Projet : analyseur de réseau radiofréquence à base de STM32 et de synthétiseur numérique

15 septembre 2019

1 Principes généraux

Notre objectif est de caractériser la réponse spectrale, en phase et en magnitude, de dispositifs radiofréquences fonctionnant en réflexion (au contraire des dispositifs fonctionnant en transmission). Nous allons donc nous efforcer de mesurer des coefficients de réflexion, S_{11}. Diverses architectures d'analyseur de réseaux existent, soit en utilisant une source de bruit et en analysant le signal reçu par transformée de Fourier, une approche gourmande en ressources de calcul, soit en balayant la fréquence d'excitation du dispositif et en mesurant les caractéristiques du signal transmis. La seconde approche est plus simple à mettre en œuvre, surtout pour des fréquences dépassant quelques centaines de kilohertz, et sera donc sélectionnée pour ce projet.

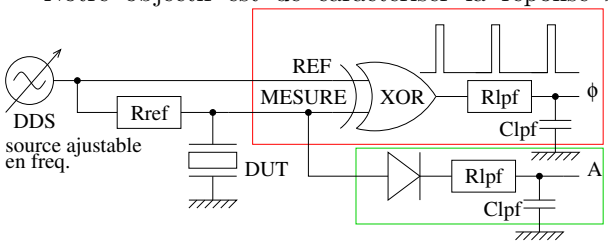


Figure 1: Principe de la mesure : en vert la mesure d'amplitude, en rouge la mesure de phase, du dispositif analysé (DUT).

Pour ce faire, nous avons besoin de trois éléments :

— une source de signaux radiofréquences, idéalement périodiques, ajustable en fréquence,
— un détecteur de puissance,
— un détecteur de phase.

Afin de synchroniser l'ensemble des mesures, un microcontrôleur sera chargé de contrôler ces périphériques, d'une part en configurant la source de fréquence, d'autre part en effectuant les mesures de puissance et de phase, et finalement en communiquant le résultat des mesures à l'utilisateur.

Diverses sources agiles de fréquence existent : oscillateur contrôlé en tension (VCO), boucle à vérouillage de phase (PLL et PLL fractionnaire). Cependant, la première solution souffre d'un manque de linéarité entre le signal de commande et la fréquence de sortie, caractéristique qui évolue avec les conditions environnementales (température notamment) du composant. La seconde solution n'offre qu'un pas de mesure grossier qui sera insuffisant pour caractériser des dispositifs de fort facteur de qualité. Nous sélectionnerons donc un composant numérique capable de générer un signal radiofréquence de fréquence définie finement : le synthétiseur numérique de fréquence (*Direct Digital Synthesizer*, DDS). Ce composant est formé d'un compteur (accumulateur), d'une table de conversion de l'accumulateur vers une sinusoïde, et d'un convertisseur numérique-analogique pour générer le signal exploitable. Un DDS cadencé à f_{CK} dont la fréquence est définie sur N bits définit sa fréquence de sortie f par la relation au mot de configuration (numérique) W (fourni sur N bits) par

$$f = \frac{f_{CK}}{2^N} \cdot W$$

Le pas de fréquence est donc directement fourni par $f_{CK}/2^N$. À titre d'exemple, un AD9954, pour lequel $N = 32$ bits, cadencé à 400 MHz propose une résolution de 100 mHz ! Dans notre cas, nous nous contenterons d'un AD9834 cadencé entre 50 et 100 MHz. Puisque $N = 28$, la résolution est de l'ordre de 200 à 400 mHz. Si une meilleure résolution est nécessaire pour des dispositifs basse fréquence – par exemple pour caractériser un diapason à quartz à 32 kHz – il suffira d'abaisser la fréquence d'horloge cadençant le DDS : par exemple en proposant $f_{CK} = 1$ MHz, la résolution en fréquence devient 4 mHz, largement suffisant pour caractériser un dispositif même si son facteur de qualité atteint 10^5

2 Choix des composants

Les fonctionnalités étant définies, il nous faut choisir les composants qui réaliseront les objectifs annoncés. Une contrainte de coût – nous nous imposons que la carte complète et assemblée coûte moins de 50 euros – en plus des fonctionnalités, et de disponibilité des composants en faibles volumes, détermine le choix des références. Nous nous imposons de travailler sur un microcontrôleur 32 bits supportant

potentiellement un environnement exécutif tel que FreeRTOS ou RTEMS : le STM32 de ST Microelectronics est un choix qui permet de profiter de l'expérience sur cette plateforme d'une des équipes du département temps-fréquence de FEMTO-ST. Par ailleurs, une bibliothèque libre, `libopencm3` [1] et ses exemples [2], simplifie la prise en main du microcontrôleur. Cette plateforme de travail est évidemment supportée par `gcc`, y compris l'unité de calcul en virgule flottante.

Le bus USB est un protocole pénible, gourmand en ressources, et malgré sa prolifération dans l'informatique grand publique, est peu approprié pour les applications industrielles. Afin de faciliter le développement sur ordinateur personnel, nous munissons notre circuit d'un convertisseur RS232-USB, FTDI FT230XS, qui par ailleurs se charge de réguler le signal 5 V fourni par USB et de générer une source stable de tension à 3,3 V.

Value	Description	Coût
FT230XS	farnell 2081321	2,12 /p
APX809-29SAG-7	farnell 1825363	0,263/p
LT5537	LT samples (2/pers.)	0,263/p
ADA4853-1	farnell 2312825	2,62 /p
ADA4853-1	farnell 2312825	2,62 /p
STM32F410RBT6	farnell 2518136	3,02/p
AD9834	farnell 2377023	10,62/p
SN74AHC1G86DCKR	farnell 1287455	0,117/p
PMBT2222A	farnell 1081475	0,107/p
MINI-USB-32005-201	farnell 2313554	0,918/p
32.768KHz	farnell 1457085	0,353/p
8M10AHC49T	farnell 2509289	0,374/p
AD8611	farnell 9604324	5,55/p
Somme		28.92

Figure 2: Liste des composants, excluant les passifs (condensateurs, résistances) dont le coût est inférieur à 1 euro. Malgré son faible coût, le détecteur de puissance de Linear Technology est obtenu en échantillons car indisponible en petit volume chez les fournisseurs.

La source de fréquence est un point délicat : nous nous imposons la résolution spectrale d'un DDS, mais ces composants dépassent souvent notre budget (prés de 30 euros pour un AD9954, 22 euros pour un AD9851), et d'autre part nous désirons alimenter l'ensemble du circuit par le régulateur de tension fourni par le convertisseur USB-RS232. La consommation totale du circuit ne doit donc pas dépasser 50 mA. Heureusement, Analog Devices propose désormais un DDS cadencé à 75 MHz (donc largement suffisant pour caractériser des dispositifs jusqu'à quelques dizaines de MHz), de faible consommation, et à coût réduit, l'AD9834 (9 euros chez Farnell). Nous évitons par ailleurs d'empiéter sur le budget en obtenant ce composant ainsi que le comparateur comme échantillon chez le fabriquant.

Finalement, en ajoutant la détection de phase et de magnitude, avec quelques amplificateurs opérationnels de mise en forme des signaux, nous obtenons un circuit nécessitant moins de 30 euros de composants.

3 Conception du circuit imprimé

La conception du circuit imprimé n'est pas anodine car le coût de fabrication du circuit est proportionnel à sa surface : un circuit compact est donc une garantie de gestion efficace du budget. Par ailleurs, un circuit voit son coût croître avec le nombre de couches : bien qu'il soit souvent difficile de gérer les composants numériques aussi complexes qu'un microcontrôleur à 64 broches sur une simple face, il est rare de devoir dépasser les 2 couches, sauf dans le cas particulier des circuits radiofréquences pour lesquels le 4-couches se justifie, avec des plans d'alimentation et de masse internes, le routage des signaux numériques et radiofréquences étant repoussé aux deux couches externes. Ici, 2 couche suffisent, avec une distribution de la masse par plan de masse sur chaque couche, et peu de soucis de gestion des signaux radiofréquences qui sont confinés à un coin du circuit imprimé, sans aller empiéter dans la zone numérique.

Le circuit est centré autour d'un STM32 et de sa déclinaison la plus récente, le STM32F4. Un compromis coût-performance-disponibilité est atteint avec le STM32F410RBT6 proposé pour moins de 4 euros à l'unité, avec une mémoire conséquente (32 KB de RAM et 128 KB de flash) pour le projet que

1. `https://github.com/libopencm3/libopencm3`
2. `https://github.com/libopencm3/libopencm3-examples`

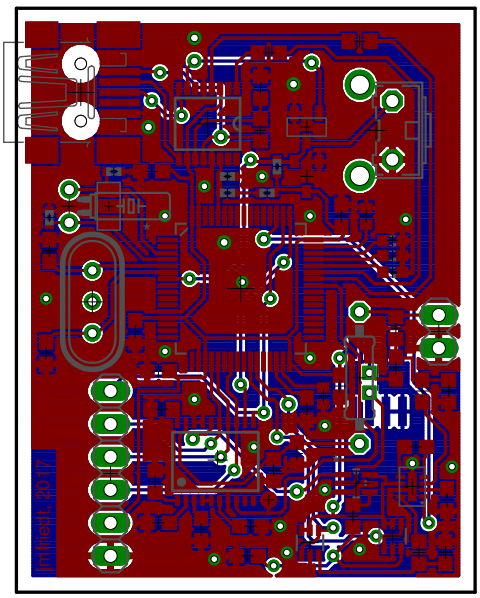


Figure 3: Circuit double faces comprenant un STM32F410, un synthétiseur de fréquence numérique (DDS) comme source de signaux radiofréquences, et pour la mesure un détecteur de phase basé sur un XOR et un détecteur de puissance.

nous nous proposons d'atteindre. Cette architecture est supportée par `libopencm3` (`https://github.com/libopencm3/libopencm3`) qui nous facilitera la phase de prise en main d'un microprocesseur aux périphériques nombreuses et relativement complexes. L'utilisation de cette bibliothèque ne nous affranchira pas d'une lecture minutieuse des documentations techniques. Le DDS est cablé selon le schéma proposé dans la documentation technique, avec une sortie unipolaire. Au lieu d'utiliser une simple diode pour la détection de puissance, avec les risques de manquer de sensibilité, nous exploitons un composant dédié à cette tâche, le LT5537 de Linear Technology. Enfin, le détecteur de phase est formé d'une porte XOR suivie d'un filtre passe-bas. Dans tous les cas, nous prenons soin de fournir un signal de faible impédance aux convertisseurs analogique- numériques en les précédant d'amplificateurs opérationnels. Les condensateurs de découplage sont fondamentaux pour un circuit qui fonctionnera à plusieurs dizaines de megahertz : chaque point d'alimentation se voit associé, au plus près, d'un condensateur de découplage de 100 nF, compromis entre une bonne capacité de filtrage et un comportement radiofréquence proche du condensateur et loin de l'inductance. L'ensemble du circuit est alimenté par la sortie régulée en 3,3 V du convertisseur RS232-USB, dont on aura pris soin de vérifier qu'il est capable de fournir la puissance nécessaire à tous les composants dans le pire cas d'utilisation. Le schéma résultant est disponible sur la Fig. 4.

4 Aspects logiciels : programmation des périphériques

4.1 Cadencement sur le quartz externe

Par défaut, `libopencm3` ne prévoit qu'une configuration des diverses horloges internes au microcontrôleur sur une source multiple de 8 MHz. Compte tenu des composants disponibles, nous avons choisi un quartz de fréquence de résonance 20 MHz, qui nécessite donc de reconfigurer les horloges. Par ailleurs, nous désirons cadencer le DDS avec une fréquence aussi élevée que possible, et ce sur une sortie de *timer* du microcontrôleur. Cependant, le *timer* ne peut être cadencé au plus vite qu'à la fréquence du cœur du processeur divisée par 2 (une période haute et une période basse). Ainsi, pousser le cœur du processeur au-delà de 100 MHz – et donc au-delà de ses spécifications (une très mauvaise pratique en terme de production) serait souhaitable pour pouvoir émettre un signal dans la bande FM commerciale (88–108 MHz). En effet, un DDS possède la propriété de non seulement émettre sur la fréquence programmée f, mais aussi sur la fréquence d'horloge à laquelle s'ajoute la fréquence programmée $f_{CK} + f$. Si $f_{CK} = 60 = 120/2$ MHz, alors la bande FM est accessible en programmant $f = 28$ MHz. Cette condition ne respecte pas $f < f_{CK}/3$ qui permettrait d'implémenter des filtres adéquats pour une sortie monochromatique, mais est techniquement réalisable. Nous verrons plus loin que nous serons capables de cadencer à plus de 140 MHz un cœur de processeur conçu pour fonctionner à 100 MHz.

La configuration des sources d'horloge doit respecter un certain nombre de contraintes, en particulier le fait que la source de fréquence des périphériques APB2 ne doit pas dépasser 100 MHz (dans la famille des F4, 72 MHz sinon), APB1 la moitié de cette valeur, et la fréquence du cœur du processeur est issue d'une multiplication par PLL fractionnaire de la fréquence du quartz selon **SysCoreClock=((HSE/PLL_M)*PLL_N)/PLL_P**.

Un outil de configuration des horloges – sous forme de feuille de calcul Excel – est fourni par ST Microelectronics dans sa note d'application AN3988 et le logiciel associé STSW-STM32091.

Le résultat d'une configuration tenant compte d'un résonateur 20 MHz pour sur-cadencer (*overclock*) le cœur du microcontrôleur à 140 MHz donne :

```
1 // APB2 max=100 MHz but when the APB prescaler is NOT 1, the interface clock is fed
```

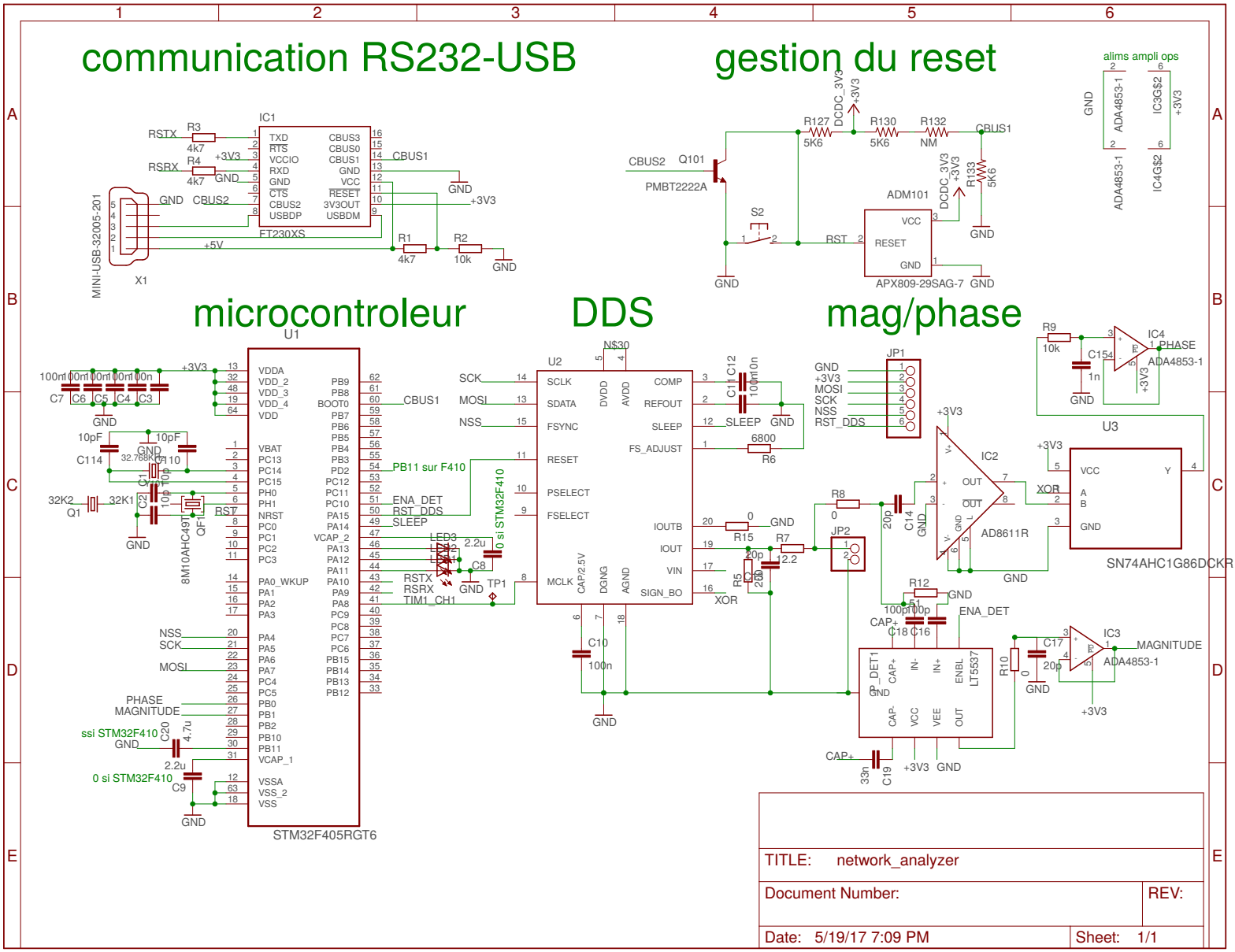


FIGURE 4 – Schéma du circuit : à gauche le microcontrôleur, au milieu le DDS, à droite le détecteur de phase (haut) et la mesure de puissance (bas). En haut à gauche le convertisseur USB-RS232 aussi chargé de fournir la source de tension régulée. Finalement, en haut à droite une petite série de circuits chargés de manipuler les signaux de programmation du microcontrôleur et fournir un signal de ré-initialisation propre à la mise sous tension.

```
2  // twice the frequency => Sysclk = 140 MHz, APB2=2 but Timers are driven at twice that is 140.
3  const struct rcc_clock_scale rcc_hse_20mhz_3v3 = {
4                  .pllm = 20,                              // 20/20=1 MHz
5                  .plln = 280,                             // 1*280/2=140 MHz
6                  .pllp = 2,                               //       ^
7                  .pllq = 6,
8                  .pllr=0,
9                  .pll_source = RCC_CFGR_PLLSRC_HSE_CLK, // 190913 mandatory to use external clock (!=HSI)
10                 .hpre = RCC_CFGR_HPRE_DIV_NONE,
11                 .ppre1 = RCC_CFGR_PPRE_DIV_4,
12                 .ppre2 = RCC_CFGR_PPRE_DIV_2,
13                 .flash_config = FLASH_ACR_ICEN | FLASH_ACR_DCEN |
14                                 FLASH_ACR_LATENCY_4WS, // 4 WS d'apres configuration par ST
```

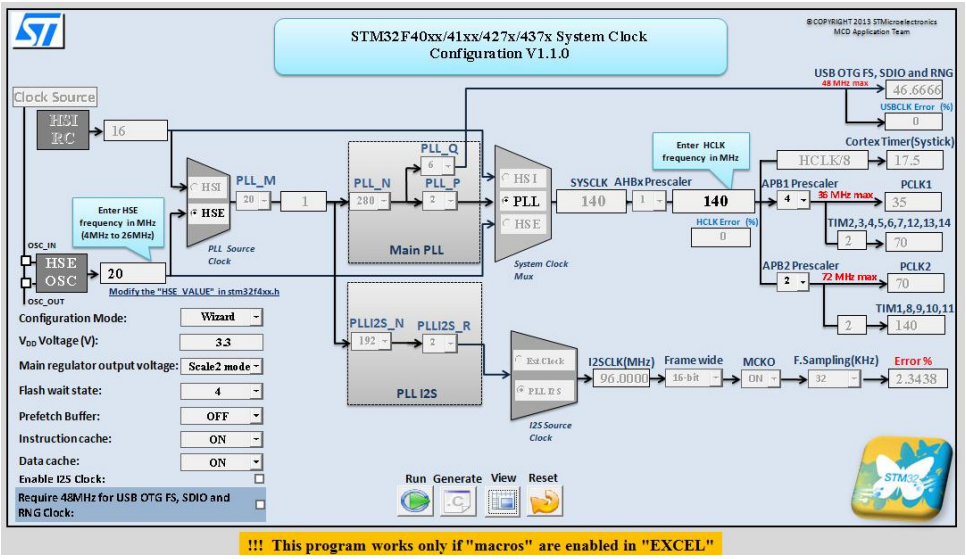


FIGURE 5 – Configuration des horloges pour un cadencement par un quartz à 20 MHz.

```
15                .ahb_frequency  = 140000000,
16                .apb1_frequency = 35000000,
17                .apb2_frequency = 70000000,
18        };
19
20  /**
21    * @file    system_stm32f4xx.c
22    * @author  MCD Application Team
23    *============================================================================
24    *        Supported STM32F40xx/41xx/427x/437x devices
25    *        System Clock source                    | PLL (HSE)
26    *        SYSCLK(Hz)                             | 140000000
27    *        HCLK(Hz)                               | 140000000
28    *        AHB Prescaler                          | 1
29    *        APB1 Prescaler                         | 4
30    *        APB2 Prescaler                         | 2
31    *        HSE Frequency(Hz)                      | 20000000
32    *        PLL_M                                  | 20
33    *        PLL_N                                  | 280
34    *        PLL_P                                  | 2
35    *        PLL_Q                                  | 6
36    *        Flash Latency(WS)                      | 4
37    *        Require 48MHz for USB OTG FS,          | Disabled
38    *        SDIO and RNG clock                     |
39  */
40
41  static void clock_setup(void)
42  {// rcc_clock_setup_hse_3v3(&rcc_hse_20mhz_3v3); // obsolete
43   rcc_clock_setup_pll(&rcc_hse_20mhz_3v3);        // custom clock configuration
44   [...]
45  }
46
47  static void init_usart (void)
48  {[...]
49   usart_set_baudrate (USART1, (115200));
50   [...]
51  }
52
53  int main(void)
54  {clock_setup();
55   init_usart();
56   [...]
57  }
```

Noter que cette configuration des fréquences d'horloge et les constantes associées permet d'informer du baudrate recherché sans nécessiter de bricoler des valeurs adaptées par une utilisation de valeurs par défaut supposant un quartz multiple de 8 ou 25 MHz. Par ailleurs, la cadencement du timer sera particulier : étant donné que nous devons diviser l'horloge pour maintenir APB2 sous 100 MHz, nous pourrions croire que la fréquence maximale de cadencement du timer serait 100 MHz se traduisant par une sortie à 100/2=50 MHz (une période haute et une période basse en sortie du timer). Il n'en est rien : le timer est un cas particulier dans lequel, si le facteur de division de APB2 n'est pas unitaire, alors le périphérique est cadencé par la fréquence double de APB2. Cela se traduit par une fréquence maximale de sortie du timer de 84 MHz. Dans notre cas, nous avons choisi de multiplier par 7 par PLL le quartz 20 MHz pour donner 140 MHz (soit 40% au dessus de la norme d'utilisation du microcontrôleur), ce qui permet de cadencer en sortie de timer le DDS à 70 MHz. Cette valeur, proche du maximum de 75 MHz (si on ne cherche pas à sur-cadencer le DDS – ce que nous pourrions aussi faire), permet d'optimiser la gamme de fonctionnement du synthétiseur numérique de fréquence.

4.2 GPIO

LED1	PA11
LED2	PA12
LED3	PA13

L'initialisation des GPIO ne présente aucune difficulté si ce n'est de ne *pas* tenter d'accéder à un périphérique qui n'existe pas sur une déclinaison du STM32 utilisé. Par exemple le STM32F410 n'implémente pas le port D qui est néanmoins utilisé dans la STM32F4-Discovery : bien prendre soin d'utiliser la compilation conditionnelle pour sélectionner le port effectivement disponible sur une plateforme donnée. Cet exemple montre comment initialiser l'horloge globale du système, l'horloge du périphérique utilisé, configurer le GPIO dans sa fonction et sa direction, et finalement dans la boucle infinie faire clignoter les LEDs.

```
1  #include <libopencm3/stm32/rcc.h>
2  #include <libopencm3/stm32/gpio.h>
3  #include <libopencm3/stm32/flash.h> // definitions du timer
4
5  void core_clock_setup(void)
6  { // rcc_clock_setup_hse_3v3(&rcc_hse_20mhz_3v3); // custom version
7    rcc_clock_setup_pll(&rcc_hse_20mhz_3v3);
8  }
9
10 void init_gpio(void)
11 { gpio_mode_setup(GPIOA, GPIO_MODE_OUTPUT, GPIO_PUPD_NONE, GPIO11|GPIO12|GPIO13);
12 }
13
14 void clock_setup(void)
15 { core_clock_setup();
16   rcc_periph_clock_enable(RCC_GPIOA);
17 }
18
19 void led_set(int msk)
20 { gpio_set(GPIOA, msk);
21 }
22
23 void led_clr(int msk)
24 { gpio_clear(GPIOA, msk);
25 }
26
27 int main()
28 {int msk=(1<<11)|(1<<12)|(1<<13);
29  clock_setup();
30  init_gpio();
31  while(1)
32   {led_set(msk); delai();
33    led_clr(msk); delai();
34   }
35  return 0;
36 }
```

4.3 GPIO et RS232

STM32 TX	PA9
STM32 RX	PA10

```
// regarder sur le connecteur, broche la plus proche du CPU (RST_DDS) : clignote
//   + message a 9600 bauds sur UART
#include <libopencm3/stm32/rcc.h>
#include <libopencm3/stm32/gpio.h>
#include <libopencm3/stm32/exti.h>
#include <libopencm3/cm3/nvic.h>
#include <libopencm3/stm32/timer.h>
#include <libopencm3/stm32/usart.h>
#include <libopencm3/stm32/f4/memorymap.h>

char buf[20]="Hello world\r\n\0";

static void jmf_putchar (unsigned char ch)   // Write character to Serial Port
{usart_send_blocking (USART1, ch);}

static void jmfputs (char *s_de_char)
{int j = 0;
 int d;
 do
   {d = s_de_char[j];
    if (d != 0) jmf_putchar (d);
    j++;
   }
 while (d != 0);
}

static void delay(unsigned int delay)
{volatile unsigned int i;
 for (i = 0; i < delay; i++)   /* Wait for N=delay cycles of clock.(delay*(1/168e6) here) */
    __asm__("nop");
}

static void gpio_setup(void)
{rcc_periph_clock_enable(RCC_GPIOA); // Enable GPIOA clock.
 gpio_mode_setup(GPIOA, GPIO_MODE_OUTPUT,GPIO_PUPD_NONE, GPIO15);// Enable LED pin (DDS CS)
}

static void init_usart (void)
{rcc_periph_clock_enable(RCC_USART1);
 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);
  // Setup USART1 parameters.
 usart_set_baudrate (USART1, (9600));
 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);              // Finally enable the USART.
}

int main(void)
{core_clock_setup();
 gpio_setup();
 init_usart();
 while (1) {
   delay(15000);
   gpio_toggle(GPIOA,GPIO15); // DDS reset
   jmfputs(buf);
 }
 return 0;
}
```

4.4 GPIO, RS232 et interruption

Par défaut les broches numériques sont configurées en entrées-sorties programmables généralistes (*General Purpose Input-Output*, GPIO). Toutes les broches numériques présentent des fonctions alternatives, qu'il s'agisse de *timer* ou communication numérique (synchrone ou asynchrone). Le STM32 permet une grande variété de combinaison et de routage de signaux vers diverses broches : il s'agit des fonctions alternatives AF. La Fig. 6 résume le choix de quelle fonction alternative connecte quel signal à quelle broche.

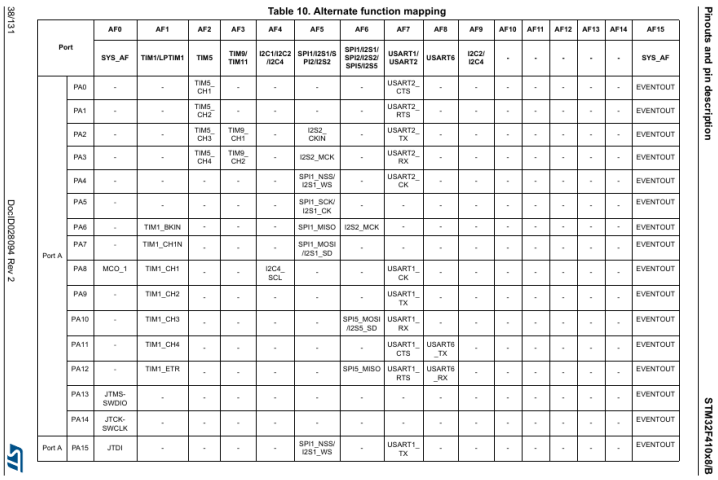


Table 10. Alternate function mapping

Port		AF0	AF1	AF2	AF3	AF4	AF5	AF6	AF7	AF8	AF9	AF10	AF11	AF12	AF13	AF14	AF15
		SYS_AF	TIM1/LPTIM1	TIM5	TIM9/ TIM11	I2C1/I2C2 /I2C4	SPI1/I2S1/S PI2/I2S2	SPI1/I2S1/ SPI2/I2S2/ SPI5/I2S5	USART1/ USART2	USART6	I2C2/ I2C4	-	-	-	-	-	SYS_AF
Port A	PA0	-	-	TIM5_ CH1	-	-	-	-	USART2_ CTS	-	-	-	-	-	-	-	EVENTOUT
	PA1	-	-	TIM5_ CH2	-	-	-	-	USART2_ RTS	-	-	-	-	-	-	-	EVENTOUT
	PA2	-	-	TIM5_ CH3	TIM9_ CH1	-	I2S2_ CKIN	-	USART2_ TX	-	-	-	-	-	-	-	EVENTOUT
	PA3	-	-	TIM5_ CH4	TIM9_ CH2	-	I2S2_MCK	-	USART2_ RX	-	-	-	-	-	-	-	EVENTOUT
	PA4	-	-	-	-	-	SPI1_NSS/ I2S1_WS	-	USART2_ CK	-	-	-	-	-	-	-	EVENTOUT
	PA5	-	-	-	-	-	SPI1_SCK/ I2S1_CK	-	-	-	-	-	-	-	-	-	EVENTOUT
	PA6	-	TIM1_BKIN	-	-	-	SPI1_MISO	I2S2_MCK	-	-	-	-	-	-	-	-	EVENTOUT
	PA7	-	TIM1_CH1N	-	-	-	SPI1_MOSI /I2S1_SD	-	-	-	-	-	-	-	-	-	EVENTOUT
	PA8	MCO_1	TIM1_CH1	-	-	I2C4_ SCL	-	-	USART1_ CK	-	-	-	-	-	-	-	EVENTOUT
	PA9	-	TIM1_CH2	-	-	-	-	-	USART1_ TX	-	-	-	-	-	-	-	EVENTOUT
	PA10	-	TIM1_CH3	-	-	-	-	SPI5_MOSI /I2S5_SD	USART1_ RX	-	-	-	-	-	-	-	EVENTOUT
	PA11	-	TIM1_CH4	-	-	-	-	-	USART1_ CTS	USART6 _TX	-	-	-	-	-	-	EVENTOUT
	PA12	-	TIM1_ETR	-	-	-	-	SPI5_MISO	USART1_ RTS	USART6 _RX	-	-	-	-	-	-	EVENTOUT
	PA13	JTMS-SWDIO	-	-	-	-	-	-	-	-	-	-	-	-	-	-	EVENTOUT
	PA14	JTCK-SWCLK	-	-	-	-	-	-	-	-	-	-	-	-	-	-	EVENTOUT
Port A	PA15	JTDI	-	-	-	-	SPI1_NSS/ I2S1_WS	-	USART1_ TX	-	-	-	-	-	-	-	EVENTOUT

FIGURE 6 – Fonctions alternatives des diverses broches numériques du microcontrôleur.

```
// regarder sur le connecteur, broche la plus proche du CPU (RST_DDS) : clignote
//   + message a 9600 bauds sur UART + premiere lettre du msg = uart input
[...]

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

4.5 SPI

Les transactions entre le microcontrôleur et le DDS se font au travers d'un bus synchrone, SPI. Nous apprenons sur la datasheet les deux caractéristiques ajustables de ce bus : l'état au repos de l'horloge (état haut) et le front d'horloge sur lequel le signal de données est échantillonné (descendant).

CS#	PA15
MOSI	PA7
MISO	PA6
CK	PA5

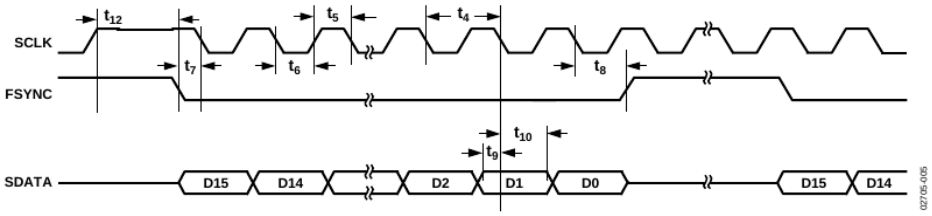
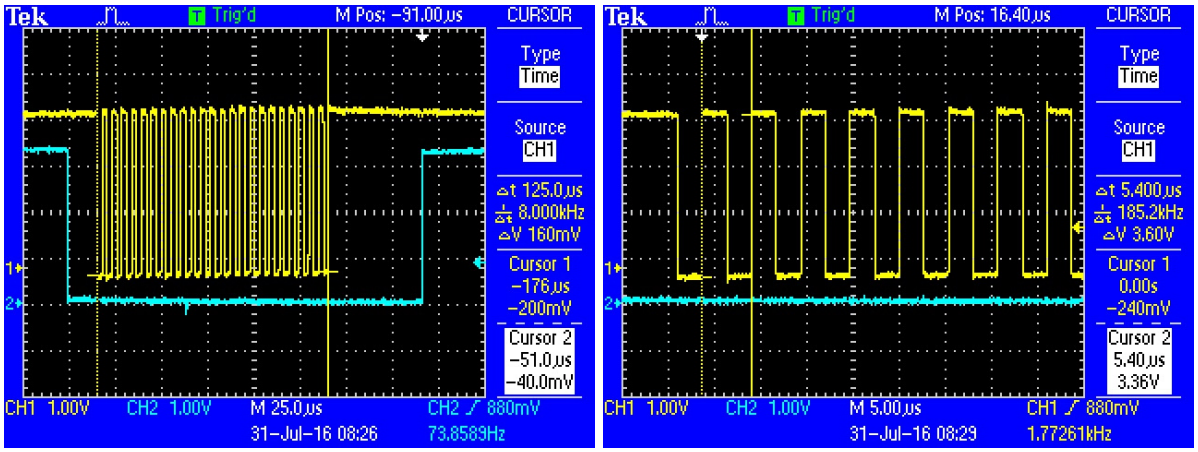


Figure 5. Serial Timing



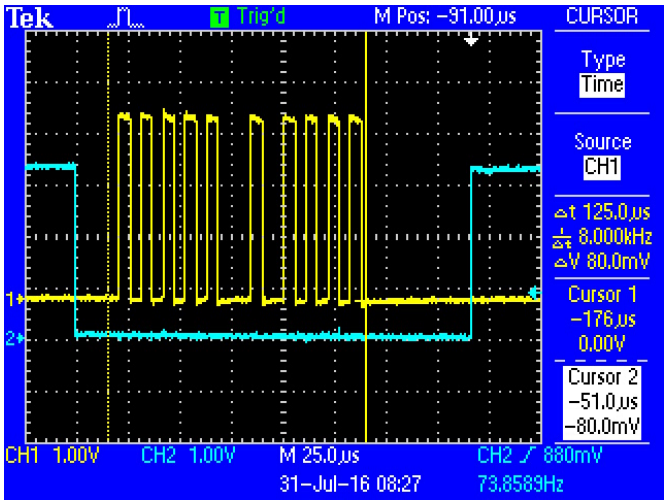


FIGURE 7 – Signaux du bus SPI : en haut l'horloge, en bas les données lorsque nous transmettons le triplet {0x55,0x44,0xaa}.

```
1 [...]
2 #include <libopencm3/stm32/spi.h>
3
4 static void delay(unsigned int );
5 static void clock_setup(void);
6 static void init_gpio(void);
7 static void init_usart (void);
8 void init_spi (void);
```

```
 9 void init_time1 (void);
10 void dds_cs_set (void);
11 void dds_cs_clr (void);
12 void envoi_DDS (unsigned char *, int);
13
14 [...]
15 static void clock_setup(void)
16 {rcc_periph_clock_enable (RCC_SPI1);
17 }
18
19 void init_spi (void)                    //initialisation spi et gpio pour DDS
20 {gpio_mode_setup (GPIOA, GPIO_MODE_AF, GPIO_PUPD_PULLUP, GPIO5 | GPIO6 | GPIO7); // GPIO in output
21  gpio_set_af (GPIOA, GPIO_AF5, GPIO5 | GPIO6 | GPIO7);   //GPIO in SPI mode GPIOA5=SCK 6=MISO 7=MOSI
22  //MSB first is th default configuration for the DDS AD9959 obtained after a master reset.
23  // le DDS prend les donn'ees en falling edge.
24  spi_disable (SPI1);    //turn off SPI for configuration
25  spi_init_master (SPI1,
26 //                      SPI_CR1_BAUDRATE_FPCLK_DIV_16,//DIV_16: 2.63MHz
27 //                      0x28, //les trois derniers bits ne servent pas: 0x1A=0x18
28 //                 SPI_CR1_BAUDRATE_FPCLK_DIV_64,
29    SPI_CR1_BAUDRATE_FPCLK_DIV_256, SPI_CR1_CPOL_CLK_TO_1_WHEN_IDLE, // clk to 1 when idle
30    SPI_CR1_CPHA_CLK_TRANSITION_1,   // falling edge
31    SPI_CR1_DFF_8BIT, //8 bits
32    SPI_CR1_MSBFIRST);   //MSB first
33   spi_enable_software_slave_management (SPI1);
34   spi_set_nss_high (SPI1); // SPI1 active
35   spi_enable (SPI1);
36 }
37
38 void dds_cs_set () {gpio_set (GPIOA, GPIO4);}
39 void dds_cs_clr () {gpio_clear(GPIOA, GPIO4);}
40
41 void envoi_DDS (unsigned char *entree, int n)
42 { unsigned char i;
43   dds_cs_set (); delay (100);
44   dds_cs_clr (); delay (100);
45   for (i = 0; i < n; i++) spi_send (SPI1, entree[i]);
46 //      while ((SPI_SR(SPI2) & 0x40)==0x00); //(!(SPI_SR(SPI2) & SPI_SR_TXE));
47   delay (1000); // JMF : a revoir
48   dds_cs_set ();
49 }
50
51 [...]
52 int main(void)
53 {unsigned char entree[3]={0x55,0x44,0xaa};
54  core_clock_setup();
55  clock_setup();
56  init_gpio();
57  init_usart();
58  init_spi();
59  while (1) {
60    delay(15000);
61    gpio_toggle(GPIOA,GPIO15); // DDS reset
62    jmfputs(buf);
63    envoi_DDS(entree,3);
64  }
65  return 0;
66 }
```

4.6 Timer

Le DDS est un composant numérique nécessitant une horloge maîtresse pour se cadencer. Afin d'éviter d'ajouter un composant additionnel – un oscillateur – et se laisser la liberté d'ajuster cette horloge selon les conditions d'utilisation, nous décidons de cadencer le DDS par la sortie d'un timer du microcontrôleur. Ce faisant, la fréquence maximale de commande du DDS est au mieux la fréquence de cadencement du microconôleur divisée par deux (le timer est cadencé par la fréquence de cœur du processeur, et il faut une période haute et une période basse, chacune cadencée sur cette horloge du cœur, donc la fréquence

est divisée par deux). Parmi les divers timers, nous avons choisi le canal 1 du timer 1.

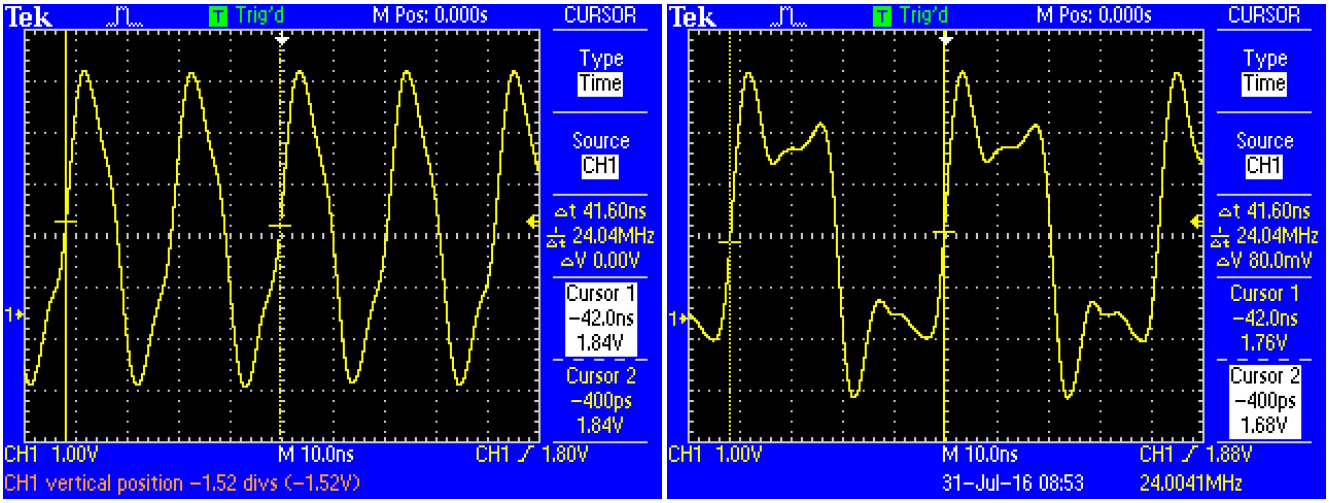


FIGURE 8 – Sortie timer 1 : à gauche avec un prescaler de 0, à droite avec un pre-scaler de 1. Ce signal alimentera l'horloge principale du DDS.

MCK	PA8=TIM1_CH1

```
1  [...]
2  #include <libopencm3/stm32/timer.h>
3
4  static void clock_setup(void)
5  {rcc_periph_clock_enable (RCC_SPI1);
6   rcc_periph_clock_enable (RCC_TIM1);
7   rcc_periph_clock_enable(RCC_USART1);
8   rcc_periph_clock_enable(RCC_GPIOA); // Enable GPIOA clock.
9  }
10
11 void init_time1 (void)
12 {gpio_set_output_options(GPIOA, GPIO_OTYPE_PP, GPIO_OSPEED_50MHZ, GPIO8 );
13  gpio_mode_setup (GPIOA, GPIO_MODE_AF, GPIO_PUPD_NONE, GPIO8);
14  gpio_set_af (GPIOA, GPIO_AF1, GPIO8);
15  // timer_reset (TIM1); // modification du 180826 /w new libopencm3 version
16  rcc_periph_reset_pulse(RST_TIM1); // voir https://github.com/libopencm3/libopencm3-examples/commits/master/examples/stm32/f
17  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 TI
18
19 // https://github.com/libopencm3/libopencm3/blob/master/lib/stm32/common/timer_common_all.c
20   timer_set_oc_mode(TIM1, TIM_OC1, TIM_OCM_PWM2);
21   timer_enable_oc_output(TIM1, TIM_OC1);
22   timer_enable_break_main_output(TIM1);
23   timer_set_oc_value(TIM1, TIM_OC1, 1);
24
25   timer_set_prescaler (TIM1, 1);
26   timer_set_period (TIM1, 1); // pourquoi 1 et pas 2 ? il FAUT que period >= OC1
27   timer_enable_counter (TIM1);
28 }
29
30 [...]
31 int main(void)
32 {unsigned char entree[3]={0x55,0x44,0xaa};
33  core_clock_setup();
34  clock_setup();
35  init_gpio();
36  init_usart();
37  init_spi();
38  init_time1();
39  while (1) {
40    delay(15000);
41    gpio_toggle(GPIOA,GPIO15); // DDS reset
42    jmfputs(buf);
```

```
   envoi_DDS(entree,3);
 }
 return 0;
}
```

4.7 Organisation des données

Le DDS attend les informations de configuration de la fréquence dans un ordre bien précis. Les transactions de font par paquets de 16 bits, déclenchées par l'abaissement de FSYNC et achevées par la remontée de ce signal. Soit les 28 bits définissant la fréquen sont envoyés en deux coups successifs, avec le bit 13 du registre de contrôle définissant s'il s'agit du mot de poids fort ou faible ; soit les 28 bits sont fournis sous forme d'une transaction de 32 bits, avec chaque mot de 16 bits préfixé de l'adresse définissant quelle fréquence – mot 0 ou 1 – est mis à jour.

DB15	DB14	DB13	DB12	DB11	DB10	DB9	DB8	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	B28	HLB	FSEL	PSEL	PIN/SW	RESET	SLEEP1	SLEEP12	OPBITEN	SIGN/PIB	DIV2	0	MODE	0

Table 6. Description of Bits in the Control Register

Bit	Name	Description
DB13	B28	Two write operations are required to load a complete word into either of the frequency registers. B28 = 1 allows a complete word to be loaded into a frequency register in two consecutive writes. The first write contains the 14 LSBs of the frequency word and the next write contains the 14 MSBs. The first two bits of each 16-bit word define the frequency register the word is loaded to and should, therefore, be the same for both of the consecutive writes. Refer to Table 10 for the appropriate addresses. The write to the frequency register occurs after both words have been loaded. An example of a complete 28-bit write is shown in Table 11. Note however, that consecutive 28-bit writes to the same frequency register are not allowed, switch between frequency registers to do this type of function. B28 = 0, the 28-bit frequency register operates as two 14-bit registers, one containing the 14 MSBs and the other containing the 14 LSBs. This means that the 14 MSBs of the frequency word can be altered independent of the 14 LSBs, and vice versa. To alter the 14 MSBs or the 14 LSBs, a single write is made to the appropriate frequency address. The Control Bit DB12 (HLB) informs the AD9834 whether the bits to be altered are the 14 MSBs or 14 LSBs.
DB12	HLB	This control bit allows the user to continuously load the MSBs or LSBs of a frequency register ignoring the remaining 14 bits. This is useful if the complete 28-bit resolution is not required. HLB is used in conjunction with DB13 (B28). This control bit indicates whether the 14 bits being loaded are being transferred to the 14 MSBs or 14 LSBs of the addressed frequency register. DB13 (B28) must be set to 0 to be able to change the MSBs and LSBs of a frequency word separately. When DB13 (B28) = 1, this control bit is ignored. HLB = 1 allows a write to the 14 MSBs of the addressed frequency register. HLB = 0 allows a write to the 14 LSBs of the addressed frequency register.

Figure 9: Registre de contrôle du DDS, tel que décrit dans la datasheet du AD9834.

Le passage d'un mot de 32 bits b définissant la fréquence f_0 telle que

$$b = \frac{f_0}{f_{CK}} \cdot 2^{28}$$

avec f_{CK} la fréquence cadençant le DDS, vers deux mots de poids faible et fort successivement [3] (selon l'ordre imposé par le protocole de communication) s'obtient par

```
void dds_cs_set () {gpio_set   (GPIOA, GPIO4);}
void dds_cs_clr () {gpio_clear(GPIOA, GPIO4);}
void dds_rst_set (){gpio_set   (GPIOA, GPIO15);}
void dds_rst_clr (){gpio_clear(GPIOA, GPIO15);}
void dds_slp_clr (){gpio_clear(GPIOA, GPIO14);}
void dds_slp_set (){gpio_set   (GPIOA, GPIO14);}

void envoi_DDS (unsigned short entree)  // 16 bit sent
{ dds_cs_clr (); delay (100);
  spi_send (SPI1, (entree>>8)&0xff);
  spi_send (SPI1, (entree&0xff));
//      while ((SPI_SR(SPI2) & 0x40)==0x00); //(!(SPI_SR(SPI2) & SPI_SR_TXE));
  delay (1000);
  dds_cs_set ();
  delay (100);
}

int main()
{unsigned long frequence=0x028f5c2; // 0.1 MHz/10 MHz*2^28
[...]
 dds_cs_set();
 dds_slp_clr();
 dds_rst_set();
 delay(15000);
 dds_rst_clr();
 delay(15000);
 envoi_DDS((frequence&0x3fff)|0x4000);       // LSB
 envoi_DDS(((frequence>>14)&0x3fff)|0x4000); // MSB

 envoi_DDS((frequence&0x3fff)|0x8000);       // LSB
```

3. L. Riordan, "Programming the AD9833/AD9834", note d'application AN-1070 de Analog Devices

```
 envoi_DDS(((frequence>>14)&0x3fff)|0x8000); // MSB
}
```

4.8 ADC

Nous avons choisi de connecter les mesures de phase et amplitude sur deux convertisseurs que sont PB0 et PB1 respectivement. La documentation (`en.DM002` table 9 p.35) nous informe que ces deux convertisseurs sont associés aux positions 8 et 9 du multiplexeur de ADC1.
La mise en œuvre des convertisseurs analogi numériques nécessite d'activer la fonction analogique de la broche (option `GPIO_M` de `gpio_mode_setup()`), l'horloge associée, et lancer manuellement une conversion (`adc_start_conversion_regular()` après avoir informé le convertisseur du canal à mesurer.

Table 9. STM32F410x8/B pin definitions (continued)

Pin Number			Pin name (function after reset)(1)	Pin type	I/O structure	Notes	Alternate functions	Additional functions
WLCSP36	UFQFPN48	LQFP64						
-	18	26	PB0	I/O	FT	-	TIM1_CH2N, SPI5_SCK/I2S5_CK, EVENTOUT	ADC1_8
-	19	27	PB1	I/O	TC	-	TIM1_CH3N, SPI5_NSS/I2S5_WS, EVENTOUT	ADC1_9

Figure 10: Fonctions analogiques

```
static void clock_setup(void)
{[...]
 rcc_periph_clock_enable(RCC_ADC1);  //clock ADC
}

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()
{[...]
 init_adc();

 while (1) {
   [...]
   v=read_adc(8);jmf_put16(v); jmf_putchar ('␣');
   v=read_adc(9);jmf_put16(v); jmf_putchar ('␣');jmf_puts("\r\n\0");
}
```

Nous vérifions en connectant les vias allant à ces deux entrées analogiques tantôt à la masse, tantôt à la tension d'alimentation, que la mesure est bien 0 ou 0xFFF, tel que prévu pour un convertisseur codé

sur 12 bits.

5 Analyse du synthétiseur numérique de fréquence (DDS)

La sortie du DDS fournit l'opportunité de valider le bon fonctionnement des convertisseurs analogiques et d'établir leur fréquence d'échantillonnage (Fig. 11) sur un signal variable.

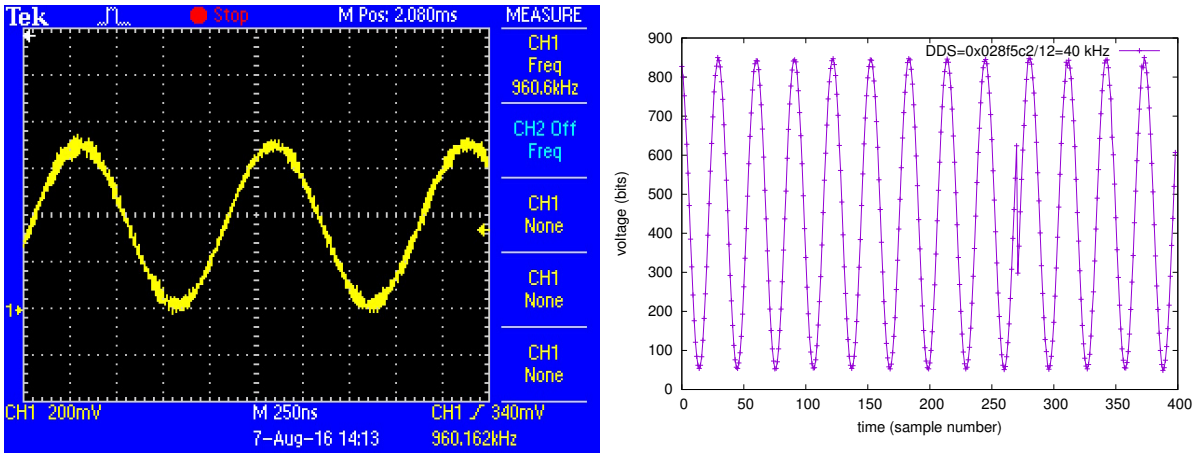


FIGURE 11 – Gauche : capture à l'oscilloscope de la sortie du DDS configuré pour générer une sortie à 960 kHz, soit $f_{CK}/50$. Droite : capture par ADC de la sortie du DDS programmé pour fournir 40 kHz. Avec 32 points/période, la fréquence d'échantillonnage est de 1,28 MHz, proche de la borne de 2,4 MHz puisque nous échantillonnons alternativement deux voies.

Le synthétiseur direct de fréquence (DDS) est un composant numérique, qui respecte donc les contraintes d'un signal échantillonné en temps discret, à savoir convolution de la fonction de transfert par un sinus cardinal (Fig. 12) et repliement spectral. Il est habituel d'affirmer que la fréquence générée par un DDS f_0 doit se trouver au plus au tiers de la fréquence d'horloge f_{CK} afin d'avoir suffisamment de place pour implémenter le filtre passe-bas visant à rejeter les raies parasites. Au contraire, si nous voulons sonder la réponse d'un dispositif de bande passante réduite – qui fera donc lui même office de filtre – on pourra volontairement exploiter une des raies considérées habituellement comme parasites : $f_{CK} - f_0$ ou $f_{CK} + f_0$ pour les plus puissantes. Cela signifie qu'un DDS cadencé à 75 MHz et programmé pour générer un signal à 25 MHz produit une raie à 100 MHz, en plein dans la bande FM commerciale ... (Fig. 13).

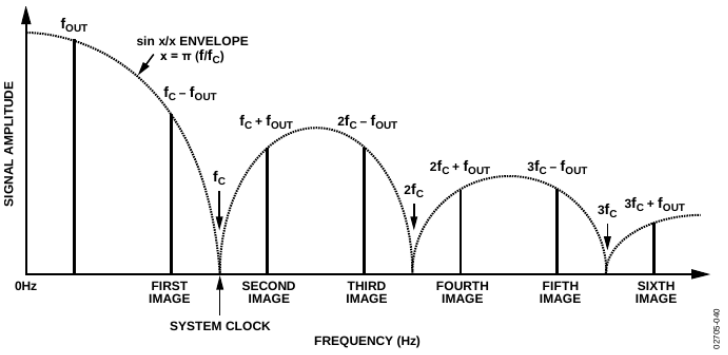


Figure 28. The DAC Output Spectrum

Figure 12: Le DDS est un composant numérique synchrone : il respecte la fonction de transfert d'un DAC.

6 Émission dans la bande FM

Le DDS permet de rapidement reprogrammer sa fréquence d'émission, ce qui revient à effectuer de la modulation de fréquence. Il nous faut, pour obtenir un signal accessible avec tout récepteur FM, atteindre la bande FM commerciale comprise entre 88 et 108 MHz. Pour ce faire, nous devons générer un signal de cadencement du DDS qui soit suffisamment élevé : cela nécessite de sur-cadencer [4] (*overclock*) le microcontrôleur pour générer un MCLK suffisamment important. Nous vérifions que le microcontrôleur semble encore fonctionner correctement à 134,4 MHz (contre une fréquence nominale maximale de 100 MHz),

4. `stm32f4-discovery.net/2014/11/overclock-stm32f4-device-up-to-250mhz/`

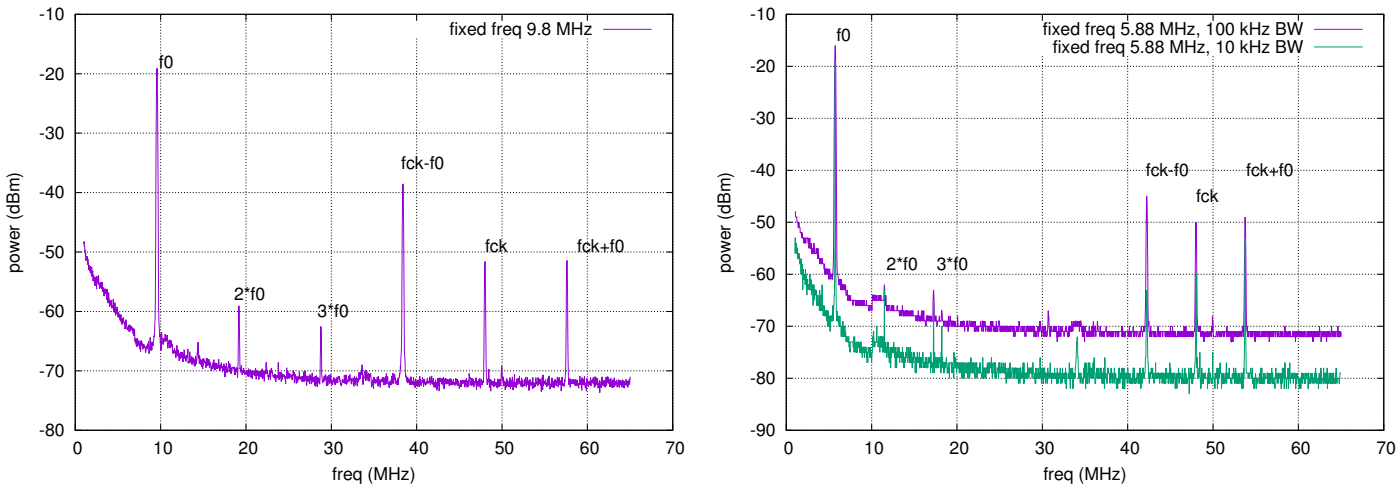


FIGURE 13 – Illustrations des diverses composantes spectrales générées par un DDS. Noter l'influence de la bande passante de mesure – IF BW – sur la largeur des raies mais surtout sur le plancher de bruit.

ce qui permet de générer un signal en sortie de timer à 67,2 MHz. Nous atteignons donc la partie basse de la bande FM commerciale (90 MHz) en programmant la fréquence de sortie $f_0 = 22,8$ MHz (Fig. 14).

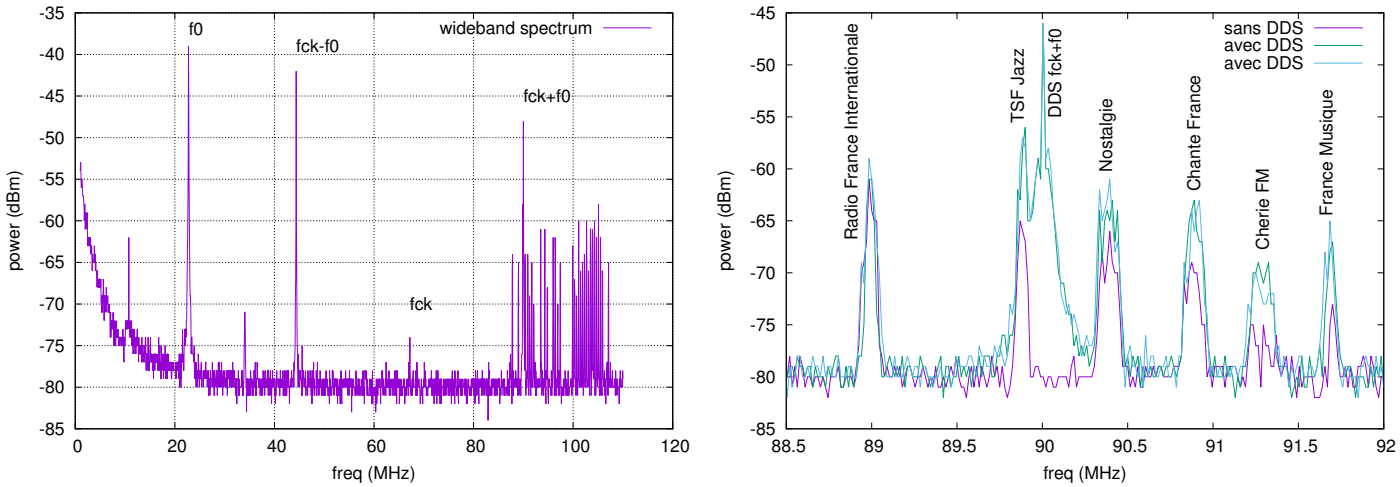


FIGURE 14 – Spectres large bande (gauche) et zoom sur la bande FM (droite) pour vérifier notre capacité à émettre une porteuse dans cette gamme de fréquences.

Nous avons donc la capacité à émettre sur une porteuse comprise dans la bande FM. Il reste désormais à reprogrammer périodiquement, par le bus SPI, la fréquence émise par le DDS pour représenter, sous forme de fréquence autour de la porteuse, le signal acquis sur un des ADCs. De cette façon, nous aurons réalisé un émetteur FM avec un unique composant en plus du microcontrôleur. Noter la flexibilité de la synthèse directe des signaux, au détriment de la propreté du signal émis (il faudrait en pratique filtrer la raie autour de 90 MHz pour éliminer les autres sources parasites de signaux), qui permet de générer tout signal dans la limite de la bande passante du SPI : illustration de la puissance de l'approche logicielle de l'émission radiofréquence (*software defined radio*).

7 Caractérisation d'un dispositif radiofréquence : quartz 8 MHz

Le circuit se comporte comme un pont diviseur de tension, avec le dispositif à tester présentant une

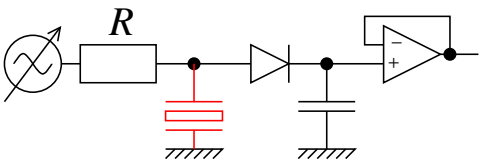


Figure 15: Principe de la mesure de puissance

faible impédance vers la masse à la résonance. Comme une diode présente une tension de seuil trop élevée – ajustable en polarisant par un faible courant DC – il semble judicieux de la remplacer par un composant dédié à cet effet, ici le Linear Technology LT5537. On notera que ce composant propose par ailleurs une fonction de mise en veille pour les applications alimentées sur batterie : nous prendrons soin d'activer le composant (placer ENBL au niveau haut par commande de GPIOC10 du microcontrôleur) pour effectuer la mesure.

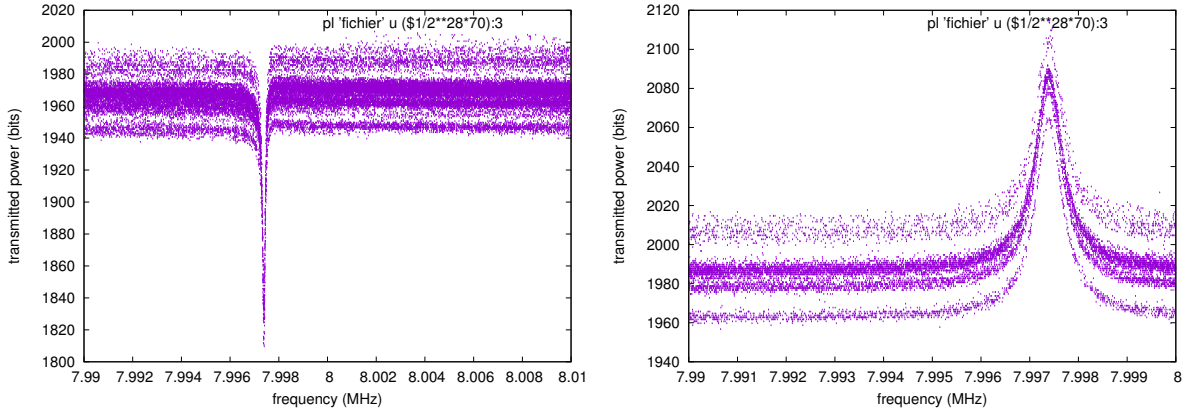


FIGURE 16 – Caractérisation, de la fonction de transfert d'un résonateur à quartz de fréquence nominale 8 MHz. Gauche : magnitude issue du circuit en pont diviseur. Droite : mesure en transmission. Noter la traduction du mot de fréquence en une valeur en hertz pour l'axe des abscisses. Les deux graphiques sont affichés avec `gnuplot`.

La Fig. 16 présente le résultat d'une telle mesure : l'écart de la fréquence de résonance aux 8 MHz nominaux reste à déterminer (décallage du résonateur, effet de température, ou décallage de l'oscillateur de référence). Pour une résistance du pont de 400 Ω, la chute de puissance observée à l'analyseur de spectre est de 1 dB dans le pont diviseur, et 5 dB en transmission. Ces mesures sont cohérentes avec la caractérisation au moyen d'un instrument de laboratoire dédié (Fig. 17).

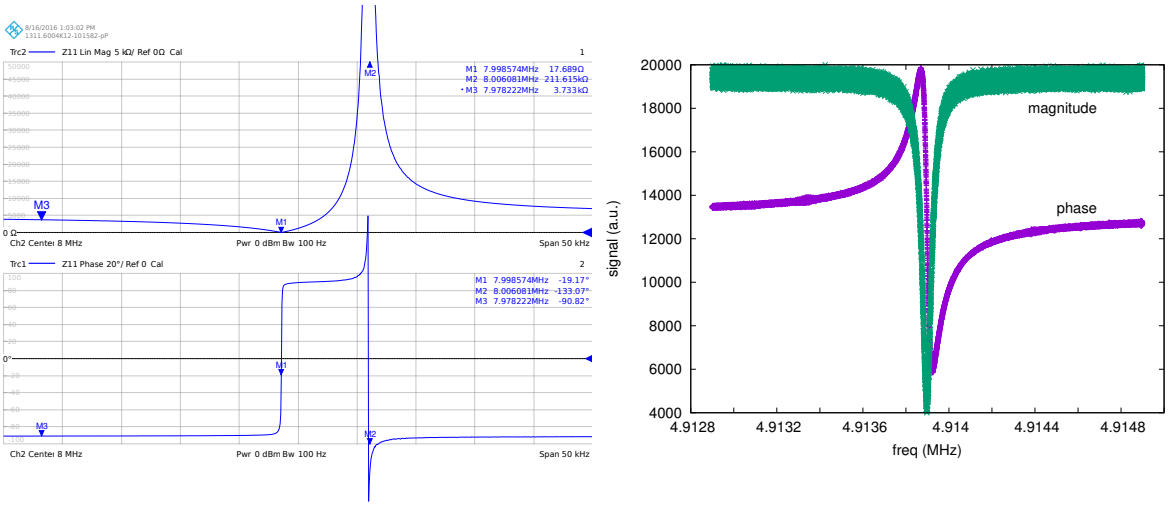


FIGURE 17 – Gauche : caractérisation du résonateur 8 MHz en réflexion à l'analyseur de réseau Rohde & Schwartz ZNC-3. Droite : caractérisation d'un quartz à 4,9 MHz en amplitude et en phase. Chaque mesure est le résultat de 16 moyennes.

8 Interfaces graphiques

Au lieu d'enregistrer les données acquises dans un fichier
`stty -F /dev/ttyUSB0 115200 && cat < /dev/ttyUSB0 > fichier`
il peut parfois être intéressant d'observer en temps réel le résultat de la mesure. Cette fonctionnalité est présentée ici sous Python, avec dans un premier temps la découpe des trames transmises par le microcontrôleur (nombres en hexadécimal précédés de 0x) et affichage en fin d'acquisition dans Matplotlib (Fig. 18). Cette bibliothèque ne permet pas au programme de continuer son exécution alors que l'affichage est en cours (peut être possible au moyen de plusieurs threads et un affichage non-bloquant, mais la portabilité du code résultant ne semble pas bonne) : la seconde solution proposée passe par `pyqtgraph`. Dans ce cas, l'ordonnanceur Qt est sollicité, et l'acquisition se fait dans un nouveau thread.

```
# voir https://gist.github.com/turbinenreiter/7898985
import time
import io
import thread
import serial   # http://www.lfd.uci.edu/~gohlke/pythonlibs/#pyserial
import matplotlib.pyplot as plt
import numpy as np

# open serial port
sprfd = serial.Serial(
#  port='COM1',
   port='/dev/ttyUSB0',
   baudrate=115200,
   parity=serial.PARITY_NONE,
   stopbits=serial.STOPBITS_ONE,
   bytesize=serial.EIGHTBITS,
   xonxoff =False,
   rtscts=False,
   dsrdtr=False,
   timeout=1      # 1 second timeout
)

# main program: initialize peripherals and run through a loop
x=[]
y=[]
k=1
sprfd.isOpen()
sio = io.TextIOWrapper(io.BufferedRWPair(sprfd, sprfd))
# fspr=open('spr_alone.dat', 'w')
# fspr.write("% time(s) frequency phase amplitude\r\n")
# init_time=time.time();
while k < 8192 :
   res=sio.readline()
   resplit=res.split(' ')           # split sentence: separator is ' '
   if len(resplit)>2 and res[0]=='0' :
#     current_time=round(time.time()-init_time,3); # time since beginning of experiment (in s)
#     print(resplit[0]+" / "+resplit[1]+" / "+resplit[2])
      freq=float(int(resplit[0],16))/2**28*70;
      phase=int(resplit[1],16);
      amplitude=int(resplit[2],16);
#     fspr.write(str(current_time)+" "+str(freq)+" "+str(amplitude)+" "+str(phase));
#     print str(freq)+" "+str(amplitude)
      x=np.append(x,freq)
      y=np.append(y,amplitude)
      k=k+1
      print(k)
plt.plot(x, y)
plt.xlabel('freq (MHz)')
plt.ylabel('magnitude (bits)')
plt.title('8 MHz resonator -- transmission mode')
plt.grid(True)
plt.draw()
plt.show()
sprfd.close()
plt.close('all')
```

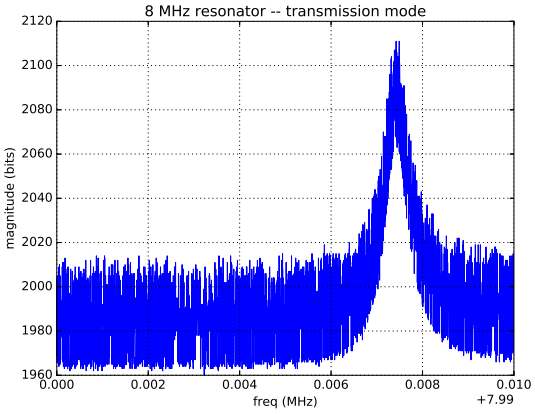


FIGURE 18 – Graphique généré par Matplotlib à l'issue de l'acquisition de 8192 points que nous savons couvrir la gamme de fréquences analyée. Comparer la sortie avec celle fournie en Fig. 16.

Ce second programme est inspiré du premier, mais fait appel à `pyqtgraph` pour un affichage en temps réel, déplace l'acquisition dans un thread séparé, et conclut son acquisition par le même affichage que précédemment dans Matplotlib pour comparaison.

```
# voir https://gist.github.com/turbinenreiter/7898985
import time
import io
import thread
import serial   # http://www.lfd.uci.edu/~gohlke/pythonlibs/#pyserial
import matplotlib.pyplot as plt
import numpy as np
import pyqtgraph as pg
from pyqtgraph.Qt import QtGui, QtCore

def measure_thread(k):
   global x,y
   while k < 8192 :
      res=sio.readline()
      resplit=res.split(' ')           # split sentence: separator is ' '
      if len(resplit)>2 and res[0]=='0' :
#        print(resplit[0]+" / "+resplit[1]+" / "+resplit[2])
         freq=float(int(resplit[0],16))/2**28*70;
         phase=int(resplit[1],16);
         amplitude=int(resplit[2],16);
#        print str(freq)+" "+str(amplitude)
# JMF : faut-il proteger par mutex x et y ?
# Benny : alternative aux mutex = pyqtSignal et pyqtSlot
         x=np.append(x,freq)
         y=np.append(y,amplitude)
         k=k+1
         print(k)
   plt.plot(x, y)
   plt.xlabel('freq (MHz)')
   plt.ylabel('magnitude (bits)')
   plt.title('8 MHz resonator -- transmission mode')
   plt.grid(True)
   plt.draw()
   plt.show()
   sprfd.close()
   plt.close('all')

# open serial port
sprfd = serial.Serial(
#  port='COM1',
   port='/dev/ttyUSB0',
   baudrate=115200,
   parity=serial.PARITY_NONE,
```

```
   stopbits=serial.STOPBITS_ONE,
   bytesize=serial.EIGHTBITS,
   xonxoff =False,
   rtscts=False,
   dsrdtr=False,
   timeout=1
)

def update():
   print "timer"
#  print x
   pw.plot(x, y, clear=True)

x=[]
y=[]
k=1
thread.start_new_thread(measure_thread,(k,))
sprfd.isOpen()
sio = io.TextIOWrapper(io.BufferedRWPair(sprfd, sprfd))
pw=pg.plot()
timer = pg.QtCore.QTimer()
timer.timeout.connect(update)
timer.start(2000)

if __name__ == '__main__':
   import sys
   if (sys.flags.interactive != 1) or not hasattr(QtCore, 'PYQT_VERSION'):
      QtGui.QApplication.instance().exec_()
```

Le gain en interactivité se fait au détriment de la portabilité, puisque nous sommes doucement en train de quitter le monde de Python pour programmer une application Qt.

9 Interfaçage sur Olinuxino A13-micro sous GNU/Linux

Communiquer par SPI sous GNU/Linux affranchit d'une compréhension détaillée des interfaces matérielles si un pilote existe pour ce périphérique. C'est le cas pour le SPI de microcontrôleur équipant la carte Olinuxino A13-micro. Pour ce faire, nous devons informer le système d'exploitation de la configuration des broches (Fig. 19) associées au bus SPI, ici SPI2 connecté au port UEXT de la carte.

Cette information se place dans le *devicetree* situé à `linux-4.4.2/arch/arm/boot/dts/sun5i-a13-olinuxino-micro` dans l'arborescence des sources du noyau. Nous y ajoutons un nouveau nœud décrivant les ressources occupées par le bus SPI :

```
&spi2 {
 pinctrl-names = "default";
 pinctrl-0 = <&spi2_pins_a>,
 <&spi2_cs0_pins_a>;
 status = "okay";
 spidev0: spidev@0{ compatible = "spidev";
   reg = <0>;  /* CE0 */
   #address-cells = <1>;
   #size-cells = <0>;
   spi-max-frequency = <500000>;
 }
};

spi2_pins_a: spi2@0 {
 allwinner,pins = "PE1", "PE2", "PE3";
 allwinner,function = "spi2";
 allwinner,drive = <SUN4I_PINCTRL_10_MA>;
 allwinner,pull = <SUN4I_PINCTRL_NO_PULL>;
};

spi2_cs0_pins_a: spi2_cs@0 {
 allwinner,pins = "PE0";
 allwinner,function = "spi2";
 allwinner,drive = <SUN4I_PINCTRL_10_MA>;
 allwinner,pull = <SUN4I_PINCTRL_NO_PULL>;
```

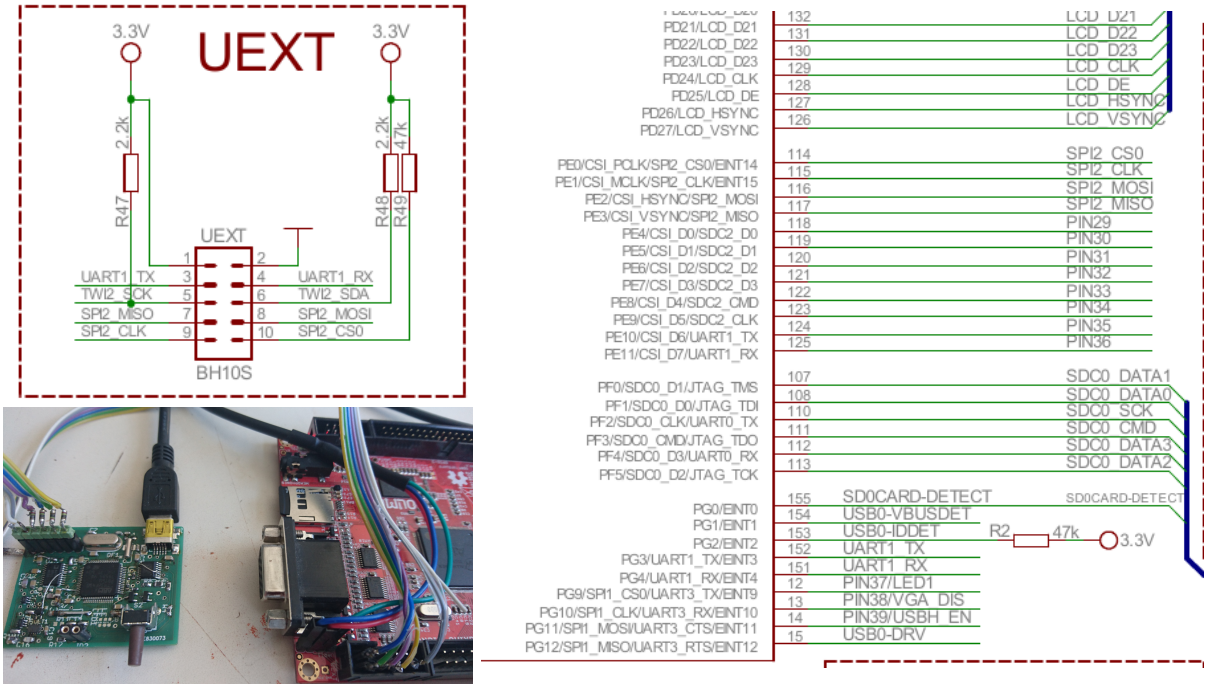


FIGURE 19 – Broches associées au bus SPI sur carte Olinuxino A13-micro (extrait de la documentation disponible à `https://www.olimex.com/Products/OLinuXino/A13/A13-OLinuXino-MICRO/resources/A13-OLINUXINO-MICRO.pdf`

```
};
```

Nous ajoutons par ailleurs le support au bus SPI et au point d'accès `spidev` dans la configuration du noyau, que nous recompilons (Fig. 20).

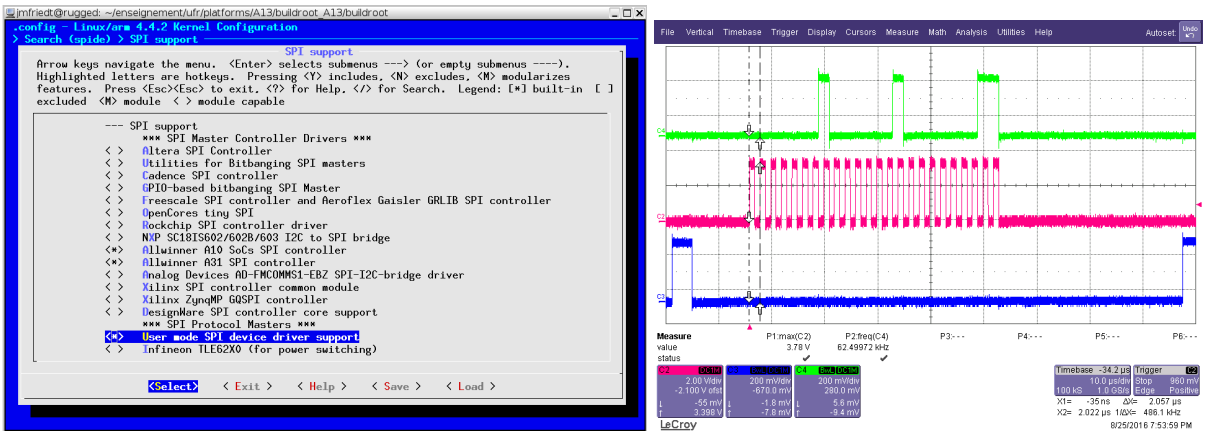


FIGURE 20 – Gauche : configuration SPI dans le noyau linux sous buildroot. Droite : observation à l'oscilloscope des signaux issus du bus SPI (en vert les données, en rouge l'horloge et en bleu le Chip Select).

Suite au boot de la carte munie de ce nouveau noyau, nous constatons l'apparition d'un nouveau point d'accès dans `/dev` nommé `spidev32766.0`. Nous y accédons depuis le *shell* pour envoyer des mots sur le bus SPI par

```
while true; do echo -n -e "\x01\x02\x03" > /dev/spidev32766.0;sleep1;done
```

Le DDS nécessite par ailleurs une broche pour commuter le signal de réinitialisation RESET. Nous avons accès aux GPIO au travers de l'interface appropriée de Linux :

```
echo "50" > /sys/class/gpio/export    # 50 = PB18
echo "out" > /sys/class/gpio/gpio50/direction
while true; do echo "1" > /sys/class/gpio/gpio50/value; sleep 0.1; \
      echo "0" > /sys/class/gpio/gpio50/value; sleep 0.2;done
```

Nous avons donc généré l'ensemble des signaux nécessaires à la commande du DDS depuis GNU/-Linux. Cependant, nous constatons que le protocole de communication attendu par le DDS n'est pas respecté : il faut que l'état au repos de l'horloge soit au niveau haut et que la donnée soit échantillonnée sur le front descendant de l'horloge. Ces deux paramètres sont définis par convention par les valeurs des paramètres CPHA et CPOL (Fig. 21). Soit ces valeurs sont définies dans le *devicetree* dans lequel il est possible de définir leur valeur par défaut, mais, plus simple, nous pouvons atteindre ces valeurs par `ioctl`. Cela nécessite de quitter le shell pour passer à une programmation en C.

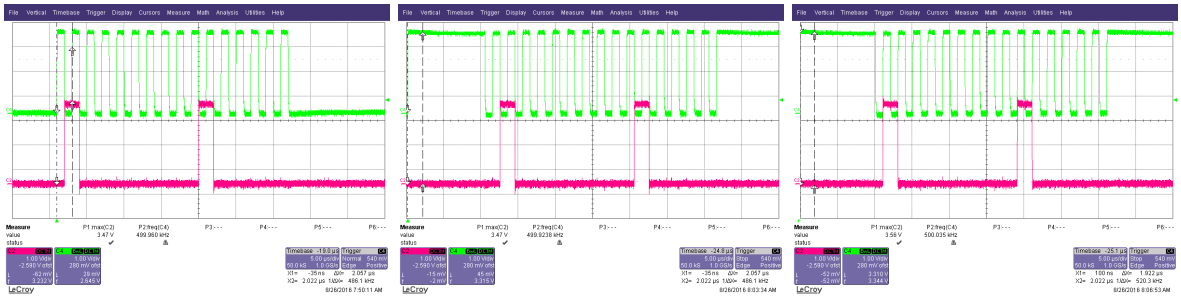


FIGURE 21 – Gauche : CPHA=0, CPOL=0. Milieu : CPHA=1, CPOL=1. Droite : la bonne configuration.

Il nous reste désormais à systématiser la programmation du DDS pour balayer la fréquence. Deux petites subtilités dans le code ci-dessous : dans un premier temps, les mots sont définis avec un ordre des octets (*endianness*) inversée par rapport aux attentes du DDS, d'où l'utilisation de la fonction `htonl` pour passer en *big endian*, et ensuite la nécessité d'envoyer en premier le mot de poids faible.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>           // write, close
4  #include <fcntl.h>            // open, O_RDWR
5  #include <sys/ioctl.h>        // ioctl
6  #include <linux/spi/spidev.h> // ioctl de spidev
7  #include <arpa/inet.h>        // hton
8
9  int main(int argc, char** argv)
10 {unsigned short mode=0;
11  unsigned long freq;
12  long ff;
13  int f,ret=0;
14  //mode |= SPI_CPHA; // 1
15  mode |= SPI_CPOL; // 2  falling edge, idle clock high
16  f=open("/dev/spidev32766.0",O_RDWR);
17  if (f<0) printf("error\n");
18  else
19   {ret=ioctl(f,SPI_IOC_WR_MODE,&mode);
20    if (ret==-1) printf("failure\n");
21    mode=0x2020;
22    write(f,&mode,2);
23    if (argc>1) freq=strtol(argv[1], NULL, 16); else freq=(0x41204020); // mot en hexa, 0x4XXX4XXX
24    ff=((freq&0x3fff0000)>>2)+(freq&0x00003fff);      // retire les bits definissant le reg. f0
25    printf("0x%lx=%lld␣Hz\n",ff,((long long)ff*(long long)70000000)/(long long)(1<<28)); // fref=70 MHz
26    freq=htonl(freq);                                 // il faut se placer dans la bonne endianness
27    freq=((freq&0xffff)<<16)+((freq&0xffff0000)>>16); // puis envoyer d'abord le mot de poids faible !
28    while (1) write(f,&freq,4);
29    close(f);
30   }
31  return(0);
32 }
```

Le résultat (Fig. 22) est conforme à nos attentes.

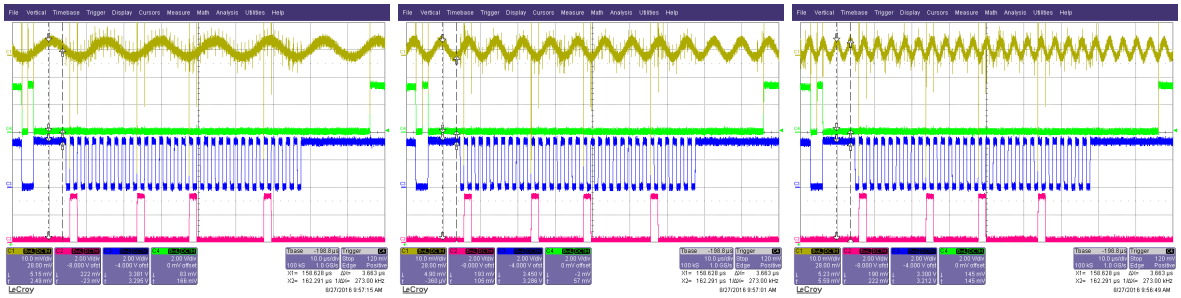


FIGURE 22 – Mot de programmation : 0x40104010 à gauche, 0x40204020 au milieu, 0x40404040 à droite.

10 Interfaçage sur FPGA

Le DDS est interfacé à un FPGA en plaçant les broches du microcontrôleur en entrée (voir programme ci-dessous). Les résistances de protection en série entre les signaux issus du FPGA (5 V) pour le DDS (3,3 V) sont de l'ordre de 200 Ω, afin de limiter l'effet capacitif sur les fronts des signaux numériques tout en protégeant les broches numériques contre les sur-tensions. Seuls le signal de veille (SLEEP) et l'horloge maîtresse du DDS (MCLOCK) sont fournis par le microcontrôleur. Le FPGA fournit les signaux de réinitialisation (interrupteur activé manuellement), FSYNC et les signaux du bus SPI.

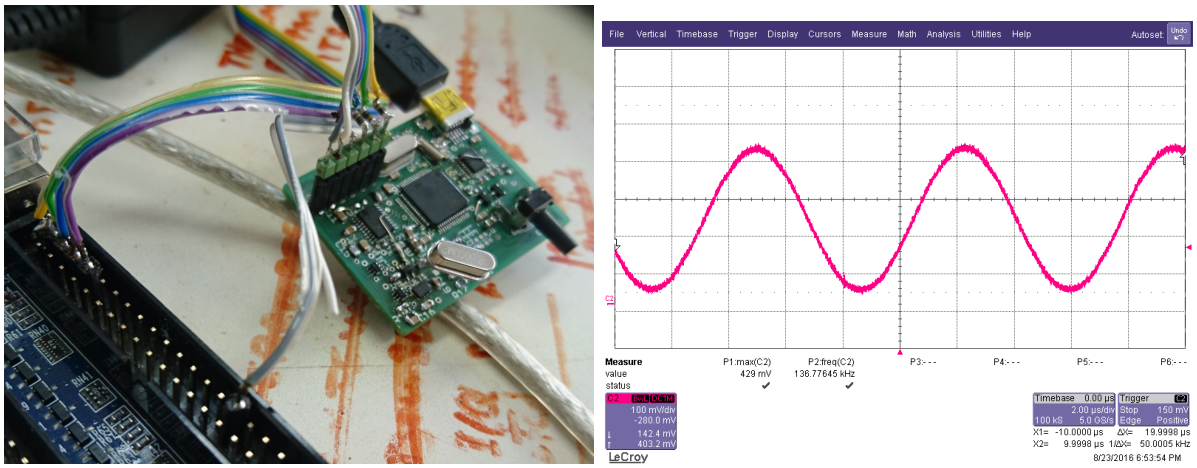


FIGURE 23 – Gauche : montage expérimental. Droite : signal de sortie du DDS lorsque le mot programmé est 0x080020, soit une fréquence de $\frac{0x080020}{2^{28}} \cdot 70$ MHz=136,73 kHz.

Les mots programmés dans le DDS sont configurés au moyen des interrupteurs : le DDS est initialisé par le mot 0x2028. Suit le mot de programmation du DDS 0x6020 répété deux fois pour définir une fréquence de l'ordre de 35 MHz, complexe à analyser compte tenu de la fréquence de cadencement du DDS de 70 MHz (somme de deux composantes spectrales proches autour de 35 MHz induisant un battement de quelques kHz). Placer le bit du mot de fréquence de poids le plus fort à 0 se traduit par la programmation du mot 0x080020 (résultat de 0x4020 transmis deux fois), ou une fréqeunce 136,73 kHz. La visualisation sur oscilloscope valide la programmation du DDS par le FPGA (Fig. 23). Pour le moment le mot est défini par des interrupteurs (Fig. 24 : il serait envisageant de définir la séquence de valeurs transmises par SPI depuis le FPGA, par exemple issues du DDS conçu en VHDL.

```
1  #include <libopencm3/stm32/rcc.h>
2  #include <libopencm3/stm32/gpio.h>
3  #include <libopencm3/stm32/timer.h>
4  #include <libopencm3/stm32/flash.h> // definitions du timer
5
6  static void clock_setup(void);
7  static void init_gpio(void);
8  void init_time1 (void);
9  void dds_slp_clr(void);
10 void dds_slp_set(void);
```

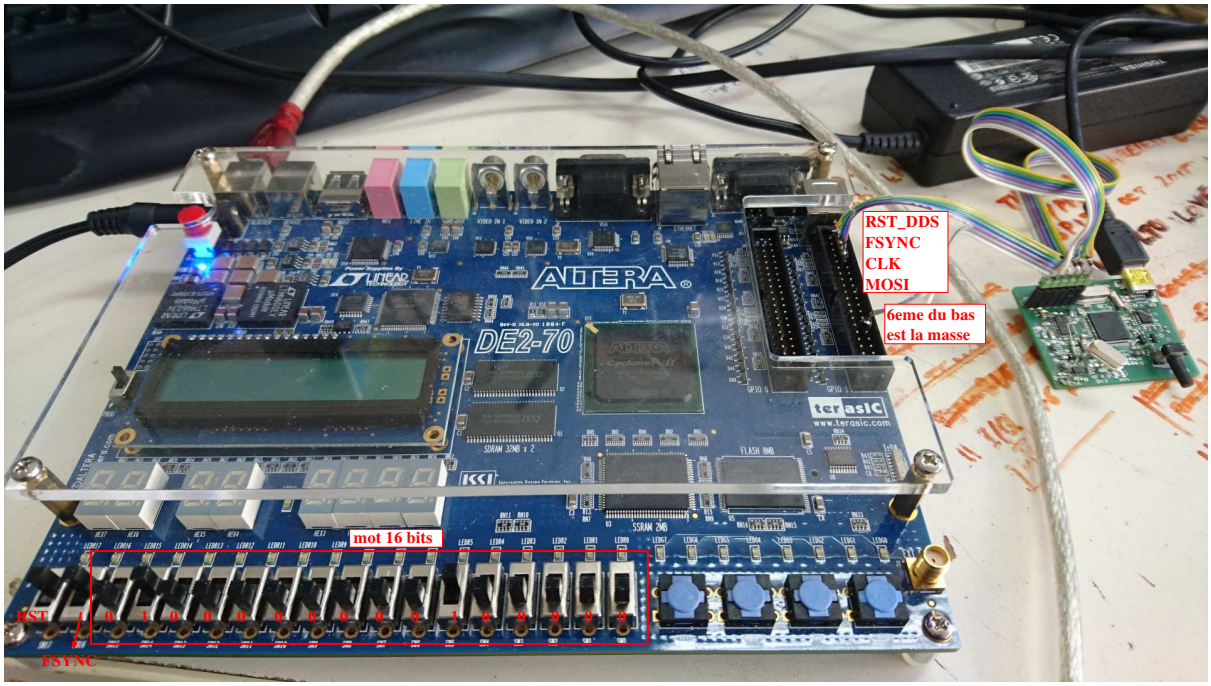


FIGURE 24 – Définitions des interfaces d'entrée du FPGA.

```
11 void det_clr(void);
12 void det_set(void);
13
14 // APB2 max=84 MHz but when the APB prescaler is NOT 1, the interface clock is fed
15 // twice the frequency => Sysclk = 140 MHz, APB2=2 but Timers are driven at twice that is 140.
16 const struct rcc_clock_scale rcc_hse_20mhz_3v3 = {
17                 .pllm = 20,                             // 20/20=1 MHz
18                 .plln = 280,                            // 1*280/2=140 MHz
19                 .pllp = 2,                              //       ^
20                 .pllq = 6,
21                 .hpre = RCC_CFGR_HPRE_DIV_NONE,
22                 .ppre1 = RCC_CFGR_PPRE_DIV_4,
23                 .ppre2 = RCC_CFGR_PPRE_DIV_2,
24                 .flash_config = FLASH_ACR_ICE | FLASH_ACR_DCE |
25                                 FLASH_ACR_LATENCY_4WS, // 4 WS d'apres configuration par ST
26                 .ahb_frequency  = 140000000,
27                 .apb1_frequency = 35000000,
28                 .apb2_frequency = 70000000,
29         };
30
31 static void clock_setup(void)
32 {// rcc_clock_setup_hse_3v3(&rcc_hse_20mhz_3v3); // custom version
33  rcc_clock_setup_pll(&rcc_hse_20mhz_3v3);
34  rcc_periph_clock_enable(RCC_TIM1);
35  rcc_periph_clock_enable(RCC_GPIOA); // Enable GPIOA clock.
36  rcc_periph_clock_enable(RCC_GPIOC); // Enable GPIOC clock.
37  rcc_periph_clock_enable(RCC_ADC1);  //clock ADC
38 }
39
40 static void init_gpio(void)
41 {gpio_mode_setup(GPIOA, GPIO_MODE_INPUT,GPIO_PUPD_NONE, GPIO4 | GPIO5 | GPIO7 | GPIO15 ); // MOSI, CK, CS, DDS RST
42  gpio_mode_setup(GPIOA, GPIO_MODE_OUTPUT,GPIO_PUPD_NONE,GPIO14); // DDS sleep
43  gpio_mode_setup(GPIOC, GPIO_MODE_OUTPUT,GPIO_PUPD_NONE, GPIO10); // Enable DET
44 }
45
46 void init_time1 (void)
47 {gpio_set_output_options(GPIOA, GPIO_OTYPE_PP, GPIO_OSPEED_50MHZ, GPIO8 );
48  gpio_mode_setup (GPIOA, GPIO_MODE_AF, GPIO_PUPD_NONE, GPIO8);
49  gpio_set_af (GPIOA, GPIO_AF1, GPIO8);
```

```
 // timer_reset (TIM1); // modification du 180826 /w new libopencm3 version
 rcc_periph_reset_pulse(RST_TIM1); // voir https://github.com/libopencm3/libopencm3-examples/commits/master/examples/stm32/f
 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_OC1, TIM_OCM_PWM2);
  timer_enable_oc_output(TIM1, TIM_OC1);
  timer_enable_break_main_output(TIM1);
  timer_set_oc_value(TIM1, TIM_OC1, 1); // 1

  timer_set_prescaler (TIM1, 0);
  timer_set_period (TIM1, 1);  // pourquoi 1 et pas 2 ? il FAUT que period >= OC1
  timer_enable_counter (TIM1);
}

void dds_slp_clr (){gpio_clear(GPIOA, GPIO14);}
void det_set     (){gpio_set  (GPIOC, GPIO10);}

int main(void)
{
 clock_setup();
 init_gpio();
 init_time1();
 det_set();
 dds_slp_clr ();

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

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ENTITY spi_send_switch IS
PORT(
  iSW : IN std_logic_vector(15 downto 0);
  go : IN std_logic;
--  ena : buffer std_logic;
  iclk_50 : IN std_logic;
  iFSYNC : IN std_logic;
  oFSYNC: OUT std_logic;
  iclr : IN std_logic;
  oclr : OUT std_logic;
  MOSI : OUT std_logic;
  SCLK : OUT std_logic
);
END ENTITY;

ARCHITECTURE archi of spi_send_switch IS

 COMPONENT spi_send IS
  GENERIC(
   N : integer := 16; -- taille des mots
   tscale : integer := 10 -- division de l'horloge (mini 2)
  );
  PORT (
   h : IN std_logic;
   ena : IN std_logic  :='1';
   data: IN std_logic_vector (N-1 downto 0) := ( N-2 downto N-3 => '1', others =>'0');
   finished : buffer std_logic; -- 1= fini, 0= en cours
   MOSI : OUT std_logic;
   SCLK : OUT std_logic
  );
 END COMPONENT;

 SIGNAL finished : std_logic;
 SIGNAL ena, oFSYNC_tmp : std_logic;
 SIGNAL go_reg : std_logic_vector (5 downto 0);
```

```
 signal etat : integer range 0 to 4:=0; -- 0 = repos; FSYN=0, en = 0 (attend appui sur go)
             -- 1 = appui sur go : oFYNC=0 (en=0) pdt 1 coup d'horloge
             -- 2 = coup horloge suivant en =1, check finished passe bien 'a 0 (1 coup d'horloge normalement) pour aller 'a
             -- 3 = envoi de la transmission (FSYN=0, en=1), attend finished = 1
             -- 4 = en=0, FSYNC encore 'a 0; return 'a etat 0 au coup d'horloge suivant

BEGIN
 PROCESS
 BEGIN
  wait until rising_edge(iclk_50);
--  oFSYNC <= iFSYNC;
  oFSYNC <= oFSYNC_tmp;
  oclr <= iclr;
  go_reg <= go_reg(4 downto 0) & go;

  CASE etat IS
   WHEN 0 => IF go_reg = "111000" THEN etat <= 1;
       ELSE etat <= 0;
       END IF;
   WHEN 1 => etat <= 2;
   WHEN 2 => IF finished = '0' then etat <= 3; ELSE etat <= 2; END IF;
   WHEN 3 => IF finished = '1' then etat <= 4; ELSE etat <= 3; END IF;
   WHEN 4 => etat <= 0;
   WHEN OTHERS => etat <= 0;
  END CASE;
 END PROCESS;

 ena <= '1' WHEN (etat = 2) OR (etat = 3) ELSE '0';
 oFSYNC_tmp <= '0' WHEN (etat = 1) OR (etat = 2) OR (etat = 3) OR (etat = 4) ELSE '1';

 SPI : spi_send
 GENERIC MAP
 (
  N => 16,
  tscale => 10
 )
 port map
 (
   h => iclk_50,
   ena => ena,
   data => iSW,
   finished => finished,
   MOSI => MOSI,
   SCLK => SCLK
 );
END archi;
```

```
-- envoi simple de 16 bits par SPI pour commande du DDS
-- SS/ (FSYNC) non g'er'e
-- SE aout 2016

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ENTITY spi_send IS
 GENERIC(
  N : integer := 16; -- taille des mots
  tscale : integer := 10 -- division de l'horloge (mini 2)
 );
 PORT (
  h : IN std_logic;
  ena : IN std_logic  :='1';
  data: IN std_logic_vector (N-1 downto 0) := ( N-2 downto N-3 => '1', others =>'0');
  finished : buffer std_logic; -- 1= fini, 0= en cours
  MOSI : OUT std_logic;
  SCLK : OUT std_logic
 );
END spi_send;

```

```
ARCHITECTURE archi OF spi_send IS
 SIGNAL t_compt : integer range 0 to tscale -1 :=0;
 SIGNAL n_bit : integer range 0 to N := N-1; -- N = fini
BEGIN
 PROCESS
  VARIABLE v_n_bit : integer range 0 to N :=N-1;
 BEGIN
  wait until rising_edge(h);
  if ena = '1'
  then
   v_n_bit := n_bit;

   if t_compt < (tscale -1)/2 then
    t_compt <= t_compt +1;
    SCLK <= '1';
   elsif (t_compt < (tscale -1)) and (n_bit < N) then
    t_compt <= t_compt +1;
    SCLK <= '0';
   else
    t_compt <= 0;
    SCLK <= '1';
    if v_n_bit = 0 then
     v_n_bit := N;
    elsif v_n_bit < N then
     v_n_bit := n_bit-1;
    end if;
   end if;

   n_bit <= v_n_bit;

   if (v_n_bit < N) then
    MOSI <= data(v_n_bit);
    finished <= '0';
   else MOSI <= '0';
    finished <= '1';
   end if;
  else
   t_compt <=0;
   n_bit <= N-1;
   SCLK <='1';
   MOSI <='0';
   finished <= '1';
  end if;
 end process;
end archi;
```