. Vertterling, *Numerical recipies in Pascal*, Cambridge University Press (1994), ou sur le web `http://en.wikipedia.org/wiki/Cooley%E2%80%93Tukey_FFT_algorithm`