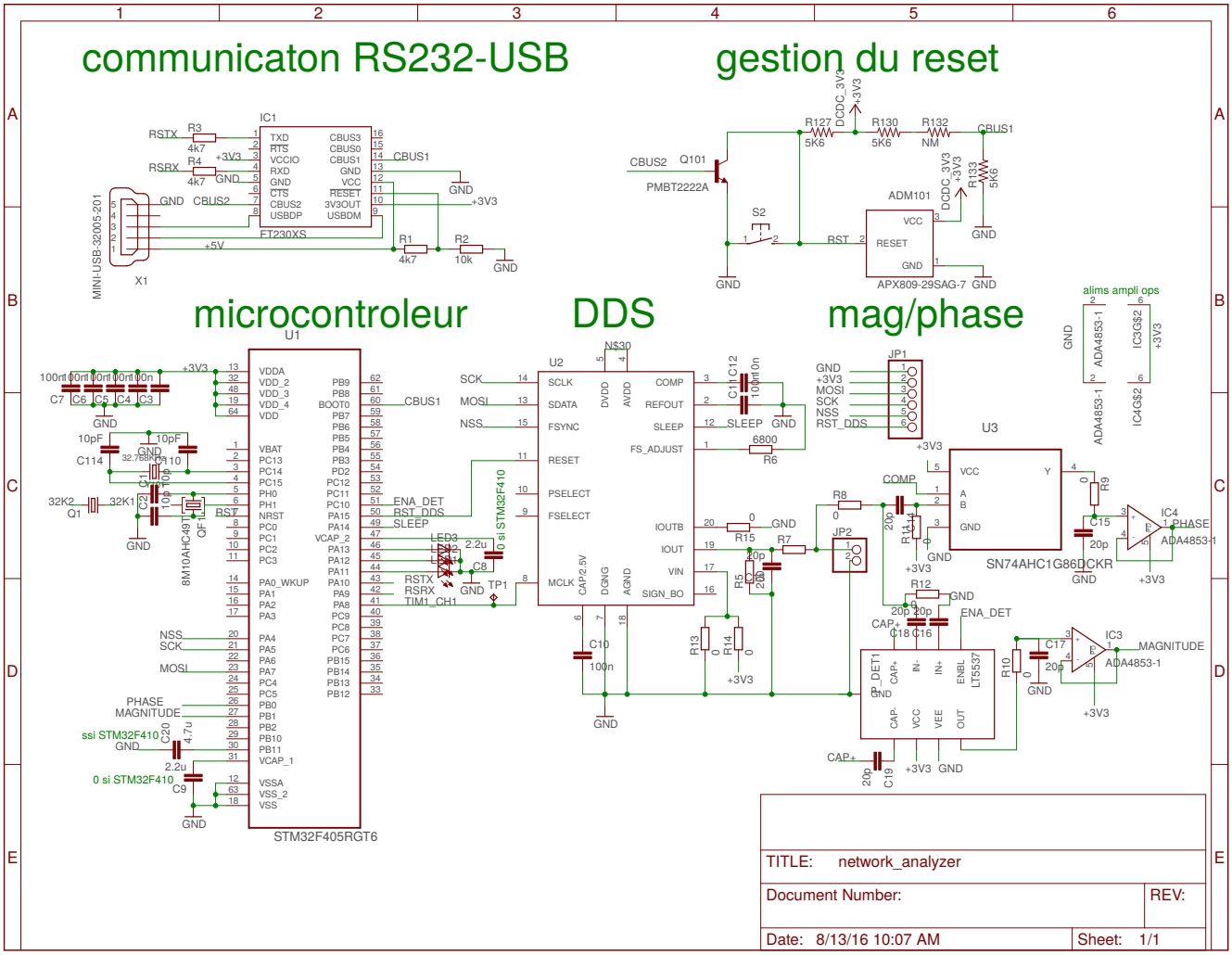
# TP M1

D. Rabus, J.-M Friedt

17 novembre 2016

La carte de développement est munie d'un microcontrôleur STM32F410. Ce processeur, complexe, est décrit en détail dans le manuel d'utilisation RM0401.

**Recherche sur internet les deux documents descriptifs du microcontrôleur que sont le manuel d'utilisation et la documentation du matériel.**




**Identifier les LEDs et les broches auxquelles elles sont connectées.**
**Identifier le signal qui cadencera le DDS (MCLOCK).**

# 1 Achitecture du développement logiciel

L'objectif de ces travaux pratiques est d'accumuler des briques de base permettant, à terme, de programmer la carte de

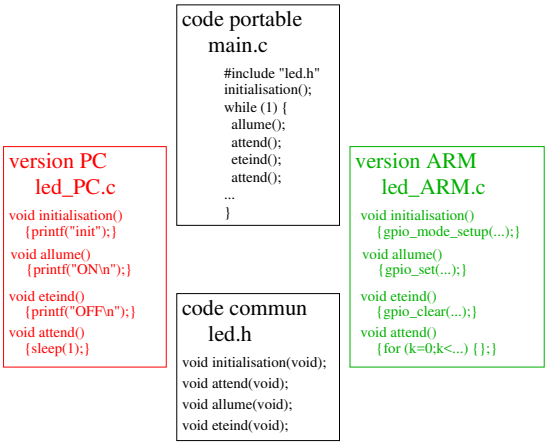



FIGURE 1:

TP pour effectuer un balayage de fréquences en vue de caractériser la réponse spectrale d'un microsystème. Par ailleurs, nous désirons architecturer le développement logiciel afin que la partie algorithmique du code soit portable et puisse être exécutée sur tout environnement de travail, y compris un PC. Pour ce faire, des "fausses" fonction d'accès au matériel permettront de compiler le programmer et d'indiquer à l'utilisateur qu'une ressource matérielle est sollicitée. Ainsi, nous proposons un développement du logiciel exploitant un programme principal contenant exclusivement du code portable (ne faisant pas appel aux spécificités matérielles de la plateforme cible), et des bibliothèques spécifiques à chaque plateforme pour émuler ou implémenter l'accès au matériel.
**Proposer un programme qui tournera sur PC, architecturé autour de deux fichiers sources (.c) et un fichier d'entête (.h), avec le programme principal ne contenant que des instructions indépendantes du matériel, et une bibliothèque émulant sur PC l'accès aux GPIOs du microcontrôleur et indiquant à l'utilisateur la polarité de la sortie. Compiler ce programme sur PC au moyen de `gcc` et démontrer son exécution. On prendra soin, lors de la compilation séparée des objets, de compiler les deux codes sources en objets (.c vers .o) puis de lier ces objets pour générer un exécutable.**

## 2 Prise en main de l'environnement de travail : les GPIO

Nous appuyerons nos développements sur une bibliothèque libre pour exploiter les cœurs ARM Cortex M3 et M4 nommée `libopencm3`, disponible à `git://github.com/libopencm3/libopencm3.git`. Une multitude d'exemples est par ailleurs proposée à `git://github.com/libopencm3/libopencm3-examples.git`.

```
1  #include <libopencm3/stm32/rcc.h>
2  #include <libopencm3/stm32/gpio.h>
3
4  static void clock_setup(void);
5  static void init_gpio(void);
6  void led_set(void);
7  void led_clr(void);
8  void attend(void);
9
10 static void clock_setup(void) // suppose quartz 25 MHz pour donner 120 MHz
11 {rcc_clock_setup_hse_3v3(&rcc_hse_25mhz_3v3[RCC_CLOCK_3V3_120MHZ]); // ANCIENNE_VERSION
12  rcc_periph_clock_enable(RCC_GPIOA); // Enable GPIOA clock.
13 }
14
15 static void init_gpio(void)
16 {gpio_mode_setup(GPIOA, GPIO_MODE_OUTPUT,GPIO_PUPD_NONE,GPIO11|GPIO12|GPIO13);
17 }
18
19 void led_set     (){gpio_set  (GPIOA, GPIO11|GPIO12|GPIO13);}
20 void led_clr     (){gpio_clear(GPIOA, GPIO11|GPIO12|GPIO13);}
21
22 void attend()
23 {volatile unsigned long i;
24  for (i=0;i<0x4fffff;i++) __asm("nop");
25 }
26
27 int main(void)
28 {clock_setup();
29  init_gpio();
30  while (1) {led_set();attend();led_clr();attend();}
31  return 0;
32 }
```

Le programme se compile au moyen de la déclinaison de `gcc` pour processeur ARM n'exécutant pas de système d'exploitation, nommé `arm-none-eabi-gcc`. Nous nous facilitons la vie en générant les commandes de compilation par un Makefile :

```
1 PROJ_NAME=test
2
3 CC=arm-none-eabi-gcc
4 OBJCOPY=arm-none-eabi-objcopy
```

```
5  CFLAGS = -Wall
6  CFLAGS += -mthumb -mcpu=cortex-m4 -mfloat-abi=hard -mfpu=fpv4-sp-d16 -DSTM32F4
7  CFLAGS += --static -nostartfiles -T./stm32f4-discovery.ld
8  LDFLAGS = -lopencm3_stm32f4
9
10 all: $(PROJ_NAME).elf
11
12 $(PROJ_NAME).elf: led_arm.o main.o
13    $(CC) led_arm.o main.o -o $(PROJ_NAME).elf $(LDFLAGS)
14    $(OBJCOPY) -O ihex $(PROJ_NAME).elf $(PROJ_NAME).hex
15    $(OBJCOPY) -O binary $(PROJ_NAME).elf $(PROJ_NAME).bin
16
17 main.o: main.c
18    $(CC) $(CFLAGS) -c main.c
19
20 led_arm.o: led_arm.c
21    $(CC) $(CFLAGS) -c led_arm.c
22
23 clean:
24    rm -f *.o $(PROJ_NAME) $(PROJ_NAME).elf $(PROJ_NAME).hex $(PROJ_NAME).bin
25
26 flash: $(PROJ_NAME).elf
27    stm32flash.sh -w $(PROJ_NAME).bin /dev/ttyUSB0
```

La règle `all` indique l'objectif à atteindre, à savoir compiler le projet nommé par la variable `PROJ_NAME`. Cet objectif dépend des fichiers sources définis par la variable `SRCS`, et les dépendances sont résolues par l'appel au compilateur dont le nom est fourni dans la variable `CC`. En fin de compilation, l'exécutable (au format ELF) est converti aux formats binaires (image de la mémoire flash) et hexadécimal (format ASCII) pour transfert au microcontrôleur. Une instruction `flash` est fournie pour lancer la séquence de programmation par `make flash`.

```
$ sudo make flash
stm32flash.sh -w test.bin /dev/ttyUSB0
/dev/ttyUSB0
0 reset 2 0 dload 1 0 dev /dev/ttyUSB0
0 reset 4 0 dload 2 0 dev /dev/ttyUSB0
stm32flash 0.5

http://stm32flash.sourceforge.net/

Using Parser : Raw BINARY
Interface serial_posix: 57600 8E1
Version      : 0x31
Option 1     : 0x00
Option 2     : 0x00
Device ID    : 0x0458 (STM32F410xx)
- RAM        : 32KiB  (12288b reserved by bootloader)
- Flash      : 128KiB (size first sector: 1x16384)
- Option RAM : 16b
- System RAM : 30KiB
Write to memory
Erasing memory
Wrote address 0x08000a44 (100.00%) Done.

1 reset 2 0 dload 1 0 dev /dev/ttyUSB0
1 reset 4 0 dload 2 0 dev /dev/ttyUSB0
```

Parfois des caractères restent dans le tampon de communication et le protocole d'acquittement du transfert de programme échoue, résultant dans un message d'erreur du type

```
$ sudo make flash
stm32flash.sh -w test.bin /dev/ttyUSB0
/dev/ttyUSB0
0 reset 2 0 dload 1 0 dev /dev/ttyUSB0
0 reset 4 0 dload 2 0 dev /dev/ttyUSB0
stm32flash 0.5

http://stm32flash.sourceforge.net/

Using Parser : Raw BINARY
Interface serial_posix: 57600 8E1
Failed to init device.
```

```
1 reset 2 0 dload 1 0 dev /dev/ttyUSB0
1 reset 4 0 dload 2 0 dev /dev/ttyUSB0
```

Dans ce cas, on relance la ligne de commande de programmation jusqu'à ce que le transfert du programme s'achève avec succès. On vérifiera éventuellement que le cable est convenablement connecté en consultant les messages du système par `dmesg` qui doit se conclure par :

```
[902653.402575] ftdi_sio 3-1:1.0: FTDI USB Serial Device converter detected
[902653.402627] usb 3-1: Detected FT-X
[902653.402891] usb 3-1: FTDI USB Serial Device converter now attached to ttyUSB0
```

Dans cet exemple, nous avons "mal" initialisé la boucle à vérouillage de phase (PLL) qui convertit l'oscillation du quartz externe en horloge interne au microcontrôleur. En effet, la fonction
`rcc_clock_setup_hse_3v3(&rcc_hse_25mhz_3v3[RCC_CLOCK_3V3_120MHZ]);`
suppose que nous alimentons le microcontrôleur avec un quartz à 25 MHz et que nous générons par multiplication de la fréquence du 120 MHz. Cependant, nos cartes sont cadencées par un quarts de 20 MHz : le fréquence de sortie de la PLL est donc $\frac{20}{25} \times 120 = 96$ MHz. Cet écart n'est pas important pour l'exemple que nous venons de proposer, mais va devenir significatif dans la section suivante qui nécessite une fréquence de cadencement exacte.
**Modifier la séquence de compilation pour faire appel à la fonction d'initialisation de l'horloge** `core_clock_setup()` **disponible dans le fichier** `http://jmfriedt.free.fr/stm32_initialisation.c`**. Le prototype de cette fonction est** `void core_clock_setup(void);`. Le principe de fonctionnement de cette fonction est décrit en annexe A.

# 3 Communication avec l'utilisateur : l'UART

```
1  #include <libopencm3/stm32/rcc.h>
2  #include <libopencm3/stm32/gpio.h>
3  #include <libopencm3/stm32/usart.h>          // fonctions UART
4  #include <libopencm3/stm32/f4/memorymap.h>
5
6  char buf[20]="Hello␣world\r\n\0";
7
8  static void jmf_putchar (unsigned char ch)   // write character to Serial Port
9  {usart_send_blocking (USART1, ch);}
10
11 static void delay(unsigned int delay)
12 {volatile unsigned int i;
13  for (i=0; i<delay; i++) __asm__("nop"); // wait
14 }
15
16 static void gpio_setup(void)
17 {[...]
18 }
19
20 static void init_usart (void)
21 {rcc_periph_clock_enable(RCC_USART1);
22  gpio_mode_setup (GPIOA, GPIO_MODE_AF, GPIO_PUPD_NONE, GPIO9);  //GPA9 : Tx send from STM32 to ext
23  gpio_mode_setup (GPIOA, GPIO_MODE_AF, GPIO_PUPD_NONE, GPIO10); //GPD10: Rx recieve from ext to STM32
24 // gpio_set_output_options (GPIOA, GPIO_OTYPE_OD, GPIO_OSPEED_25MHZ, GPIO10);
25  gpio_set_af (GPIOA, GPIO_AF7, GPIO9);
26  gpio_set_af (GPIOA, GPIO_AF7, GPIO10);
27   // Setup USART1 parameters.
28  usart_set_baudrate (USART1, (9600));
29  usart_set_databits (USART1, 8);
30  usart_set_stopbits (USART1, USART_STOPBITS_1);
31  usart_set_mode (USART1, USART_MODE_TX_RX);
32  usart_set_parity (USART1, USART_PARITY_NONE);
33  usart_set_flow_control (USART1, USART_FLOWCONTROL_NONE);
34  usart_enable (USART1);              // Finally enable the USART.
35 }
36
37 int main(void)
38 {core_clock_setup();
39  clock_setup();
40  gpio_setup();
41  init_usart();
```

```
 while (1) {
   delay(15000);
   mon_puts(buf);
 }
 return 0;
}
```

1. **Proposer la fonction `mon_puts()` qui prend en argument une chaîne de caractères et l'affiche sur le port série en faisant appel à `jmf_putchar`. On se rappellera de la convention du C qui définit la fin de chaîne de caractères par la valeur 0.**
2. **Proposer une fonction qui affiche en hexadécimal le contenu d'une variable de type short.**
3. **Proposer une fonction qui affiche en décimale le contenu d'une variable de type short.**

# 4 Maîtriser le temps : les *timers*

## 4.1 Mode *polling*

```
#include <libopencm3/stm32/rcc.h>
#include <libopencm3/stm32/gpio.h>
#include <libopencm3/cm3/nvic.h>
#include <libopencm3/stm32/timer.h>
#include <libopencm3/stm32/f4/memorymap.h>

static void clock_setup(void);
static void init_gpio(void);
void init_time1 (void);

static void clock_setup(void)
{rcc_periph_clock_enable (RCC_TIM1);
 rcc_periph_clock_enable(RCC_GPIOA); // Enable GPIOA clock.
}

static void init_gpio(void)
{gpio_mode_setup(GPIOA, GPIO_MODE_OUTPUT,GPIO_PUPD_NONE, GPIO13);}

void init_time1 (void)
{timer_reset (TIM1);
 timer_set_mode (TIM1, TIM_CR1_CKD_CK_INT, TIM_CR1_CMS_EDGE, TIM_CR1_DIR_UP);
//TIM_CR1_CKD_CK_INT = clock division ratio TIM_CR1_CMS_EDGE=counting mode TIM_CR1_DIR_UP=count direction
  timer_set_prescaler (TIM1, 4096);   // 140 MHz
  timer_set_period (TIM1, 17090);
  timer_enable_counter (TIM1);
}

int main(void)
{core_clock_setup();
 clock_setup();
 init_gpio();
 init_time1();
 while (1) {
   if (timer_get_flag(TIM1, TIM_EGR_UG)!=0)
      {timer_clear_flag(TIM1, TIM_EGR_UG);gpio_toggle(GPIOA,GPIO13);}
 }
 return 0;
}
```

1. **Calculer la fréquence de clignottement de la LED, sachant que le cœur du processeur est cadencé à 140 MHz.**
2. **Modifier le programme pour faire cligoter la LED du milieu.**
3. **Pourquoi ne pouvons nous pas utiliser un *prescaler* de 1024 pour cadencer la LED à 0,5 Hz?**
4. **Modifier le programme pour afficher sur port RS232 la valeur d'un compteur qui s'incrémente chaque seconde.**

## 4.2 Mode automatique

Le *timer* permet d'agir sur des broches sans intervention logicielle explicite. Cette fonction, nommée *output compare*, permet de commuter l'état d'une broche lorsqu'une condition sur le compteur est atteinte. La broche reprend son état initial lorsque la période du compteur est atteinte et le décompte recommence à 0.

```
#include <libopencm3/stm32/rcc.h>
#include <libopencm3/stm32/gpio.h>
#include <libopencm3/stm32/timer.h>
#include <libopencm3/stm32/f4/memorymap.h>

static void clock_setup(void);
void init_time1 (void);

static void clock_setup(void)
{rcc_periph_clock_enable (RCC_TIM1);
 rcc_periph_clock_enable(RCC_GPIOA); // Enable GPIOA clock.
}

void init_time1 (void)
{gpio_set_output_options(GPIOA, GPIO_OTYPE_PP, GPIO_OSPEED_50MHZ, GPIO11 );
 gpio_mode_setup (GPIOA, GPIO_MODE_AF, GPIO_PUPD_NONE, GPIO11);
 gpio_set_af (GPIOA, GPIO_AF1, GPIO11);
 timer_reset (TIM1);
 timer_set_mode (TIM1, TIM_CR1_CKD_CK_INT, TIM_CR1_CMS_EDGE, TIM_CR1_DIR_UP);
//TIM_CR1_CKD_CK_INT = clock division ratio TIM_CR1_CMS_EDGE=counting mode TIM_CR1_DIR_UP=count direction

// https://github.com/libopencm3/libopencm3/blob/master/lib/stm32/common/timer_common_all.c
  timer_set_oc_mode(TIM1, TIM_OC4, TIM_OCM_PWM2);
  timer_enable_oc_output(TIM1, TIM_OC4);
  timer_enable_break_main_output(TIM1);
  timer_set_oc_value(TIM1, TIM_OC4, 8000);

  timer_set_prescaler (TIM1, 4096);   // 140 MHz
  timer_set_period (TIM1, 16000); //
  timer_enable_counter (TIM1);
}

int main(void)
{core_clock_setup();
 clock_setup();
 init_time1();
 while (1) {}
 return 0;
}
```

1. **Pourquoi avons nous utilisé `OC4` ?**
2. **Modifier ce programme pour générer un signal à 70 MHz sur MCLOCK du DDS. On s'inspirera pour ce faire de la figure 2.**

## 4.3 Mode interruption

Finalement, le *timer* peut déclencher périodiquement une fonction complexe appelée par interruption. C'est ainsi par exemple qu'est cadencé l'ordonnanceur d'un système d'exploitation. Une interruption doit absolument être associée à un gestionnaire d'interruption lors de son activation : la liste des vecteurs d'interruption – adresse à laquelle saute l'unité arithmétique et logique lorsqu'une source d'interruption se déclenche, est proposée à `arm-none-eabi/include/libopencm3/stm32/f4/nvic.h`.

On prendra soin de toujours minimiser le temps d'exécution d'un gestionnaire d'interruption afin de ne pas risquer d'avoir à gérer une seconde interruption alors que la première n'a pas fini d'être acquittée. Dans l'exemple ci-dessous, le programme principal est informé du déclenchement de l'interruption et gère la partie la moins urgente de l"évèmenenbt lorsqu'il en a le temps. Exceptionnellement, nous nous autorisons une variable globale pour effectuer cet échange.

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

**Figure 12. Clock tree**

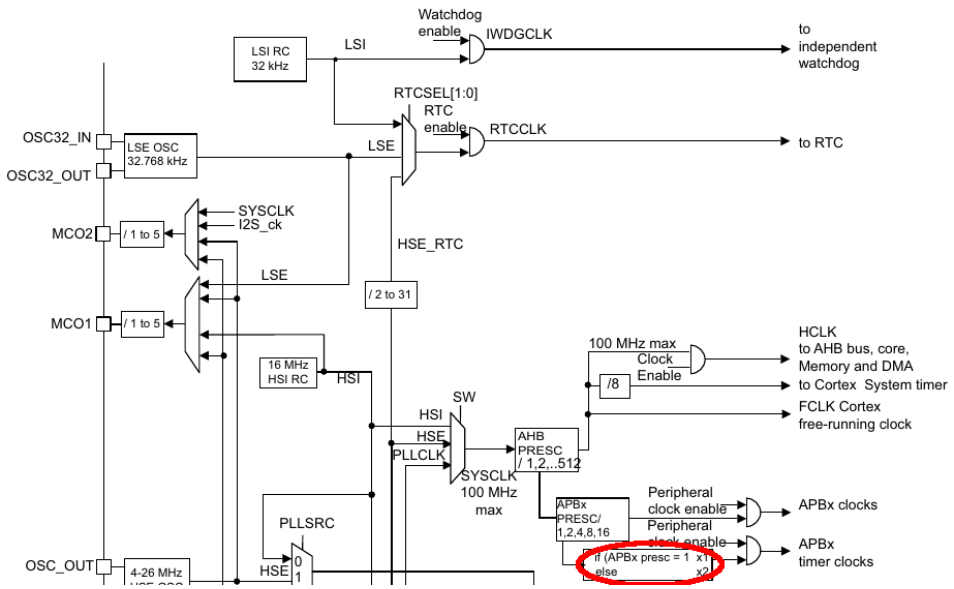



FIGURE 2 – Extrait de la page 93 du manuel RM0401.

```
4  #include <libopencm3/stm32/timer.h>
5  #include <libopencm3/stm32/f4/memorymap.h>
6
7  static void clock_setup(void);
8  static void init_gpio(void);
9  void init_time1 (void);
10
11 volatile int flag=0;
12
13 static void clock_setup(void)
14 {rcc_periph_clock_enable (RCC_TIM1);
15  rcc_periph_clock_enable(RCC_GPIOA); // Enable GPIOA clock.
16 }
17
18 static void init_gpio(void)
19 {gpio_mode_setup(GPIOA, GPIO_MODE_OUTPUT,GPIO_PUPD_NONE, GPIO12);
20 }
21
22 void init_time1 (void)
23 {nvic_enable_irq (NVIC_TIM1_CC_IRQ);    //uart interrupt
24  nvic_set_priority(NVIC_TIM1_CC_IRQ,1);
25  timer_reset (TIM1);
26  timer_set_mode (TIM1, TIM_CR1_CKD_CK_INT, TIM_CR1_CMS_EDGE, TIM_CR1_DIR_UP);
27 //TIM_CR1_CKD_CK_INT = clock division ratio TIM_CR1_CMS_EDGE=counting mode TIM_CR1_DIR_UP=count direction
28  timer_enable_irq(TIM1,TIM_DIER_CC1IE);
29
30  timer_set_prescaler (TIM1, 1024);   // 140 MHz
31  timer_set_period (TIM1, 16000); //
32  timer_enable_counter (TIM1);
33 }
34
35 void tim1_cc_isr(void)  // voir arm-none-eabi/include/libopencm3/stm32/f4/nvic.h la liste des interrupts
36 {timer_clear_flag(TIM1, TIM_SR_CC1IF); // acquitter l'interruption
37  flag=1;
38 }
39
40 int main(void)
41 {core_clock_setup();
42  clock_setup();
43  init_gpio();
```

```
 init_time1();
 while (1) {
   if (flag==1)
      {gpio_toggle(GPIOA,GPIO12);flag=0;}
 }
 return 0;
}
```

## 4.4 Exemple de la réception d'un caractère sur UART

Un cas classique d'utilisation d'interruptions est la transaction sur bus de communication. Un processeur cadencé à 140 MHz peut effectuer jusqu'à une instruction toutes les 7 ns, alors qu'une transaction sur port série (UART) à 115200 bauds nécessite 8,7 $\mu$s par bit ou 87 $\mu$s par octet. Il est donc inefficace de demander au processeur d'attendre qu'une transaction soit finie pour continuer à exécuter le programme : une interruption peut nous informer qu'une telle action est achevée. De même, une interruption peut se déclencher à la réception d'un caractère sur le port série : au lieu de perdre du temps à continuellement sonder le port de communication, ou pire de rater un caractère transmis car nous n'avons pas eu le temps de sonder l'état du port de communication entre deux transactions, nous déclenchons une interruption qui capture le caractère et le mémorise en vue de son traitement lorsque le programme principal en aura le temps.

```
#include <libopencm3/stm32/rcc.h>
#include <libopencm3/stm32/gpio.h>
#include <libopencm3/stm32/exti.h>
#include <libopencm3/cm3/nvic.h>
#include <libopencm3/stm32/usart.h>
#include <libopencm3/stm32/f4/memorymap.h>

static void clock_setup(void)
{[...]
}

static void init_usart (void)
{rcc_periph_clock_enable(RCC_USART1);
 nvic_enable_irq (NVIC_USART1_IRQ);    //uart interrupt
 nvic_set_priority(NVIC_USART1_IRQ,1);
 gpio_mode_setup (GPIOA, GPIO_MODE_AF, GPIO_PUPD_NONE, GPIO9);  //GPA9 : Tx send from STM32 to ext
[...]
 usart_set_flow_control (USART1, USART_FLOWCONTROL_NONE);
 usart_enable_rx_interrupt (USART1); // Enable USART1 Receive interrupt.
 usart_enable (USART1);              // Finally enable the USART.
}

void usart1_isr (void)
{static uint8_t data = 'A';
 // Check if we were called because of RXNE.
 if (((USART_CR1 (USART1) & USART_CR1_RXNEIE) != 0) &&
      ((USART_SR (USART1) & USART_SR_RXNE) != 0))
    {data = usart_recv (USART1);         // Retrieve the data from the peripheral.
     buf[0]=data;
     usart_enable_tx_interrupt (USART1); // Enable transmit interrupt so it sends back the data.
    }
  if (((USART_CR1 (USART1) & USART_CR1_TXEIE) != 0) && // Check if we were called because of TXE.
      ((USART_SR (USART1) & USART_SR_TXE) != 0))
    {usart_send (USART1, data);           // Put data into the transmit register.
     usart_disable_tx_interrupt (USART1); // Disable the TXE interrupt as we don't need it anymore.
    }
}

int main(void)
{core_clock_setup();
 clock_setup();
 init_usart();
 while (1) {
   delay(15000);
   jmfputs(buf);
 }
```

```
 return 0;
}
```

## 5 Communication synchrone : le bus SPI

De nombreux périphériques, dont le DDS, communiquent par bus synchrone. Les divers paramètres à régler sont l'état au repos de l'horloge, le front sur lequel les données sont échantillonnées, la vitesse d'horloge et l'ordre dans lequel un octet est transmis (bit de poids fort ou faible en premier).
**Identifier ces paramètres dans la datasheet du DDS AD9834.**

```
#include <libopencm3/stm32/rcc.h>
#include <libopencm3/stm32/gpio.h>
#include <libopencm3/stm32/spi.h>
#include <libopencm3/stm32/exti.h>
#include <libopencm3/cm3/nvic.h>
#include <libopencm3/stm32/usart.h>
#include <libopencm3/stm32/f4/memorymap.h>

static void delay(unsigned int );
static void clock_setup(void);
static void init_gpio(void);
static void init_usart (void);
void init_spi (void);
void init_time1 (void);
void dds_cs_set (void);
void dds_cs_clr (void);
void envoi_DDS (unsigned char *, int);

static void clock_setup(void)
{rcc_periph_clock_enable (RCC_SPI1);
 rcc_periph_clock_enable (RCC_TIM1);
}

static void init_gpio(void)
{rcc_periph_clock_enable(RCC_GPIOA); // Enable GPIOA clock.
 gpio_mode_setup(GPIOA, GPIO_MODE_OUTPUT,GPIO_PUPD_NONE, GPIO15|GPIO4); // Enable (DDS RST,CS)
}

static void init_usart (void)
{[...]
}

//initialisation spi et gpio pour DDS
void init_spi (void)
{gpio_mode_setup (GPIOA, GPIO_MODE_AF, GPIO_PUPD_PULLUP, GPIO5 | GPIO6 | GPIO7); // GPIO in output
 gpio_set_af (GPIOA, GPIO_AF5, GPIO5 | GPIO6 | GPIO7);   //GPIO in SPI mode GPIOA5=SCK 6=MISO 7=MOSI
 //MSB first is th default configuration for the DDS AD9959 obtained after a master reset.
 //le DDS prend les donnÃ¨es en falling edge.
 spi_disable (SPI1);    //turn off SPI for configuration
 spi_init_master (SPI1,
   SPI_CR1_BAUDRATE_FPCLK_DIV_256, SPI_CR1_CPOL_CLK_TO_1_WHEN_IDLE, // clk to 1 when idle
   SPI_CR1_CPHA_CLK_TRANSITION_2,   //falling EDGE: DDS needs falling edge
   SPI_CR1_DFF_8BIT, //8 bits
   SPI_CR1_MSBFIRST);   //MSB first
  spi_enable_software_slave_management (SPI1);
  spi_set_nss_high (SPI1);
  spi_enable (SPI1);
}

void dds_cs_set () {gpio_set (GPIOA, GPIO4);}
void dds_cs_clr () {gpio_clear(GPIOA, GPIO4);}

void envoi_DDS (unsigned char *entree, int n)
{ unsigned char i;
  dds_cs_set (); delay (100);
```

```
  dds_cs_clr (); delay (100);
  for (i = 0; i < n; i++) spi_send (SPI1, entree[i]);
//      while ((SPI_SR(SPI2) & 0x40)==0x00); //(!(SPI_SR(SPI2) & SPI_SR_TXE));
  delay (1000); // JMF : a revoir
  dds_cs_set ();
}

int main(void)
{unsigned char entree[3]={0x55,0x44,0xaa};
 core_clock_setup();
 clock_setup();
 init_gpio();
 init_usart();
 init_spi();
 while (1) {
   delay(15000);
   gpio_toggle(GPIOA,GPIO15); // DDS reset
   jmfputs(buf);
   envoi_DDS(entree,3);
 }
 return 0;
}
```

## 6 Mesurer une grandeur analogique : ADC

Le convertisseur analogique-numérique (ADC) de $N$ bits de résolution génère un mot proportionnel à la tension $V$ lue sur une de ses entrées, normalisée par rapport à une tension de référence $V_{ref}$

$$mot = \frac{V}{V_{ref}} \times \left(2^N - 1\right)$$

```
#include <libopencm3/stm32/rcc.h>
#include <libopencm3/stm32/gpio.h>
#include <libopencm3/stm32/adc.h>
#include <libopencm3/stm32/usart.h>
#include <libopencm3/stm32/f4/memorymap.h>

#define NBMES 512

static void delay(unsigned int );
static void clock_setup(void);
static void init_gpio(void);
static void init_usart (void);
void init_adc(void);
unsigned short read_adc (unsigned char);

void init_adc(void)
{gpio_mode_setup (GPIOB, GPIO_MODE_ANALOG, GPIO_PUPD_NONE, GPIO0|GPIO1);
 adc_power_off (ADC1);
 adc_disable_scan_mode (ADC1);
//attention : ADCCLOCK est limit'e a 30 MHz, issue de APB2 => facteur de div tq APB2/DIV<30 MHz
 adc_set_clk_prescale (ADC_CCR_ADCPRE_BY4);
 adc_set_sample_time_on_all_channels (ADC1, ADC_SMPR_SMP_112CYC);
//si ADCCLOCK = 21 MHz, sampletime = 144 => Tconv = 7.4286us.
 adc_power_on (ADC1);
}

unsigned short read_adc (unsigned char channel)
{uint8_t channel_array[16];
 channel_array[0] = channel;
 adc_set_regular_sequence (ADC1, 1, channel_array);    //config used channel
 adc_start_conversion_regular (ADC1);
 while (!adc_eoc (ADC1));
 return(adc_read_regular(ADC1));
```

```
}

int main(void)
{int k;
 clock_setup();
 init_gpio();
 init_usart();
 init_adc();

 while (1) {
      delay(1000);
      for (k=0;k<NBMES;k++)
        {v8[k]=read_adc(8); v9[k]=read_adc(9);}
      for (k=0;k<NBMES;k++)
        {jmf_puts("0x\0");jmf_put16(v8[k]);jmf_putchar ('␣');
         jmf_puts("0x\0");jmf_put16(v9[k]);jmf_puts("\r\n\0");
        }
     }
 }
 return 0;
}
```

1. **Générer un signal de fréquence connue sur la sortie du DDS et l'échantillonner sur l'ADC. Quelle est la cadence d'échantillonnage de l'ADC?.**

# 7 Caractérisation du dispositif radiofréquence

Balayage de fréquence, mesure de la puissance transmise.

# 8 Déclencher une interruption sur un évènement matériel

De nombreux GPIO peuvent être connectés à un gestionnaire d'interruption pour déclencher un signal lorsqu'un relais par exemple s'ouvre et se ferme. Cette fonctionnalité se nomme EXTI (*EXTernal Interrupt*). Dans notre cas nous proposons de déclencher une interruption sur PA4 qui fait office habituellement de CS$ du bus SPI. Ainsi, si le DDS est commandé par un composant externe (FPGA, CPU), nous pourrons déclencher une fonction (par exemple conversion analogique-numérique) sur le STM32 chaque fois qu'une programmation du SPI est achevée. L'utilisateur doit prendre soin de ne pas mélanger les fonctionnalités de EXTI : toutes les broches 0 de tous les ports sont connectées à EXTI0, donc une seule de ces broches peut servir de source d"interruption à un instant donné. De même, il n'existe qu'un nombre limité de vecteurs d'interruption (cf `sat/arm-none-eabi/include/libopencm3/stm32/f4/nvic.h`) : EXTI0, 1, 2, 3, 4, 15-10 et 9-5. Ces deux derniers ensembles d'interruptions sont donc communes à toutes les sources. Une fois ces restrictions identifiées, l'utilisation de EXTI est très simple :

```
// TOP: GND VCC SPI SPI CS RST
//           \--- R ----/

#include <libopencm3/stm32/rcc.h>
#include <libopencm3/stm32/gpio.h>
#include <libopencm3/stm32/exti.h>
#include <libopencm3/cm3/nvic.h>
#include <libopencm3/stm32/f4/memorymap.h>

void core_clock_setup(void);

static void gpio_setup(void)
{rcc_periph_clock_enable(RCC_GPIOA); // Enable GPIOA clock.
 gpio_mode_setup(GPIOA, GPIO_MODE_OUTPUT,GPIO_PUPD_NONE, GPIO11); // Enable LED pin
}

static void exti_setup(void)
{nvic_enable_irq(NVIC_EXTI4_IRQ);
 gpio_mode_setup(GPIOA, GPIO_MODE_INPUT,GPIO_PUPD_PULLDOWN, GPIO4);
 exti_select_source(EXTI4, GPIOA);           // GPIOn se connects a EXTIn
 exti_set_trigger(EXTI4, EXTI_TRIGGER_BOTH);
 exti_enable_request(EXTI4);
```

```
}

void exti4_isr(void)
{gpio_toggle(GPIOA, GPIO11);
 exti_reset_request(EXTI4);
}

int main(void)
{core_clock_setup();
 gpio_setup();
 exti_setup();
 while (1) {}
 return 0;
}
```

Ce programme se teste en connectant est déconnectant une résistance entre l'alimentation et CS# du SPI (PA4). Comme nous déclenchons sur les deux fronts, la LED s'allume et s'éteint selon que nous connections ou déconnections la résistance de Vcc.

# A Configuration des horloges

```
// 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, // 4 WS d'apres configuration par ST
                .ahb_frequency  = 140000000,
                .apb1_frequency = 35000000,
                .apb2_frequency = 70000000,
        };

/**
  * @file    system_stm32f4xx.c
  * @author  MCD Application Team
  *==============================================================================
  *        Supported STM32F40xx/41xx/427x/437x devices
  *        System Clock source                   | PLL (HSE)
  *        SYSCLK(Hz)                            | 140000000
  *        HCLK(Hz)                              | 140000000
  *        AHB Prescaler                         | 1
  *        APB1 Prescaler                        | 4
  *        APB2 Prescaler                        | 2
  *        HSE Frequency(Hz)                     | 20000000
  *        PLL_M                                 | 20
  *        PLL_N                                 | 280
  *        PLL_P                                 | 2
  *        PLL_Q                                 | 6
  *        Flash Latency(WS)                     | 4
  *        Require 48MHz for USB OTG FS,         | Disabled
  *        SDIO and RNG clock                    |
*/
```