Le microcontrôleur STM32 : un cœur ARM Cortex-M3

G. Goavec-Mérou, J.-M Friedt

28 février 2012

Au sein de la gamme des cœurs de processeurs proposés par ARM, le Cortex-M3, opérant sur des registres de 32 bits, fournit un compromis entre une puissance de calcul appréciable et une consommation réduite qui, sans atteindre les performances du MSP430 (16 bits), propose néanmoins des modes de veille en vue de réduire la consommation moyenne d'une application. Bien que les nombreux périphériques disponibles ainsi que l'énorme quantité de mémoire associée à ce processeur puissent justifier l'utilisation de bibliothèques dont une implémentation libre est disponible sous le nom de `libopencm3`, nous verrons qu'il est possible d'appréhender un certain nombre de ces périphériques pour en faire un usage optimal en accédant directement aux registres qui en contrôlent l'accès. Le cœur M3 est décliné par de nombreux fondeurs : nous nous focaliserons ici sur l'implémentation de ST Microelectronics sous le nom de STM32F1 (dans la suite, le microcontrôleur sera nommé simplement STM32 car la plupart des applications sont portables sur les autres modèles).

1 Introduction

La famille des microprocesseurs STM32 de ST Microelectronics fournit une vaste gamme de périphériques autour d'un cœur d'ARM Cortex-M3 [CortexM3], allant du simple GPIO (port d'entrée-sortie généraliste) et interface de communication série synchrone (SPI) ou asynchrone (RS232) aux interfaces aussi complexes que l'USB, ethernet ou HDMI. Un point remarquable est qu'un certain nombre de ces processeurs possèdent deux convertisseurs analogique-numériques, permettant un échantillonnage *simultané* de deux grandeurs analogiques. Cadencé sur un résonateur interne ou sur un quartz externe haute fréquence 8 MHz (multiplié en interne au maximum à 72 MHz), ce processeur est compatible pour des applications faibles consommations (section 9) avec un mode veille dont le réveil s'obtient par une condition sur une horloge interne ou une interruption externe. La multiplicité des horloges et leur utilisation pour cadencer les divers périphériques est d'ailleurs un des aspects les plus déroutant dans la prise en main du STM32.

Notre choix d'investissement de temps sur ce microcontrôleur en particulier est dirigé par quelques contraintes techniques :

- avant tout, un double convertisseur analogique-numérique rapide (1 Méchantillons/s) sensé garantir la simultanéité des conversions sur deux voies, un point clé si du traitement numérique additionel est effectué sur une combinaison des deux voies,
- une architecture ARM Cortex-M3 exploitée par d'autres constructeurs : nous ne nous enfermons pas sur une architecture supportée par un unique fondeur, les principes de base concernant le cœur du processeur et la toolchain peuvent être réutilisés ultérieurement sur un autre processeur basée sur la même architecture (par exemple Atmel SAM3),
- un mode veille proposant une consommation raisonnable pour les application embarquées autonomes qui nous intéressent.

2 Architecture du processeur – implémentation d'un circuit

Le lecteur désireux de simplement exploiter un circuit commercialement disponible devrait pouvoir travailler sur le circuit STM32H103 de Olimex [1].

1. http://www.olimex.com/dev/stm32-h103.html

Pour notre part, nous nous proposons d'exploiter un circuit dédié, spécifiquement développé en vue d'émuler le port parallèle d'un PC, fournissant ainsi accès à la majorité des signaux utiles mais surtout illustrant le peu de composants annexes nécessaires au fonctionnement de ce microcontrôleur : un quartz 32,768 kHz, éventuellement un quartz 8 MHz, et des condensateurs de découplage aux 4 coins pour filtrer l'alimentation (Figs. 1 et 2). Ce circuit comporte par ailleurs un convertisseur USB-RS232 pour fournir une interface communément disponible sur tout PC récent. Nous verrons deux applications de cette carte, dans un cas pour émuler le comportement du port parallèle d'un PC pour le contrôle d'un instrument (section 5), et, plus ambitieux, la réalisation d'une station de mesure météorologique (section 8).

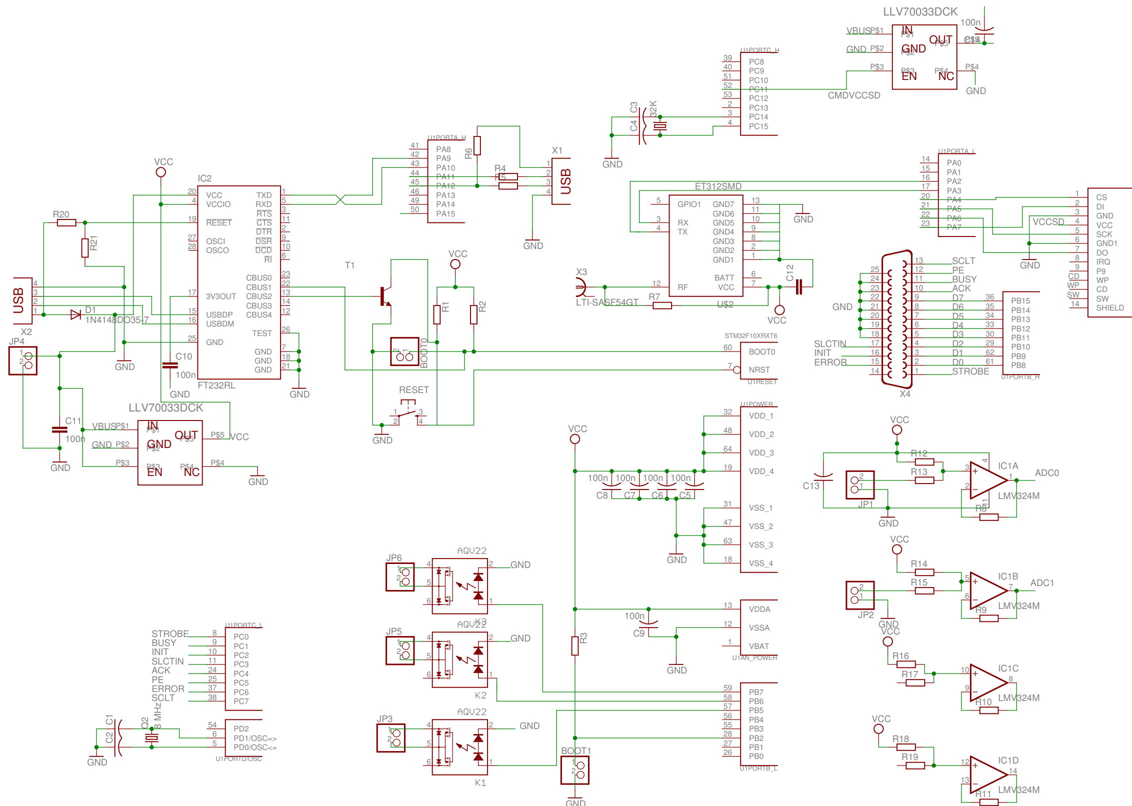

FIGURE 1 – Exemple de circuit exploitant le STM32 (schéma). Le quadruple-amplificateur opérationnel (en bas à droite) protège 4 voies de conversion analogique-numérique et ajoute une tension constante permettre la mesure de signaux de valeur moyenne nulle. Une carte SD est connectée sur bus synchrone SPI (en bas à gauche). Un régulateur linéaire (en haut à droite) permet l'alimentation de ce circuit sur le bus USB qui sert aussi à la communication au travers d'un convertisseur RS232-USB FT232.

3 Chaîne de compilation et bibliothèques

La chaîne de compilation est basée sur l'habituelle génération des binaires issus de `binutils` et `gcc` sur architecture x86 à destination du processeur ARM, et en particulier Cortex M3. Un script à peu près parfait est disponible sous la nomenclature `summon-arm-toolchain` – accessible par `git` au moyen de `git clone git://github.com/esden/summon-arm-toolchain.git` – pour aller rechercher l'ensemble des archives sur les sites appropriés et compiler les outils nécessaires à générer un binaire à destination du STM32. On notera que cette même *toolchain* est fonctionnelle

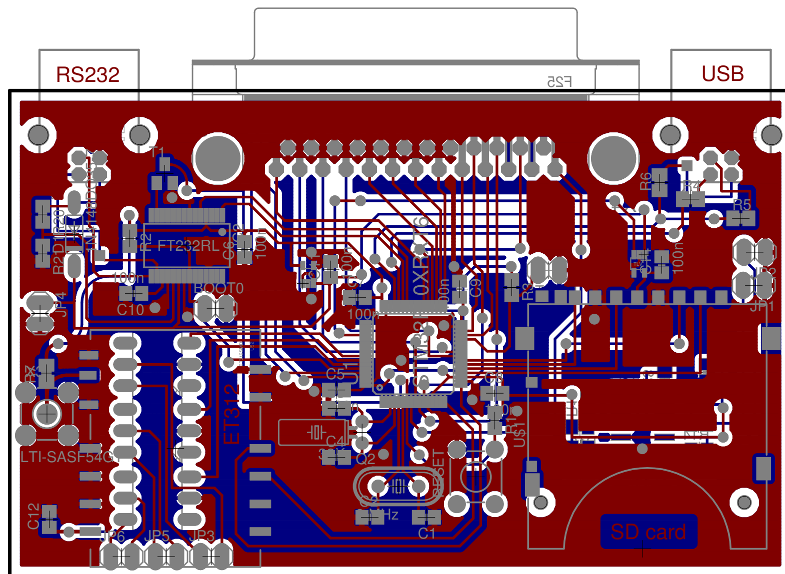

FIGURE 2 – Exemple de circuit exploitant le STM32 (implantation des composants).

pour d'autres architectures ARM, notamment l'ARM7 fourni dans l'ADuC7026 de Analog Devices déjà présenté auparavant [LM117].

La compilation se fait classiquement à l'aide de la commande suivante :

```
1 cd summon-arm-toolchain
./summon-arm-toolchain USE_LINARO=0 OOCD_EN=0
```

Celle-ci permet d'installer la toolchain avec la version Vanilla de GCC au lieu de linaro GCC (`USE_LINARO=0`), sans la compilation et l'installation de OpenOCD (`OOCD_EN=0`), disponible sous forme de paquet binaire dans toutes les distributions. Par défaut, seule la bibliothèque libre `libopencm3` est installée. Pour installer une bibliothèque propriétaire mais gratuite développée par ST, `libstm32`, l'option LIBSTM32_EN=1 devra être passée à la ligne de commande. Nous exploitons en particulier cette fonction pour compiler les nombreux exemples disponibles sur le web, en décortiquer le fonctionnement et ainsi accéder aux mêmes fonctionnalités en enrichissant `libopencm3`. Par défaut, les outils seront installés dans `$HOME/sat`.

Obtenir un compilateur fonctionnel ne constitue que le début de l'aventure du développement sur processeur ARM Cortex-M3. Ce cœur de processeur 32 bits est en effet supporté par divers fondeurs de circuits intégrés, et une tentative d'unification du support logiciel en vue de la portabilité du code d'un fondeur à l'autre est proposée sous la nomenclature CMSIS. Comme souvent dans cette thématique, l'intention est surement noble, mais le résultat consiste en une bibliothèque à la license peu rassurante (ST) exploitant abusivement des structures de données lourdes dont l'utilisation sur un système embarqué est un peu surprenante. D'un autre côté, une architecture 32 bits documentée sur plus de 1000 pages [RM0008] est difficile à appréhender par la lecture d'une datasheet décrivant la fonction de chaque registre [2] : un compromis appréciable en terme de license d'utilisation, complexité et proximité au matériel semble être le projet `libopencm3` (libopencm3.org). Accompagnée de nombreux exemples concrets – et en particulier sur le point épineux de l'exploitation du périphérique USB – cette bibliothèque est facile à appréhender malgré un manque de maturité certain et quelques périphériques absent qui seront sans doute comblés rapidement.

Une fois la compilation de toolchain finie, nous disposons dans `$HOME/sat` des outils nécessaires à la compilation d'applications : nous ajoutons le répertoire `$HOME/sat/bin` dans le `PATH`,

```
export BASE_DIR=/home/user/sat
```

2. à la fin, on revient toujours aux fondamentaux, mais l'approche est un peu rude en premier abord

```
2 export PATH=${PATH}:${CM3_BASE_DIR}/bin
  export CM_INC=${CM3_BASE_DIR}/arm-none-eabi/include
4 export CM_LIB=${CM3_BASE_DIR}/arm-none-eabi/lib
```

La ligne relative au répertoire `include` et `lib` servira lors de la compilation d'applications telle que présentée plus loin pour accéder aux bons répertoires lors de la génération du binaire.

4 Outils de programmation

Le second prérequis, après l'obtention d'une chaîne de compilation fonctionnelle, concerne l'outil pour programmer le microcontrôleur. Deux solutions sont possibles :

1. la programmation par RS232 avec un outil tel que `stm32flash` (code.google.com/p/stm32flash/).
2. la programmation par JTAG grâce à OpenOCD et à une sonde.

4.1 stm32flash

La récupération des sources de `stm32flash` se fait par :

```
svn checkout http://stm32flash.googlecode.com/svn/trunk/ stm32flash-read-only
```

Un simple make && sudo make install dans le répertoire compilera et installera l'outil.

`stm32flash` prend en argument le nom du fichier contenant l'image binaire à placer en mémoire du microcontrôleur (fichier .bin ou .hex), la commande à effectuer (lecture, écriture, vérification) et l'interface de communication. Accessoirement, le débit de communication peut être ajouté

Afin de passer le STM32 en mode programmation il faut, comme pour bon nombre d'autres microcontrôleurs, manipuler deux broches. La première est BOOT0 qui doit être mise à la tension d'alimentation VCC. Cette commande est validée par une mise à GND de la broche *reset* du STM32. À ce moment le microcontrôleur est prêt à être programmé avec la commande :

```
stm32flash -w main.bin /dev/ttyUSB0 -g 0x0
```

Le `-w` signifie que le fichier main.bin doit être écrit en flash, le `-g 0x0` spécifie l'adresse où commencera l'exécution (0x0 correspond au début de la flash). Nous proposons une modification du module `ftdi_sio.ko` permettant de manipuler deux broches du composant servant de convertisseur USB-RS232 (FT232RL) et de `stm32flash` pour manipuler ces deux signaux lors de la programmation [3]. Ces manipulations nécessitent néanmoins un transistor monté en circuit inverseur pour palier à l'état par défaut (incorrect) des broches du FT232RL, et d'un circuit de gestion du reset, Analog Devices ADM809, pour retarder l'impulsion d'initialisation lors de la mise sous tension et ainsi permettre au FT232RL de stabiliser l'état de ses sorties avant que le microcontrôleur ne tente d'exécuter son application (Fig. 3). Une présentation plus détaillée des possibilités de "détournement" des FTDIs fera l'objet d'un futur article.

4.2 OpenOCD

L'utilisation du JTAG pour la programmation d'un microcontrôleur présente de nombreux intérêts. Sans rentrer dans les détails, il permet de charger le binaire bien plus rapidement qu'avec une liaison série et de debugger le code (à l'aide de `GDB` connecté à OpenOCD) .

L'incantation pour programmer un STM32 à l'aide d'une sonde JTAG et de OpenOCD est

```
1 openocd -f interface/dp_busblaster.cfg \
          -f board/olimex_stm32_h107.cfg \
3         -c "adapter_khz 2000" \
          -c "init" \
5         -c "halt" \
          -c "stm32f1x mass_erase 0" \
7         -c "flash write_image main.hex" -c "reset run" -c "shutdown"
```

L'interface (la sonde JTAG) dépend bien entendu du matériel disponible. Bien que notre carte ne soit pas une Olimex, l'utilisation de ce fichier de configuration permet de fixer certaines informa-

3. patchs disponibles sur la page http://www.trabucayre.com/page-ftdi.html

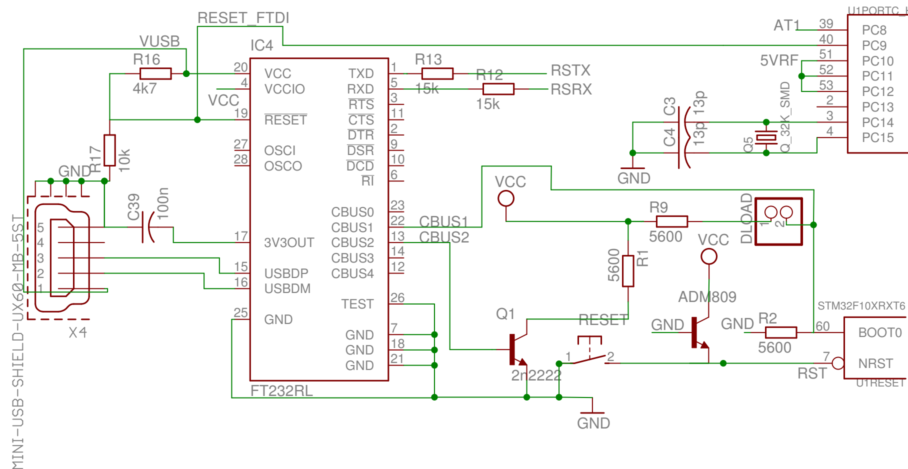

FIGURE 3 – Circuit de réinitialisation du STM32 : le jumper DLOAD et bouton poussoir RESET permettent de manipuler à la main ces fonctions pour passer le microcontrôleur en mode programmation, tandis que les broches CBUS1 et CBUS2 sont exploitées pour automatiser ces fonctionnalités en commandant la commutation des signaux depuis le PC. À ces fins, une version modifiée de `ftdi_sio.ko` et `stm32flash` sont nécessaires. Le transistor Q1 sert d'inverseur pour que la valeur par défaut de CBUS2 (lors de l'exécution d'un programme par la carte alimentée sur USB) ne réinitialise pas le microcontrôleur.

tions plus confortablement. Bien entendu là encore la configuration devra être adaptée à la version du STM32 utilisé [4].

4.3 Compilation de programme

La compilation des fichiers .c ainsi que la génération du fichier au format ELF se fait à l'aide des commandes suivantes :

```
arm-none-eabi-gcc -O0 -g3 -DSTM32F1 -Wall -Wextra \
-I${CM_INC} -I${CM_INC}/libopencm3/stm32 -I${CM_INC}/libopencm3/stm32/f1 \
-fno-common -mthumb -msoft-float -mcpu=cortex-m3 -o main.o -c main.c

arm-none-eabi-gcc -o main.elf main.o -lopencm3_stm32f1 \
-L${CM_LIB}/thumb2 -lc -lnosys -L${CM_LIB} -L${CM_LIB}/lib/stm32 \
-Tlibopencm3_stm32.ld -nostartfiles
```

Nous y activons quelques warnings pour la compilation, ajoutons les en-têtes et la `libopencm3_stm32f1.a`. Nous en profitons pour ajouter également la `libc` qui va permettre d'utiliser des fonctions telles que `sprintf()`.

Selon que `openOCD` ou `stm32flash` soit utilisé, il faudra générer un fichier au format hexadécimal ou binaire.

```
arm-none-eabi-objcopy -Oihex main.elf main.hex
arm-none-eabi-objcopy -Obinary main.elf main.bin
```

5 Premier exemple : faire clignoter une LED

Pour ne pas faillir à la régle, la découverte du STM32 va se faire en pilotant une broche.

Quelle que soit l'application, le STM32 par défaut est cadencé par son oscillateur interne, peu précis en fréquence : nous allons donc passer sur le quartz 8 MHz, nous autorisant ainsi à cadencer le microcontrôleur à la vitesse de 72 MHz (au maximum) par multiplication interne. Cadencer le STM32 à 72MHz n'est pas une obligation. La réduction de la vitesse des horloges (utilisation d'un

4. dans `/usr/share/openocd/scripts` et ses sous répertoires (interface pour les sondes, board pour les cartes).

plus faible multiplieur pour la PLL) est une solution pour baisser légérement la consommation globale du STM32.

```
rcc_clock_setup_in_hse_8mhz_out_72mhz();
```

Cette fonction, d'apparence simple, cache de nombreuses d'opérations telles que la configuration de la PLL, des horloges APB1 et APB2 utilisées pour les périphériques du STM32, le basculement sur quartz `HSE` (quartz externe haute-fréquence) et l'attente de la fin du calibrage de ce dernier.

Les ports d'entrée-sortie (*General Purpose Input Output*, GPIO) du STM32 sont synchrones : nous avons besoin d'activer l'horloge pour le(s) port(s) que nous allons utiliser. Dans notre cas la broche est PC7, ainsi il faut en premier lieu activer ce port :

```
1 /* Enable GPIOC clock. */
  rcc_peripheral_enable_clock(&RCC_APB2ENR, RCC_APB2ENR_IOPCEN);
```

puis configurer la GPIO en sortie et la mettre à l'état haut :

```
  /* Setup GPIO6 and 7 (in GPIO port C) for led use. */
2 gpio_set_mode(GPIOC, GPIO_MODE_OUTPUT_50_MHZ,
        GPIO_CNF_OUTPUT_PUSHPULL, GPIO7);
4 gpio_set(GPIOC, GPIO7);
```

Pour finir nous entrons dans une boucle infinie dans laquelle nous mettons la GPIO à l'état haut, puis après une attente (itération sur la mnémonique assembleur `nop` en boucle), nous la passons à l'état bas :

```
  while (1) {
2    gpio_set(GPIOC, GPIO7);
     Delay(0xaffff);
4    gpio_clear(GPIOC, GPIO7);
     Delay(0xaffff);
6 }
```

En quelques lignes de code nous avons ainsi pu configurer la vitesse du STM32, la nature de la source d'horloge, et avons pu manipuler une GPIO. Une application concrète de ces concepts très simples consiste à émuler le port parallèle d'un PC en vue de contrôler un instrument exploitant une telle interface. Le RADAR de sol (*Ground Penetrating RADAR*, GPR) de la société suédoise Malå est un instrument permettant de sonder les discontinuités de permittivité ou de conductivité dans le sous-sol à des profondeurs allant de quelques centimètres (état des routes ou des armatures dans les murs en béton armé par exemple) à quelques centaines de mètres (glaciers et banquise). Cet instrument, relativement ancien pour ses versions CU (Fig. 4) et CUII, se connecte à un ordinateur de contrôle de type compatible IBM au moyen du port parallèle (Centronics). Notre objectif est d'automatiser la mise sous tension de cet instrument, l'acquisition de trames contenant les mesures, stockage des informations sur support de stockage non-volatil, et mise en veille, le tout pour une consommation réduite en vue d'un fonctionnement autonome pendant plusieurs mois. Ayant identifié la fonction des diverses broches du port parallèle dans ce protocole de communication (en partie grâce à une documentation fournie par le constructeur, et en partie par nos propres écoutes sur le bus de communication lors de l'échange d'informations sous le contrôle du logiciel propriétaire commercialisé par le constructeur de l'instrument), ce protocole est émulé par GPIO. La puissance du Cortex M3 ne se justifie évidemment pas pour cette application triviale, mais nous verrons plus loin comment les informations acquises sont stockées sur carte SD (section 10) et comment le microcontrôleur est périodiquement réveillé de son mode de veille profonde par une horloge temps-réel fournie comme périphérique matériel indépendant (section 9). La puissance de calcul du Cortex M3 ne se justifie que s'il y a traitement embarqué des informations en vue d'en extraire les informations pertinentes permettant de réduire la quantité d'information stockée (compression, intercorrélation) ou en transmission par liaison sans fil.

L'extrait de programme ci-dessous propose une implémentation de la fonction d'écriture entre le STM32 et l'unité de contrôle du RADAR RAMAC (la fonction de lecture est trop longue et sans nouveauté pour être présentée ici, mais est disponible dans l'archive de programmes associée au manuscrit de cet article sur les sites http://jmfriedt.free.fr et www.trabucayre.com [5]. Connaissant le protocole de communication qui a été sondé par un analyseur logique Logic Sniffer [6], les diverses séquences de manipulation de l'état des broches sont implémentées de façon logicielle. La séquence

5. http://www.trabucayre.com/lm/lm_stm32_examples.tgz
6. http://dangerousprototypes.com/open-logic-sniffer/

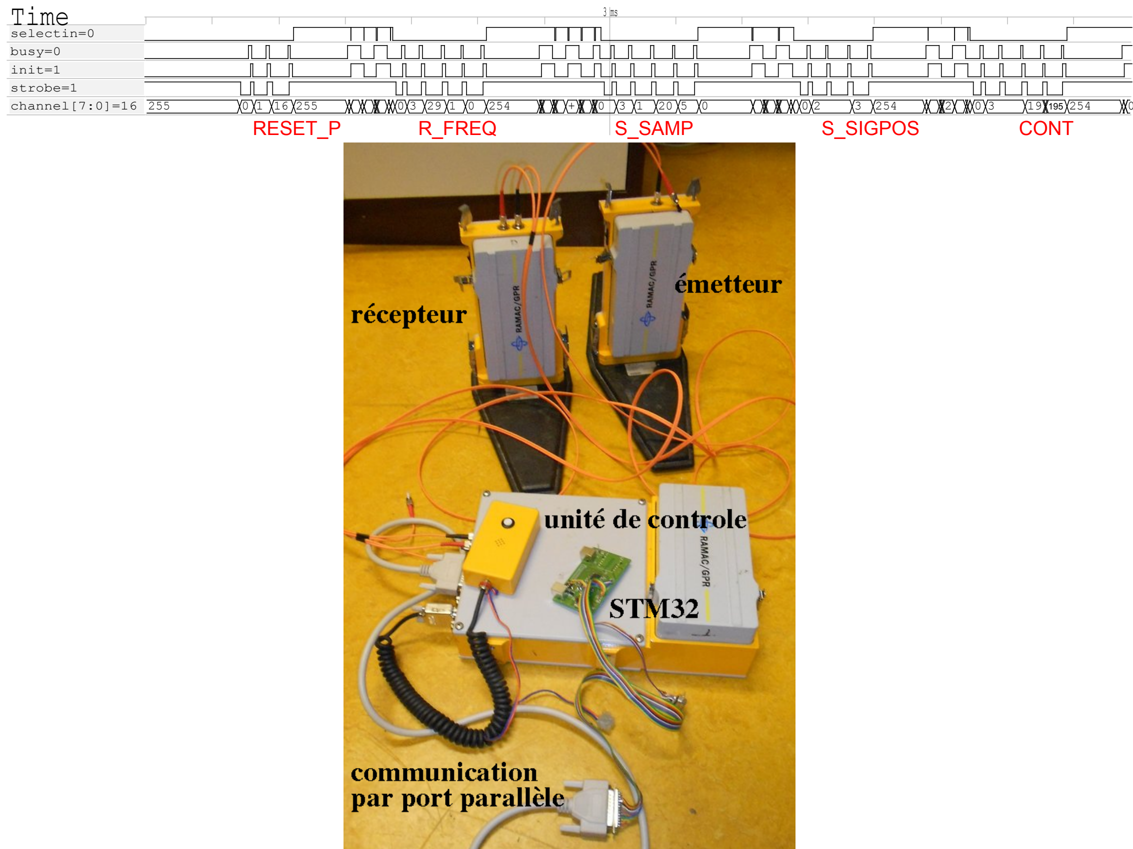

FIGURE 4 – Haut : chronogramme de la communication par port parallèle entre l'unité de contrôle du RADAR de sol RAMAC et un PC, sondé par un Logic Sniffer. Les données sont sur les broches 2 à 8 du port parallèle (bidirectionnel dans cette configuration), SELECTIN est la broche 17, BUSY est en 11, INIT est en 16 et STROBE est en 1. Tous les signaux de contrôle à l'exception de BUSY sont imposés par le microcontrôleur (sortie) à l'unité de contrôle (entrée) : BUSY sert à cadencer le flux de données et acquitter le transfert de données par l'unité de contrôle. Bas : unité de contrôle CU d'un Malå RAMAC (RADAR de sol – GPR pour *Ground Penetrating RADAR* – conçu pour communiquer avec un PC *via* le port parallèle, et ici commandé par un STM32 dont les GPIO émulent les fonctionnalités de ce bus de communication. Bien qu'une partie du protocole de communication nous ait été fournie par le constructeur, une phase d'écoute des informations transitant sur le bus a été nécessaire pour implémenter sur STM32 un protocole permettant de communiquer avec ce RADAR.

des commandes, dont seules les premières étapes d'initialisation sont décrites ici (Fig. 4, haut), consistent à placer l'unité de contrôle en liaison de données bidirectionnelles sur 8 bits, lecture de la fréquence de fonctionnement (`R_FREQ`), définition du nombre d'échantillons acquis (`S_SAMP`), position du début d'acquisition des échos par rapport à l'impulsion d'excitation (`S_SIGPOS`), et relance d'une acquisition temporairement interrompue (`CONT`).

```
 // CU input signals
2#define STROBE   0x01
 #define INIT    0x04
4#define SLCTIN   0x08
 // CU output signals
6#define BUSY    0x02

8#define TAILMAX 2700
```

```
void gpio_setup(void)
{gpio_set_mode(GPIOC, GPIO_MODE_OUTPUT_10_MHZ, GPIO_CNF_OUTPUT_PUSHPULL, INIT | STROBE | SLCTIN);
 gpio_set_mode(GPIOC, GPIO_MODE_INPUT, GPIO_CNF_INPUT_FLOAT, BUSY);
}

void busy_low(void)
{int i=0;
 do {i++; b = gpio_port_read(GPIOC);}
    while (((b & BUSY) == 0) && (i < MAXITER));
}

void busy_high(void)
{int i=0;
 do {i++;b = gpio_port_read(GPIOC);usleep(DELA);}
    while (((b & BUSY) != 0) && (i < MAXITER));
}

void write_command(unsigned char *cmd)
{int j = 0;
 do {
  gpio_port_write(GPIOB, (cmd[j] << 8));  // 1. envoi des commandes
  usleep(DELA);
  gpio_clear(GPIOC, STROBE);            // 2. strobe pulse
  busy_low();                           // 3. busy high, origine low
  gpio_set(GPIOC, INIT + STROBE);          // 4. strobe pulse
  busy_high();                            // 5. busy low (origine high)
  gpio_clear(GPIOC, INIT);            // 6. init bas
  j++;
     }
 while (j < (cmd[0]*256 + cmd[1] + 2));          // nbre elements dans cmd
}

int main(void)
{
 unsigned char cmd[10];
  [...]
 gpio_setup();
 cmd[0] = 0;cmd[1] = 1;cmd[2] = 16; // RESET_P
 write_command(cmd);
 read_command(cmd);
 [...]
}
```

6 Deuxième exemple : plus loin avec le clignotement d'une LED

Dans le premier exemple, les attentes entre deux transitions d'état du GPIO s'obtiennent par des boucles basées sur une instruction ne faisant rien. Non seulement cette solution n'est pas satisfaisante car la durée d'attente est intimement liée à la vitesse du cœur, mais en plus selon le niveau d'optimisation cette boucle peu potentiellement être supprimée par le compilateur. Nous allons présenter une solution plus fiable nous permettant d'avoir une base de temps plus stable.

Pour cela et généralement après avoir appris à manipuler une broche, le but suivant est d'apprendre à communiquer, nous allons appliquer cette régle mais d'une manière plus "ancienne", à savoir transmettre un classique *hello world* en morse [morse] au lieu du RS232.

"hello world" va donc donner ceci :

h	e	l	l	o	/	w	o	r	l	d
....	.	.-..	.-..	—		.–	—	.-.	.-..	-..

Le "." correspond à 1/4 de temps, le "-" et le temps entre deux caractères à 3/4. L'espace entre deux mots à 7/4 de période. Une attente de 1/4 de temps est placée entre chaque impulsion. Le code étant clair, l'implémentation sera également facile sans compliquer inutilement l'exemple.

Pour obtenir des durées précises et non dépendantes du compilateur nous allons utiliser un périphérique du STM32 dont le travail consiste à déclencher une interruption lors du débordement d'un compteur interne, le *systick*.

Dans la suite nous n'allons pas reprendre l'initialisation de l'horloge, pas plus que la configuration de la broche, nous allons juste nous focaliser sur la partie configuration du *systick* et son utilisation.

La première étape consiste en l'initialisation :

```
1 /* 72MHz / 8 => 9000000 counts per second */
  systick_set_clocksource(STK_CTRL_CLKSOURCE_AHB_DIV8);
3 /* 9000000/9000 = 1000 overflows per second - every 1ms one interrupt */
  systick_set_reload(9000);
5 systick_interrupt_enable();
  systick_counter_enable();
```

Le *systick* est configuré avec une fréquence de 9 MHz (l.2) et un débordement (génération d'une interruption) toutes les 1 ms (l.4). Les interruptions sont activées et le compteur est démarré. Bien entendu dans le code si dessus la durée avant interruption est codée en dur mais il est tout à fait possible de faire la même chose d'une manière plus élégante et capable de s'adapter automatiquement à la fréquence de l'horloge.

La seconde étape consiste en l'ajout du gestionnaire d'interruptions pour le *systick* et en la création d'une fonction d'attente.

```
  volatile uint32_t current_time32;
2
  /* we call this handler every 1ms */
4 void sys_tick_handler()
  {current_time++;}
6
  void Delay(uint32_t nCount)
8 {uint32_t begin = current_time;
   while (current_time-begin < nCount);
10 }
```

La fonction `sys_tick_handler` est, par convention de `libopencm3`, le gestionnaire pour l'interruption du `systick`. Cette fonction n'aura comme seul rôle que d'incrémenter une variable à chaque débordement. Quand la variable atteindra 0xffffffff, son incrément fera retomber la valeur à 0 ;

La seconde fonction (`Delay()`) va attendre que la différence entre le contenu de la variable lors de l'entrée dans la fonction et la valeur courante ait atteint la valeur désirée.

Finalement une dernière fonction qui va gérer le comportement d'attente selon le caractère fourni :

```
  void sendChar(uint8_t c)
2 {
      uint32_t delai;
4     switch (c){
      case '.': delai = SHORT_TIME; break;
6     case '-': delai = LONG_TIME; break;
      case ' ': Delay(LONG_TIME); return;
8     case '/': Delay(SPACE_TIME);
    default:
10          return;
      }
12    gpio_clear(GPIOC, GPIO7); Delay(delai);
      gpio_set(GPIOC, GPIO7);    Delay(SHORT_TIME);
14 }
```

La fonction est relativement simple à comprendre. Pour le caractère "." ou "-" une durée va être renseignée, la LED est allumée pendant un temps correspondant, puis éteinte, et une seconde attente va être réalisée. Dans les autres cas la fonction est mise en attente puis ressort sans toucher à la LED. `SHORT_TIME` correspond à 250 ms, `LONG_TIME` à 750 ms, et `SPACE_TIME` à 1750 ms.

Enfin il ne reste plus qu'à assembler le tout pour avoir un programme prêt pour la compilation.

```
  int main(void)
2 {int i;
   uint8_t message[] = ".... " //h
4         ". "        //e
```

```
    ".-.. "     //l
6   ".-.. "     //l
    "- "        //o
8   "/"         //
    ".- "       //w
10  "- "        //o
    ".-. "      //r
12  ".-.. "     //l
    "-.. \0";   //d
14 init_st();

16 while (1) {
    for (i=0;message[i]!= '\0';i++)
18   sendChar(message[i]);
    Delay(SPACE_TIME);
20 }
}
```

Hormis le tableau contenant le message, la fonction principale `main()` est très simple puisqu'après configuration du microcontrôleur, le programme rentre dans une boucle infinie qui va envoyer le message en permanence.

Nous pouvons voir clignoter une LED au rythme de l'envoi du message. Mais il faut reconnaître que ce n'est pas une manière spécialement évidente de transmettre des informations ni de debugger une application.

7 Communication RS232

Le STM32 dispose selon les modèles de 4 à 6 USARTs. Sur notre carte, l'USART1 est connecté à un convertisseur USB-série.

La configuration de ce périphérique se fait de la façon suivante : Comme tous les périphériques du STM32, il est nécessaire d'activer l'horloge pour l'USART ainsi que pour les broches de communication :

```
1 rcc_peripheral_enable_clock(&RCC_APB2ENR, RCC_APB2ENR_USART1EN);
  rcc_peripheral_enable_clock(&RCC_APB2ENR, RCC_APB2ENR_IOPAEN);
```

Chaque broche du microcontrôleur dispose de plusieurs fonctions selon les besoins, il est donc nécessaire de configurer les broches PA9 (USART1_TX) et PA10 (USART1_RX) en "alternate function push-pull" (GPIO_CNF_OUTPUT_ALTFN_PUSHPULL) pour TX et en "input floating" (GPIO_CNF_INPUT_FLOAT) ou "input pull-up" (GPIO_CNF_INPUT_PULL_UPDOWN) pour RX ([RM0008, pp.161-162]).

```
  /* Setup GPIO pin GPIO_USART1_TX/GPIO9 on GPIO port A for transmit. */
2 gpio_set_mode(GPIOA, GPIO_MODE_OUTPUT_50_MHZ,
          GPIO_CNF_OUTPUT_ALTFN_PUSHPULL,
4         GPIO_USART1_TX);
  /* Setup GPIO pin GPIO_USART1_RX/GPI10 on GPIO port A for receive. */
6 gpio_set_mode(GPIOA, GPIO_MODE_INPUT,
          GPIO_CNF_INPUT_FLOAT,
8         GPIO_USART1_TX);
```

Nous devons ensuite configurer le port et l'activer.

```
  void SetupUART()
2 {
    /* Setup UART1 parameters. */
4   usart_set_baudrate(USART1, 57600);
    usart_set_databits(USART1, 8);
6   usart_set_stopbits(USART1, USART_STOPBITS_1);
    usart_set_mode(USART1, USART_MODE_TX);
8   usart_set_parity(USART1, USART_PARITY_NONE);
    usart_set_flow_control(USART1, USART_FLOWCONTROL_NONE);
10  /* Finally enable the USART. */
    usart_enable(USART1);
12 }
```

Une fois encore, la **libopencm3** fournit des fonctions qui simplifient notre code de tous les détails spécifiques au matériel visé.

La dernière étape consiste à faire appel à cette fonction dans un *main* avant d'envoyer des messages sur le port série :

```
int main()
{
  int i;
  uint8_t message[] = "hello world!\r\n\0";
  clock_setup();
  SetupUART();
  while (1) {
    for (i = 0; message[i]!='\0';i++)
      usart_send_blocking(USART1, message[i]);
    Delay(0xaffff);
  }
  return 1;
}
```

La fonction `usart_send_blocking(..)` va s'assurer en premier lieu que l'USART est disponible (pas de caractère en cours d'envoi) en attendant que le bit `TXE` du registre `USART_SR` (status register) soit à '1'. Dès que le périphérique est disponible l'octet est chargé dans le registre `USART_DR` (data register) et la fonction rend la main.

8 Application pratique

Les premiers exemples nous ont permis de découvrir le STM32, de comprendre comment générer un binaire et de programmer le microcontrôleur. Cependant, cette compréhension est en somme relativement basique : nous savons certes comment manipuler des GPIOs, nous avons un mécanisme d'attente précis et une solution pour communiquer avec l'extérieur, mais par rapport aux périphériques disponibles sur ce microcontrôleur, cette première découverte n'est en somme qu'un "amuse-bouche", tout reste à faire en pratique.

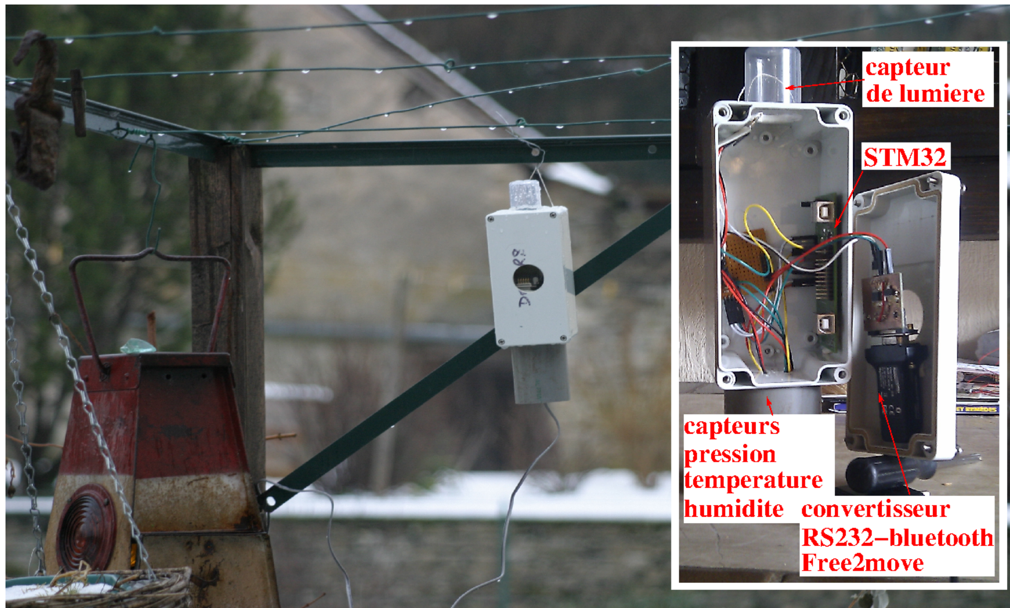

FIGURE 5 – Vue intérieure de la station et installation en extérieur. La liaison sans fil avec l'ordinateur chargé de placer les informations recueillies sur le web se fait par Bluetooth.

Nous nous sommes donc fixés comme objectif la réalisation d'une station météorologique (Fig. 5) équipée d'un capteur de pression, d'une sonde de température, d'un hygromètre et d'un capteur de lumière (Fig. 6). Au-delà de l'aspect ludique de pouvoir enregistrer les données sur une carte SD ou de les transmettre à un ordinateur, cette application va nous permettre d'aller plus avant

dans la découverte de ce microcontrôleur. Les composants choisis étant hétérogènes du point de vue du protocole, il est donc nécessaire de comprendre et maîtriser des périphériques tels que le SPI, l'I2C, l'ADC et les *timers*.

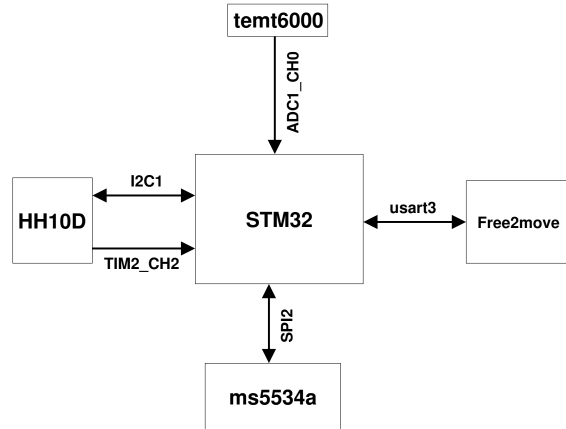

FIGURE 6 – Schéma bloc de la station. Les capteurs ont été sélectionnés de façon à illustrer l'utilisation des divers périphériques proposés par le STM32. L'acquisition de données se fait par conversion analogique-numérique (TEMT6000), communication numérique synchrone (SPI pour le MS5534A, I2C pour les paramètres du HH10D) et mesure de largeur d'impulsion (HH10D) exploitant les *timers*. Ces informations sont ensuite transmises pour une liaison sans fil Bluetooth au moyen d'une liaison numérique asynchrone (RS232).

Un second objectif de la compréhension de la structure de ce microcontrôleur est le portage de TinyOS sur celui-ci : cette partie sera présentée dans un prochain article. Afin de pouvoir coller au plus près à la structure interne du STM32 (registre, configuration, ...), nous avons décidé dans la suite de ne plus faire usage de la `libopencm3` qui, bien que pratique car permettant de développer rapidement sans avoir à connaître la structure de chaque registre pour un périphérique, empêche d'avoir une idée précise de comment faire en sorte de rendre les modules les plus génériques possibles et donc éviter la duplication de code.

8.1 MS5534A : SPI

8.1.1 Présentation du capteur

Le MS5534A[MS5534] est un capteur de pression compensé en température qui communique avec un microcontrôleur au moyen d'un bus synchrone bidirectionnel (deux signaux de données, l'un en provenance et l'autre vers le périphérique, et une horloge). Les commandes à envoyer sont sur 10 bits et les données reçues sont sur 16 bits.

Ce capteur fournit non seulement une information permettant de calculer la pression mais également la température. Afin d'obtenir ces deux informations, il est nécessaire de récupérer un jeu de constantes de calibrage qui permettront ensuite de réaliser les calculs nécessaires pour obtenir des valeurs en °C et en hPa.

La communication peut s'apparenter à du SPI à quelques exceptions près :

- il n'y a pas de Chip-Select, le capteur risque donc de parler alors qu'il n'est pas la cible ;
- le capteur lit les données sur le front montant de l'horloge et le microcontrôleur doit lire les données issues du capteur sur le front descendant. Il n'existe pas de mode SPI adapté à cette situation ;
- les commandes à lui envoyer sont codées sur 10 bits. La plupart des microcontrôleurs ne peuvent communiquer qu'en 8 ou 16 bits.

Une *application note* [AN510] explique comment utiliser ce composant avec ce protocole. Ainsi, pour palier au manque de CS, un buffer trois états (*74HC1G125* par exemple) doit être installé : ce composant dispose d'un enable actif à l'état bas. Grâce à ce composant il est possible de couper la ligne MISO afin de ne pas parasiter la communication dans le cas de l'utilisation de plusieurs esclaves SPI. Le problème du mode de communication non compatible est réglé par une reconfiguration de CPHA entre les phases de lecture et d'écriture. Finalement, ce document fournit les commandes en 16 bits (par adjonction de 0).

8.1.2 Configuration du SPI

En vue de communiquer avec le capteur, il est logiquement nécessaire de mettre en place un certain nombre de fonctions pour la configuration et la communication en SPI. Nous allons utiliser arbitrairement SPI2 mais le même code sera applicable à tous les autres SPI disponibles. Attention, la configuration des registres est identique, modulo le passage de l'adresse adaptée, mais par contre les broches ne sont (bien évidemment) pas les mêmes et le SPI1 est cadencé par APB2 et non APB1 (donc fréquence maximale de fonctionnement différente).

Comme nous l'avons vu dans les premiers exemples, il est nécessaire d'activer les périphériques ainsi que de configurer les broches :

```
void spi_rcc_gpio_config(void)
{
  rcc_peripheral_enable_clock(&RCC_APB1ENR, RCC_APB1ENR_SPI2EN);
  rcc_peripheral_enable_clock(&RCC_APB2ENR, RCC_APB2ENR_IOPBEN);

  gpio_set_mode(GPIOB, GPIO_MODE_INPUT,
        GPIO_CNF_INPUT_FLOAT, GPIO_SPI2_MISO);
  gpio_set_mode(GPIOB, GPIO_MODE_OUTPUT_50_MHZ,
        GPIO_CNF_OUTPUT_ALTFN_PUSHPULL,
        GPIO_SPI2_MOSI | GPIO_SPI2_SCK);
  gpio_set_mode(GPIOB, GPIO_MODE_OUTPUT_50_MHZ,
        GPIO_CNF_OUTPUT_PUSHPULL, GPIO1);
  gpio_set(GPIOB, GPIO1);
}
```

Vient maintenant la configuration du SPI (baudrate, mode master, CPHA, CPOL, ...). Dans le cas de l'utilisation du SPI sans interruption ni DMA, la configuration se fait uniquement au niveau du registre `SPI_CR1` :

```
void spi_config(void)
{
  /* disable SPI */
  SPI_CR1(SPI2) &=~(SPI_CR1_SPE);

  SPI_I2SCFGR(SPI2) & ~(1<<11);

  SPI_CR1(SPI2) = SPI_CR1_BIDIMODE_2LINE_UNIDIR
      | SPI_CR1_DFF_8BIT /* 8bits data */
      | SPI_CR1_SSM /* CS software */
      | SPI_CR1_MSBFIRST
      | SPI_CR1_BAUDRATE_FPCLK_DIV_8
      | SPI_CR1_MSTR /* master selection */
      | SPI_CR1_CPOL_CLK_TO_0_WHEN_IDLE
      | SPI_CR1_CPHA_CLK_TRANSITION_1;

  /* enable SPI */
  SPI_CR1(SPI2) |= SPI_CR1_SPE;
}
```

Par acquis de conscience, ne sachant pas l'état courant du SPI et comme certains paramètres ne peuvent être modifiés si le périphérique est actif, nous mettons le bit 6 (SPE : *SPI enable*) à 0 pour désactiver le SPI (l.4).

Les périphériques SPI peuvent également être configurés en I^2S (Inter-IC Sound, Integrated Interchip Sound) [7], il faut donc choisir le mode SPI en mettant à 0 le bit 11 du registre `SPI_I2SCFGR`(l.6)

7. http://wikipedia.org/wiki/I2S

Nous ne nous soucions pas du contenu antérieur du registre `SPI_CR1` que nous écrasons avec notre configuration. Le SPI est maintenant configuré en tant que maître (l.11), full duplex (MISO et MOSI)(l.6) pour des paquets de 8bits par transfert (l.7) le bit de poids fort en premier (l.9) , SCK à l'état bas lors de l'absence de transmission (l.12), avec une vitesse de transfert de 4,5 MHz (36 MHz pour APB1, divisé par 8)(l.10) et avec un Chip-Select géré manuellement (l.8) .

Il ne reste plus qu'à activer le périphérique (l.16).

Nous allons ajouter quelques fonctions utilitaires pour la communication avec le capteur : comme présenté précedemment ce capteur reçoit toutes les données lues sur le front montant de l'horloge et le microcontrôleur doit lire sur le front descendant :

```
1 #define SET_CPHA_0 do { \
          SPI_CR1(SPI2) &=~(SPI_CR1_SPE); \
3         SPI_CR1(SPI2) &= ~SPI_CR1_CPHA; \
          SPI_CR1(SPI2) |= SPI_CR1_SPE; \
5       } while(0)
  #define SET_CPHA_1 do { \
7         SPI_CR1(SPI2) &=~(SPI_CR1_SPE); \
          SPI_CR1(SPI2) |= SPI_CR1_CPHA; \
9         SPI_CR1(SPI2) |= SPI_CR1_SPE; \
        } while(0)
```

Le STM32 ne supporte pas un changement de mode alors que le SPI est actif, il faut donc désactiver celui-ci, changer le CPHA et le réactiver.

Ensuite nous créons une fonction qui va envoyer et recevoir les donnés.

```
  uint8_t spi_put(uint8_t data)
2 {
    SPI_DR(SPI2) = data;
4   while ((SPI_SR(SPI2) & (SPI_SR_TXE))==0x00);
    while ((SPI_SR(SPI2) & SPI_SR_RXNE)==0x00);
6   return SPI_DR(SPI2);
  }
```

La donnée est chargée dans le *Data Register* (*SPI_DR*), ensuite une attente est faite sur la transmission de celle-ci, puis sur la réception d'un octet depuis le capteur. Le même registre *SPI_DR* est utilisé pour récupérer l'information.

Et pour finir, la fonction qui va gérer la totalité de la communication (envoi d'un ordre, attente, puis récupération de l'information fournie par le capteur) :

8.1.3 Acquisition et traitement des données

La partie purement liée au SPI du STM32 (configuration et communication) est maintenant finie. Il nous faut finalement ajouter quelques fonctions pour envoyer et recevoir des commandes et les données, transmettre un reset au capteur, obtenir l'ensemble des paramètres nécesaires à l'exploitation de celui-ci, et traiter les informations de température et de pression obtenues.

```
1 uint16_t ms5534a_send(uint16_t cmd)
  {
3   uint8_t txbuf[2];
    uint8_t rxbuf[2];
5   uint16_t result = 0;

7   txbuf[0] = cmd >> 8;
    txbuf[1] = cmd & 0xff;
9
    SET_CPHA_0;
11  spi_put(txbuf[0]);
    spi_put(txbuf[1]);
13
    Delay(70);
15
    SET_CPHA_1;
17  rxbuf[0] = spi_put(0xff);
    rxbuf[1] = spi_put(0xff);
19
    result = rxbuf[0] << 8 | rxbuf[1];
21  return result;
```

```
}
```

Cette fonction sert à transmettre une commande et à recevoir la réponse du capteur. Les commandes, sur 16 bits, sont découpées en deux octets. Le périphérique est basculé dans le mode d'envoi, puis la commande est envoyée (l.8-10). Le ms5534a nécessite un temps de conversion de 33 ms, la donnée doit être récupérée dans un délai de maximum 100 ms après la fin de la conversion, ainsi nous utilisons la fonction *Delay()* présentée plus tôt pour garantir une attente correcte. Le SPI est ensuite basculé en mode lecture (l.16), les deux octets sont reçus (l.17-18) et la donnée sur 16 bits est reconstruite (l.20).

La seconde fonction concerne la transmission de la commande de `reset` :

```
void ms5534a_sendReset(void)
2{
  char txbuf[] = { 0x15, 0x55, 0x40 };
4 int i;
  SET_CPHA_0;
6 for (i=0; i<3;i++)
    spi_put(txbuf[i]);
8}
```

Cette fonction ne présente pas de difficultés particulières, comme elle ne fait qu'envoyer 4 octets sans faire de lecture, nous configurons le STM32 puis nous envoyons séquentiellement les octets.

Il nous faut ensuite être capable d'obtenir les données de calibrage du capteur :

```
void receiveParam(void)
2{
uint16_t w1, w2, w3, w4;
4ms5534a_sendReset();
w1 = ms5534a_send(CAL_READ_W1);
6w2 = ms5534a_send(CAL_READ_W2);
w3 = ms5534a_send(CAL_READ_W3);
8w4 = ms5534a_send(CAL_READ_W4);
ms5534a_c1 = w1 >> 1;
10ms5534a_c2 = ((w3 & 0x3F) << 6) | (w4 & 0x3F);
ms5534a_c2 *= 4;
12ms5534a_c3 = w4 >> 6;
ms5534a_c4 = w3 >> 6;
14ms5534a_c5 = (w2 >> 6) | ((w1 & 0x01) << 10);
ms5534a_c6 = (w2 & 0x3F);
16ms5534a_UT1 = (ms5534a_c5 * 8) + 20224;
ms5534a_ta = (ms5534a_c6 + 50);
18}
```

L'obtention des paramètres de calibrage du capteur n'a rien de particulièrement difficile non plus. Après avoir envoyé la commande de reset, tel que présenté dans la datasheet, nous récupérons les 4 mots (l.5-8). Ensuite nous calculons l'ensemble des informations nécessaires pour l'obtention de la pression et de la température que nous stockons dans des variables globales pour leur réutilisation ultérieure. Les `CAL_READ_Wx` sont des `#define` fournis dans [AN510].

Et pour finir nous créons la fonction de récupération et de calcul des deux grandeurs qui nous intéressent.

```
void ms5534a_read_pressure_and_temp(float *temp, uint32_t *pressure)
2{
  int t, dT, D1, D2, off, sens, x;
4
  ms5534a_sendReset();
6
  D1 = ms5534a_send(READ_PRESS);
8 D2 = ms5534a_send(READ_TEMP);

10 dT = D2 - ms5534a_UT1;
  /* temperature */
12 if (D2 >= ms5534a_UT1) {
    *temp = 200 + ((dT * ms5534a_ta) >> 10);
14 } else {
    t = dT / 128;
16  dT = dT - (t * t) / 4;
    *temp = 200 + ((dT * ms5534a_ta) >> 10) + (dT / 256);
18 }
```

```
/* Pressure */
off = ms5534a_c2 + (((ms5534a_c4 - 512) * dT) >> 12);
sens = ms5534a_c1 + ((ms5534a_c3 * dT) >> 10) + 24576;
x = ((sens * (D1 - 7168)) >> 14) - off;
*pressure = ((x * 10) >> 5) + 2500;
}
```

Aprés transmission à un ordinateur, nous pouvons générer des courbes d'évolution des informations fournies par le capteur telles que présentées Fig.7

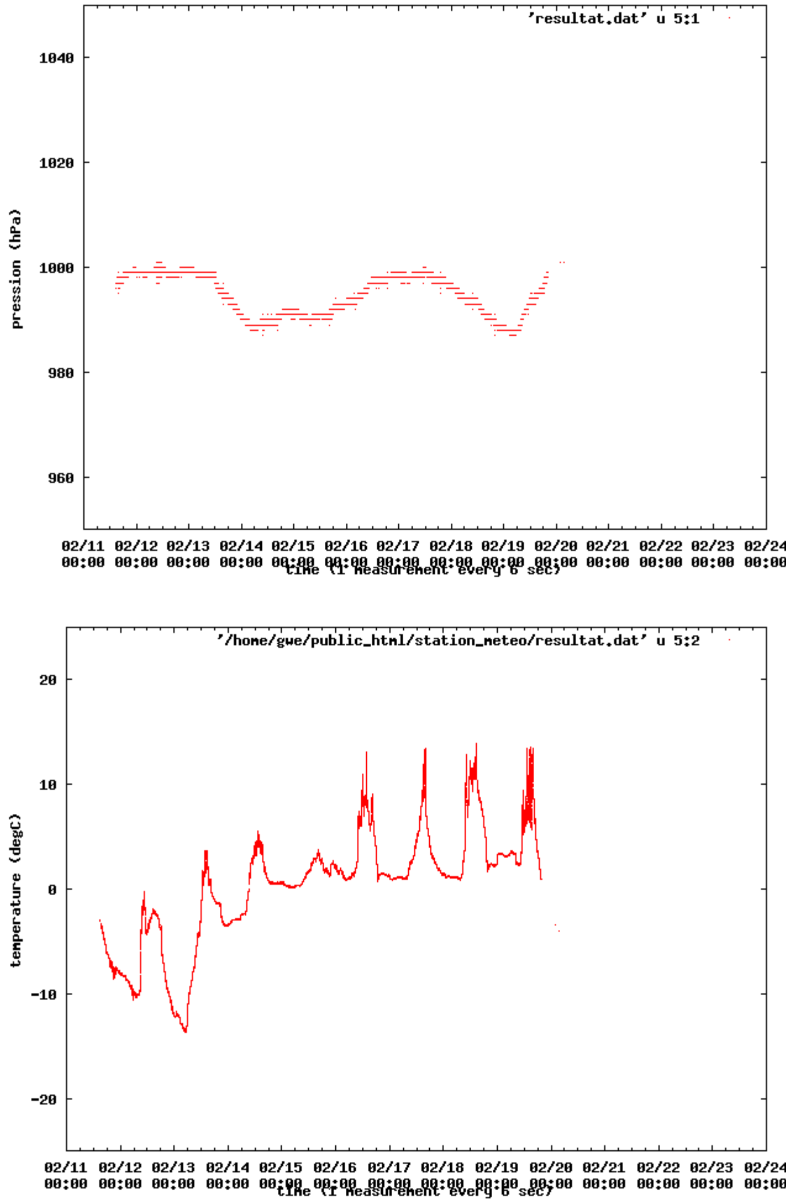

FIGURE 7 – Évolution de la pression athmosphérique (fig. haut) et de la température (fig. bas) entre le 11/02/2012 14:50 et le 19/02/2012 19:53. À partir de cette date, le capteur n'a plus donné d'informations correctes (322,1 °C et 183 hPa) ou alors la fin du monde est en avance !

8.2 HH10D : I2C et *input capture*

8.2.1 Présentation

Le HH10D [HH10D] est un capteur d'humidité qui fournit un signal périodique proportionnel au pourcentage d'humidité dans l'air. Chaque composant est étalonné individuellement, ces informations sont stockées dans une *EEPROM* accessible en *I2C*.

L'obtention de la durée de la période d'un signal se fait, sur STM32, à l'aide d'un des périphériques *timer*. Il existe deux modes de capture :

- le mode *input capture* qui permet d'obtenir uniquement la durée de la période du signal.
- le mode *pwm input capture* qui fournit en plus la durée de l'état haut ou bas (selon le front choisi pour la période) de ce même signal.

Bien que le HH10D fournisse un signal périodique (**rapport cyclique** de 50%), nous allons utiliser le *pwm input capture* pour comprendre ce mode qui est sans doute le plus complexe, sans pour autant nécessiter de grosses modifications par rapport au mode *input capture* de base.

8.2.2 Interrogation de l'EEPROM

Pour pouvoir exploiter l'information de période du signal, il est nécessaire de récupérer les données issues du calibrage du capteur, stockées dans l'*EEPROM*. Cette mémoire est accessible en *I2C* à une vitesse de communication de 400 kHz maximum.

Nous n'allons pas copier à nouveau l'activation de l'horloge pour ce périphérique. Il faut juste savoir que I2C1 est cadencé sur APB1 et utilise les broches `PB6` et `PB7` qui doivent être configurées en mode `GPIO_CNF_OUTPUT_ALTFN_OPENDRAIN`.

La configuration du périphérique est en somme relativement simple.

```
/* Disable the I2C. */
2 I2C_CR1(I2C1) &=~I2C_CR1_PE;

4 /* Set peripheral in i2c mode */
I2C_CR1(I2C1) &=~I2C_CR1_SMBUS;
6
/* APB1 is running at 36MHz.
8 * no interrupts generates */
I2C_CR2(I2C1) = 36;
```

Classiquement, le périphérique est désactivé (l.2). Pour l'heure, la configuration du registre `I2C_CR1` se résume à forcer le mode i2c et non le *SMBUS* (l.5). Le registre `I2C_CR2` ne pose pas plus de difficultés, nous ne voulons pas d'interruptions, il n'est donc nécessaire que de fournir la fréquence d'utilisation du périphérique (soit 36 MHz qui est le maximum pour APB1).

```
1 /* 400 kHz - I2C Fast Mode */
I2C_CCR(i2c) = I2C_CCR_FS // fast mode
3       | 0x00<<14 // tlow/thigh = 2
        | 0x1e; // 400 kHz : 2500 ns /(3*tpclk1)
```

Le registre `I2C_CCR` est dédié à la fréquence de transfert, à la forme du signal SCL, ainsi qu'au mode de fonctionnement. Nous le configurons en *fast mode* afin de pouvoir atteindre les 400 kHz (l.2), le signal SCL aura un rapport cyclique de 1/3 (l.3) et finalement nous fournissons la durée de la période de SCL (l.4). Cette valeur correspond à la durée de l'état haut du signal *SCL* divisée par la période de l'horloge(`I2C_CR2`) [RM0008, p.755]. Donc il nous faut en premier lieu connaître cette durée en divisant la période de SCL (400 kHz : 2500 ns) par 3. Ce qui nous donne 833,33 ns, et finalement diviser ce résultat par 27,778 ns (36 MHz) ce qui nous donne 30 ou 0x1e.

```
I2C_TRISE(I2C1) = 11;
2
/* Enable the peripheral. */
4 I2C_CR1(I2C1) |=I2C_CR1_PE;
```

La dernière partie va consister à configurer `I2C_TRISE`. Ce registre correspond au nombre maximum de période de l'horloge APB1 incrémenté de 1 pour la durée de transition entre l'état bas et l'état haut des signaux *SCL* et *SDA*. Cette information n'a pas de relation avec la vitesse de communication. Elle dépend du mode de communication et se trouve dans les spécifications du protocole *I2C*. Elle est de 1000 ns en *standard mode* et 300 ns en *fast mode*. La donnée à charger

dans le registre correspond donc, puisque nous sommes en *fast mode*, à (300 ns/27,778 ns) ce qui donne 10,8. Cette valeur n'est pas entière, il faut donc la tronquer, soit 10. Le résultat étant donc 11.

Finalement nous activons le périphérique (l.4).

La configuration du périphérique étant finie, nous allons ajouter les fonctions de communications sur bus i2c.

En se basant sur la documentation de l'EEPROM, la lecture des 4 octets nécessaires va se faire de la façon suivante :

- génération d'un *start bit*, suivit de l'envoi de l'adresse de l'EEPROM en mode écriture, le composant va produire un *ACK*, suite à quoi le microcontrôleur envoie l'adresse de la position en mémoire du premier octet à lire, cette seconde transmission va être également acquittée par l'EEPROM ;
- le microcontrôleur va re-générer un *start bit* et envoyer l'adresse de l'EEPROM mais en mode lecture (bit de poids faible à 1), comme précédemment l'EEPROM va acquitter la commande. Ensuite la mémoire va fournir les octets séquentiellement. À la fin de chacun d'eux, le microcontrôleur devra produire un *ACK* sauf pour le dernier octet où ce sera un *NACK* pour avertir le composant que la lecture est finie. Et finalement un *stop bit* est envoyé.

Par commodité nous allons créer une fonction dont le seul but sera de générer le *start bit* :

```
void i2c_send_start(void)
{
    /* send start */
    I2C_CR1(I2C1) |= I2C_CR1_START;
    while (!(I2C_SR1(I2C1) & I2C_SR1_SB));
}
```

Le bit `START` du registre `I2C_CR1` est mis à 1, ensuite le STM32 se met en attente du passage à 1 du bit `SB` de `I2C_SR1` (*start bit* envoyé).

Une seconde fonction va être utilisée pour l'envoi d'un octet au composant, en mode écriture :

```
void i2c_send_mess(uint8_t offset, uint8_t data)
{
    uint32_t reg32;
    /* Send destination address. */
    I2C_DR(I2C1) = addr;
    while (!(I2C_SR1(I2C1) & I2C_SR1_ADDR));
    /* Cleaning ADDR */
    reg32 = I2C_SR2(I2C1)

    /* Send data */
    I2C_DR(I2C1) = data;
    while (!(I2C_SR1(I2C1) & (I2C_SR1_BTF | I2C_SR1_TxE)));
}
```

L'adresse du composant est mise dans le registre de donnée, puis le STM32 se met en attente de la fin de la transmission (indiquée par le bit `ADDR` de `I2C_SR1`). Celui-ci passe à 1 lorsque l'adresse a été envoyée et que le composant esclave à transmis le ACK. La documentation du microcontrôleur précise qu'il est également nécessaire de lire `I2C_SR2` (l.8). Finalement il ne reste plus qu'à envoyer notre donnée, dans le cas présent l'offset du contenu de l'EEPROM. Cette transmission se fait de la même manière que pour l'adresse, mais avec une attente sur l'information "buffer d'envoi vide" (I2C_SR1_TxE) et transfert fini (I2C_SR1_BTF).

La partie réception est un peu plus complexe à mettre en œuvre. En fait il est nécessaire de gérer deux cas de figures. L'obtention d'un octet différent du dernier et le dernier.

La documentation du STM32 précise qu'il faut spécifier quelle sera la réponse (*(N)ACK*) faite pour la réception du prochain octet avant de se mettre en attente de la fin de la transmission de l'adresse, il n'est donc pas possible d'utiliser la fonction créée précédemment.

```
void receiveMess(uint8_t addr, uint8_t *data)
{
    uint32_t reg32;
    uint16_t pos=0;
    /* Send destination address. */
    I2C_DR(i2c) = addr; // admis que le bit de mode est fourni
    I2C_CR1(i2c) |= (I2C_CR1_POS );
```

```
  I2C_CR1(i2c) |= I2C_CR1_ACK;
9 while (!(I2C_SR1(i2c) & I2C_SR1_ADDR));
  reg32 = I2C_SR2(i2c);
```

La différence avec le code précédent réside aux lignes 7 et 8. `I2C_CR1_POS` précise que le *(N)ACK* sera émis lors de la réception du prochain octet, le `I2C_CR1_ACK` précise que ce sera un *ACK*.

```
   while (len != 0) {
2    if (len >1) {
       while(!(I2C_SR1(i2c)&I2C_SR1_RxNE));
4      data[pos++] = I2C_DR(i2c);
       len--;
6    }else {
       I2C_CR1(i2c) &= ~(I2C_CR1_POS );
8      I2C_CR1(i2c) &= ~I2C_CR1_ACK;
       I2C_CR1(i2c) |= I2C_CR1_STOP;
10     while(!(I2C_SR1(i2c) & I2C_SR1_RxNE));

12     data[pos++]= I2C_DR(i2c);
       len = 0;
14   }
   }
16}
```

Ensuite il faut boucler tant qu'il y a des octets à recevoir en dissociant deux cas :

- le cas où il reste plus qu'un octet à recevoir. L'attente se fait sur le simple fait d'avoir le registre de donnée non vide (`I2C_SR1_RxNE`) ;
- le cas où le prochain octet est le dernier. Avant de se mettre en attente de la réception il est nécessaire de configurer le STM32 pour qu'un *NACK* (l.8) soit envoyé à la fin de la réception de l'octet en cours de transfert (l.7), suivit par le *stop bit* (l.9).

Et finalement nous créons une fonction qui va faire appel à toutes les fonctions présentées pour obtenir les informations dont nous avons besoin.

```
  uint8_t res[4];
2
  i2c_init(I2C1);
4
  // envoi de l'adresse du premier octet
6 i2c_SendStart();
  i2c_sendmess(HH10D_ADDR, HH10D_SENS);
8
  // envoi de la commande de lecture
10// et recuperation des octets
  i2c_SendStart();
12i2c_receiveMess(HH10D_ADDR|0x01, res, 4);
  sensivity = res[0]<<8|res[1];
14offset = res[2]<<8|res[3];
```

Ainsi nous en avons fini avec l'obtention des paramètres de calibrage du capteur et nous pouvons passer à l'acquisition de données.

8.2.3 Récupération de la période du signal et du rapport cyclique

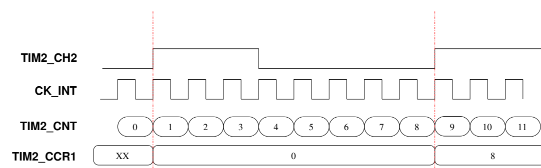

FIGURE 8 – Utilisation du mode *input capture* sur un signal appliqué à la broche `TIM2_CH2`. Le registre `TIM2_CCR1` mémorise le contenu du compteur `TIM2_CNT` pour chaque front montant du signal.

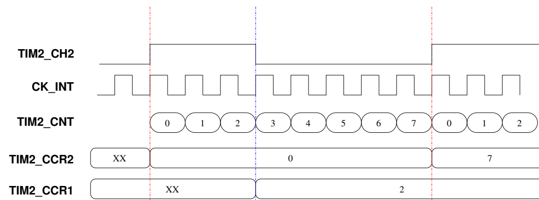

FIGURE 9 – Utilisation du mode *pwm input capture*, le registre `TIM2_CCR2` est mis à jour pour chaque front montant du signal et le registre `TIM2_CCR1` chaque front descendant. À noter que la détection d'un front montant réinitialise le contenu de `TIM2_CNT` (compteur global du *timer*).

Chaque *timer* du STM32 dispose de 4 canaux pouvant être reliés aux broches du boîtier. Chacun de ces canaux est capable de capturer la période d'un signal (durée entre deux fronts montants ou descendants successifs), c'est le mode *input capture* (Fig. 8). Tel que représenté sur cette figure, le registre correspondant au canal 1 (`TIM2_CCR1`) est mis à jour avec la valeur courante du compteur global.

Il existe une variante de ce mode dans lequel deux canaux sont liés (Fig. 9) et observent le même signal. Un des canaux (le canal 2 sur la figure) est utilisé pour l'obtention de la période de la même manière qu'en *input capture*. Le second canal (canal 1) va être sensible au front opposé du signal, ainsi sur la figure, ce canal fournira la durée de l'état haut. Afin de synchroniser les deux canaux, lorsque la condition de déclenchement du canal maître (canal 2) se produit, le compteur global du *timer* est remis à 0. Dans ce cas un *timer* ne pourra plus être utilisé que pour une seule capture.

Comme pour tous les périphériques du STM32, il est nécessaire d'activer l'horloge pour *TIM2* (sur *APB1*) et pour le port contenant la broche *PA1* (*APB2*), ainsi que de configurer cette dernière en `INPUT_FLOAT`).

La configuration du *timer2* en *pwm input capture* commence par le réglage du compteur global de ce périphérique (valeur maximale du compteur, fréquence de comptage, etc.). Nous savons que le HH10D génère une fréquence entre 5 kHz et 10 kHz. Donc nous devons pouvoir compter, au moins, jusqu'à 200 μs. La fréquence de l'horloge utilisée pour tous les *timers* (sauf *TIM1*) dépend de *APB1* et de son *prescaler*[8] :

- s'il n'est pas utilisé la fréquence de *TIMxCLK* est identique à *APB1*;
- s'il est utilisé alors la fréquence de *TIMxCLK* est le double de celle de *APB1*.

Comme la PLL qui fournit la fréquence aux *APBx* est de 72 MHz, le *prescaler* de *APB1* est utilisé (division par 2 pour ne pas dépasser les 36 MHz). Ainsi *TIM2CLK* vaut 36 MHz x 2 (= 72 MHz), soit une période de 14 ns. À cette fréquence, il est possible de compter jusqu'à 910215 ns, soit 900 μs, ce qui est plus que suffisant pour nos besoins. Il serait possible de réduire la vitesse mais nous aurions une perte de précision.

```
  void timeBaseInit()
2 {
    TIM_CR1(TIM2) = TIM_CR1_CKD_CK_INT /* clockdivision*/
4          |TIM_CR1_DIR_UP;
    /* Set the Prescaler value */
6   TIM_PSC(TIM2) = 0;
    TIM_ARR(TIM2) = 0xffff;
8 }
```

Nous configurons TIM_CR1 pour que le compteur soit cadencé à la fréquence de l'horloge d'entrée (l.1) et s'incrémente (l.2). Nous configurons TIM2_PSC pour que le compteur soit incrémenté à chaque période de l'horloge d'entrée donc à 72 MHz. Le compteur repassera à 0 lorsqu'il atteindra 0xffff.

Une fois la base de temps définie, il faut configurer les deux canaux du *timer*. Les canaux 1 et 2 peuvent être utilisés avec comme source les broches *TIM2_CH1* ou *TIM2_CH2*

8. cette information est seulement donnée sur la [RM0008, fig.8 p.90]

Du point de vue de la configuration, les 4 canaux partagent le registre `TIM2_CCER`, `TIM2_CH1` et `TIM2_CH2` partagent `TIM2_CMR1` et `TIM2_CH3` et `TIM2_CH4` partagent `TIM2_CMR2`

Voyons donc d'abord la configuration du canal 2 :

```
void channel2_config()
{
  /* Disable capture on channel 2 and reset capture polarity */
  TIM_CCER(TIM2) &= ~((1<<4)|(1<<5));
  // Set capture on rising edge
  TIM_CCER(TIM2) |= 0<<5;

  /* Reset Channel 2 configuration */
  TIM_CCMR1(TIM2) &= ~0xff00;
  /* Select the input and set the filter */
  TIM_CCMR1(TIM2) |= 1<<12// sampling = fck_int
        | (0<<10) // no prescaler, capture done for each event
        | (0x01<<8); // direct : TI2 is used as input
  // Enable capture on channel 2
  TIM_CCER(TIM2) = 1<<4;
```

Le canal 2 est configuré pour être déclenché sur le front montant du signal (l.6).

Ensuite nous configurons le registre `TIM_CCMR1` pour que le canal fonctionne à la fréquence de l'horloge sans *prescaler* et nous spécifions que le canal 2 est connecté en direct (sur `TIM2_CH2`).

Il ne nous reste plus finalement qu'à activer le canal (l.15).

Si nous ne souhaitons faire que du *input capture*, la configuration est quasiment finie à ce stade. Mais comme nous allons utiliser le mode PWM, il est encore nécessaire de configurer le canal 1. Pour éviter d'ajouter trop de code et comme les deux configurations sont pratiquement identiques seul l'offset diffère (il faut enlever 8 à chaque décalage sur `TIM_CCMR1` et 4 sur `TIM_CCER`). Les seules lignes réellement importantes concernent le front de déclenchement et le passage en mode indirect, c'est-à-dire la connexion du canal 1 sur la broche `TIM2_CH2`. Pour le canal 2 en mode direct nous avions :

```
TIM_CCER(TIM2) |= 0<<5;
```

et :

```
| (0x01<<8); // direct : TI2 is used as input
```

dans le cas du canal 1, nous aurons :

```
TIM_CCER(TIM2) |= 1<<1;
```

et :

```
| (0x02<<0); // indirect : TI2 is used as input
```

Pour finir avec la configuration du périphérique nous avons besoin de synchroniser les deux canaux.

```
    /* Set the Input Trigger source */
    TIM_SMCR(TIM2) = 1<<7 // master slave mode enable
                  | 0x05<<4 // select TI2FP2 for synchro
                  | 0x4; // rising edge reset counter
```

Nous passons donc le *timer* en mode maître esclave (l.2), avec une synchronisation des canaux sur TI2FP2 (broche TIM2_CH2) et avec remise à 0 du compteur lors d'un front montant détecté sur cette broche.

Il ne reste plus qu'à activer le périphérique :

```
TIM_CR1(TIM2) |= TIM_CR1_CEN
```

Pour finir nous ajoutons une fonction pour obtenir les informations concernant la forme du signal et pour calculer le taux d'humidité :

```
float getHumidity()
{
    uint16_t period, DutyCycle;
    float time, freq;
  while((TIM_SR(TIM2) & (TIM_IT_CC2)) == 0x00);
  /* Get the period value */
  period = TIM2_CCR2;
  /* Duty cycle value */
  DutyCycle = TIM2_CCR1;

    time = period*13.889;//TIME_BASE;
    freq = (1000000/time) * 1000.0f;
```

```
13   return (float)((offset-freq)*sensivity)/4096
```

le STM32 est mis en attente d'un front montant (et donc de la fin de la capture) (l.5), les valeurs mesurées sont disponibles dans le registre relatif au canal 2 pour la période (l.7) et au canal 1 pour la durée de l'état haut du signal (l.9). Nous convertissons la valeur de la période en une durée (l.11) puis en une fréquence en kHz (l.12) et nous réalisons le calcul pour obtenir le taux d'humidité.

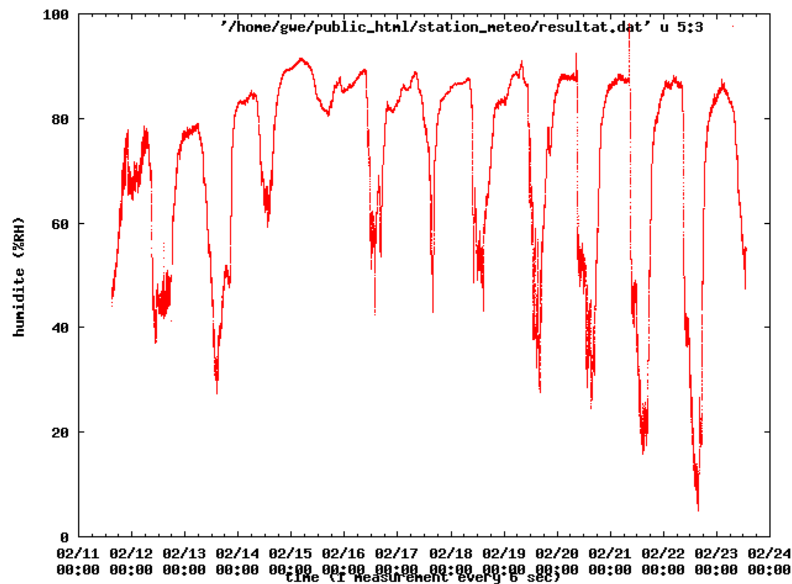

FIGURE 10 – Exemple de courbe d'humidité sur la période du 11-02-2012 14:50 au 23-02-2012 13:08. On peut constater les fortes variations entre le jour et la nuit.

8.3 TEMT6000 : ADC

8.3.1 Présentation

Le dernier composant que nous allons mettre en œuvre est le TEMT6000[TEMT6000]. C'est un capteur de lumière ambiante. Ce composant se présente sous la forme d'un transistor NPN (monté en collecteur commun avec une résistance de 10 kΩ) qui fournit une tension proportionnelle à l'intensité lumineuse.

Pour cela nous allons utiliser l'ADC du STM32, le composant étant connecté sur la broche *PA0*.

8.3.2 Configuration et acquisition

Contrairement aux précédents exemples, l'ADC dispose d'un *prescaler* au niveau du *RCC*. Ce composant ne peut pas être cadencé à plus de 14 MHz, la fréquence d'entrée du *prescaler* étant donnée par *APB2* (72 MHz dans notre cas). Il n'est possible que de faire des divisions par 2,4,6,8. Ainsi pour obtenir la fréquence la plus rapide sans dépasser la limite, il faut donc diviser par 6.

```
1 RCC_CFGR &= ~(0x02<<14);
  RCC_CFGR |= 0x02<<14;
```

Le coefficient de division étant donné [RM0008, p.125] (2 correspond à une division par 6).

L'étape, classique, et suivante consiste en l'activation du périphérique et du port contenant la GPIO qui nous intéresse. Il faut donc activer *ADC1* et le port A sur `APB2`. La broche sur laquelle est connectée étant *PA0* (*ADC1_0*), elle doit être configurée en mode entrée et de type `INPUT_ANALOG`.

Finalement nous allons pouvoir configurer l'ADC :

```
void init_adc()
{
  /* disable ADC1 */
  ADC_CR2(ADC1) &=~ADC_CR2_ADON;

  /* no watchdog, no interrupts, regular mode, one channel*/
  ADC_CR1(ADC1) = 0;
  /* no dma, data aligned right, no external events */
  ADC_CR2(ADC1) = 0;
```

Après la désactivation du phériphérique nous configurons les deux registres de contrôle. Pour `ADC_CR1` nous ne souhaitons pas l'utilisation du *watchdog*, ni des modes tels que *scan mode* ou *dual mode*, et nous n'allons pas utiliser d'interruptions, sa configuration se résume à y mettre 0. Pour `ADC_CR2` le résultat est le même car nous n'allons pas utiliser la température du STM32, nous n'exploiterons pas plus le déclenchement de la conversion sur évènement, ni le DMA et nous souhaitons que les 12 bits issus de la conversion soient alignés à droite (bit de poids faible).

Nous configurons ensuite le temps de conversion du canal que nous allons exploiter :

```
  /* 28.5 cycle for channel 0 */
  ADC_SMPR2(ADC1) &=~(0x07);
  ADC_SMPR2(ADC1) |=(0x03);
```

Pour cela plusieurs registres sont disponibles, pour le cas qui nous intéresse, le canal 0 de l'ADC se trouve (ainsi que les canaux < 10) dans `ADC_SMPR2`. Les trois bits destinés à ce canal sont remis à 0 puis nous le configurons pour avoir un temps de conversion de 28,5 cycles (valeur 3 [RM0008, p.235]). Avec cette configuration le temps de conversion sera de 3,41 μs (tconv = (sample time + 12,5 cycles)*période de ADCCLK) [RM0008, section 11.6, p.216]

Maintenant il nous faut activer et calibrer l'ADC :

```
  /* enable ADC1 */
  ADC_CR2(ADC1) |= ADC_CR2_ADON;
  /* reset calibration */
  ADC_CR2(ADC1) |= ADC_CR2_RSTCAL;
  while ((ADC_CR2(ADC1) & ADC_CR2_RSTCAL)!= 0x00);
  /* calibration */
  ADC_CR2(ADC1) |= ADC_CR2_CAL;
  while ((ADC_CR2(ADC1) & ADC_CR2_CAL) != 0x00);
```

Cette étape se fait en deux temps, nous commençons par réinitialiser le registre de calibrage (l.4) et attendons que le bit soit remis à 0 (l.5) signifiant que l'opération s'est effectuée avec succés. Ensuite nous lançons le calibrage (l.7) et attendons la fin du traitement (l.8).

Pour finir avec la configuration, il faut spécifier quel sera le canal qui sera utilisé (dans cet exemple nous n'utilisons qu'un seul canal, mais nous pourrions aller jusqu'à 16).

```
    ADC_SQR1(adc) = 0<<20|0;
    ADC_SQR2(adc) = 0;
    ADC_SQR3(adc) = 0;
}
```

Le registre `ADC_SQR1` contient à la fois 4 bits (bits [23:20]) pour spécifier le nombre de canaux à exploiter à chaque fois qu'une conversion est lancée, 0 signifiant une conversion. Le reste de ce registre contient l'ordre des canaux lorsque plus de 12 conversions sont à faire. le registre `ADC_SQR2` comporte les canaux pour les conversions 12 à 7, et finalement `ADC_SQR3` pour les conversions 6 à 1. Comme nous ne voulons qu'une seule conversion sur le canal 0 nous laissons ce dernier registre à 0.

Nous en avons donc fini avec la configuration. Une demande de conversion se fait de la façon suivante :

```
uint16_t getConversion()
{
  ADC_CR2(ADC1) |= ADC_CR2_ADON;

  /* Wait for end of conversion. */
  while ((ADC_SR(ADC1) & ADC_SR_EOC)==0x00);

  temp = ADC_DR(ADC1);
  return (3300*temp/4095);
}
```

Une conversion est déclenchée quand le bit ADON est remis à 1 (l.3). Nous attendons ensuite que le bit `SR_EOC` du registre `ADC_SR` passe à 1 (l.6) pour récupérer le résultat de la conversion (l.8). Comme l'ADC est sur 12 bits et que la valeur maximale correspond à 3,3 V nous divisons le résultat par 4095 puis nous le multiplions par 3300 pour retourner le résultat en milivolts (Fig. 11).

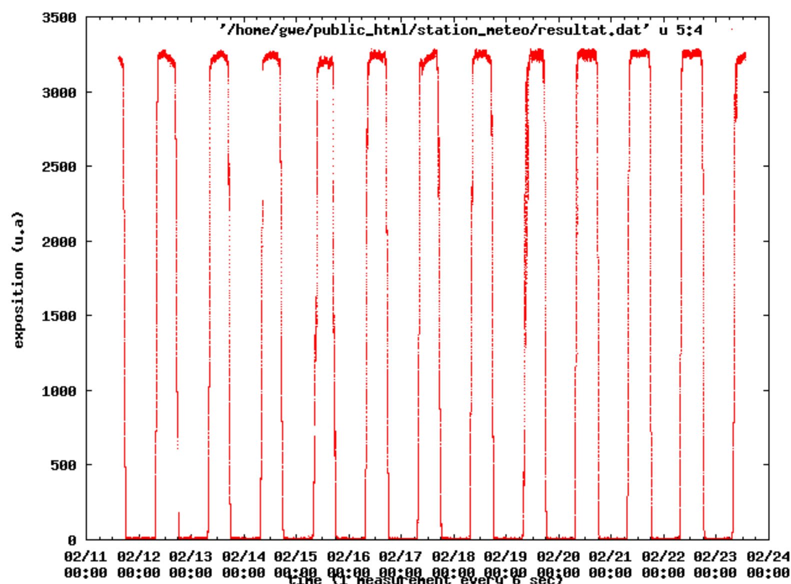

FIGURE 11 – Évolution de l'éclairage sur la période du 11-02-2012 14:50 au 23-02-2012 13:08. Ce capteur semble nécessiter un remplacement de la résistance (1 kΩ au lieu de 10 kΩ) afin d'obtenir plus de détails concernant les maxima.

8.4 Assemblage et mise en service

Nous avons à disposition l'ensemble des pilotes nécessaires à la fois pour l'exploitation du STM32 mais également pour la communication et le traitement des données acquises des capteurs de la station.

```
void init_component()
{
  init_hh10d();
  ms5534a_init();
  init_adc();
  systick_init();
}

int main(void)
{
  float temp, hygro;
  uint32_t pressure;
  uint16_t lux;
  char buffer[256];

  clock_setup();
  usart3_setup(57600);
  init_component();

  while (1) {
    hygro = getHumidity();
    ms5534a_read_pressure_and_temp(&temp, &pressure);
```

```
    lux = readAdc(0);
24
    sprintf(buffer, "%04d %03f %04f %04u\r\n", (int)pressure,
26          temp, hygro, lux);
    usart3_put_string(buffer);
28
    Delay(6000);
30  }
}
```

Il faut donc associer tout ça dans un *main()*. Pas grand chose de complexe, après la configuration du STM32 comme présentée au début, nous configurons l'USART3 sur lequel est connecté le convertisseur bluetooth-série. L'ensemble des composants est ensuite configuré ainsi que le *systick* pour appliquer une pause entre chaque envoi.

Dans notre boucle infinie, les données sont acquises, formatées et envoyées, toutes les 6 secondes.

Bien entendu, une attente active faite par notre fonction *Delay* n'est pas idéale. Il serait plus judicieux d'exploiter l'un des modes de mise en veille du microcontrôleur afin de réduire la consommation globale de la station.

9 Passage en mode faible consommation et réveil périodique

Le STM32 (et plus largement l'ensemble des cortex-m3 semble-t'il) dispose de trois modes faibles consommations :

- le *sleep mode* qui ne fait qu'arrêter l'horloge du CPU, mais laisse actif les quartz ainsi que les périphériques. Le microcontrôleur peut être réveillé à l'aide d'une interruption ;
- le *stop mode* qui arrête les quartz haute fréquence et la plupart des périphériques. La seule possibilité pour sortir le microcontrôleur de ce mode est qu'il reçoive une interruption externe (broches dédiées) ;
- le *standby mode* qui a les mêmes caractéristiques que le précédent, mais qui autorise le réveil à l'aide de la RTC(*Real Time Clock*). Ce mode est le plus économique mais présente le défaut de remettre le STM32 dans son état initial (perte des configurations des registres et remise à zéro de la mémoire), comme après une mise à `GND` de la broche `RESET`.

Nous allons nous intéresser plus spécialement au *standby mode* en reposant le réveil sur l'usage de l'horloge temps-réel (RTC). Pour simplifier cette présentation et comme ce n'est pas réellement un point critique à comprendre en terme de programmation nous allons faire usage de la `libopencm3`, seuls les parties non disponibles dans celle-ci seront faites à la main.

L'exemple ci-dessous initialise l'horloge temps-réel pour s'incrémenter toutes les secondes (compteur à 0x7fff sur le quartz à 32768 Hz) et place le microcontrôleur en mode de veille profonde tel que décrit dans la section 5.3 de [RM0008, p.70]. Lors de chaque interruption, l'alarme est incrémentée de 10 s.

```
1 #define temps_mesure 10    // exprime en seconde

3 void init_rtc(void)
{
5   // enable clock
    rcc_peripheral_enable_clock(&RCC_APB1ENR,
7     RCC_APB1ENR_BKPEN | RCC_APB1ENR_PWREN); //BKP (backup domain)

9   rtc_auto_awake(LSE, 0x7fff);

11  /* Without this the RTC interrupt routine will never be called. */
    nvic_enable_irq(NVIC_RTC_IRQ);
13  nvic_set_priority(NVIC_RTC_IRQ, 1);

15  /* Enable the RTC interrupt to occur off the SEC flag. */
    rtc_interrupt_enable(RTC_ALR);
17  rtc_enable_alarm();
}
19
void rtc_isr(void)
```

```
21 {
     if ((RTC_CRL & RTC_CRL_ALRF) != 0x00) {
23     rtc_clear_flag(RTC_ALR);
       rtc_set_alarm_time(rtc_get_counter_val() + temps_mesure);
25   }
   }
```

Afin d'être en mesure d'utiliser la RTC, il est nécessaire d'activer à la fois le *backup domain* et le *power control*. Ensuite la RTC elle-même est configurée pour être cadencée par le quartz basse fréquence externe et pour incrémenter le compteur toutes les secondes. Il ne reste plus ensuite qu'à activer globalement les interruptions pour la RTC et spécifier que l'événement "alarme" (valeur configurée du compteur atteinte) doit déclencher une interruption.

la fonction `rtc_isr` est le gestionnaire d'interruptions pour la RTC. Le seul traitement réalisé étant d'acquitter l'interruption puis de réamorcer l'alarme pour un déclenchement "temps_mesure" après la date actuelle. Dans notre cas ce traitement n'a pas réellement d'intérêt car le STM32 sera réinitialisé dès le déclenchement de cette interruption.

```
   void PWR_EnterSTANDBYMode(void)
 2 {
     /* Set SLEEPDEEP bit of Cortex System Control Register */
 4   SCB_SCR |= SCB_SCR_SLEEPDEEP;
     /* Select STANDBY mode */
 6   PWR_CR |= PWR_CR_PDDS;
     /* Clear Wake-up flag */
 8   PWR_CR |= PWR_CR_CWUF;
     __asm volatile ("WFI");
10 }
```

Cette fonction est la plus importante pour le passage en *standby mode*. Le registre `SCB_SCR` n'est pas documenté dans [RM0008] car il est relatif au cœur ARM CM3. le bit `SCB_CR_SLEEPDEEP` sert à déterminer si le processeur doit passer en *sleep mode* ou dans l'un des deux autres modes.

Le bit `PWR_CR_PDDS` spécifie que le STM32 va entrer en *standby mode* (bit à 1) ou en *stop mode* (bit à 0) lors du passage en mode *deep sleep*.

Le passage en mode basse consommation est ensuite validé par la mnémonique `WFI`.

```
   int main(void)
 2 {
     [...]
 4   init_rtc();

 6   [...]
     rtc_set_alarm_time(rtc_get_counter_val() + temps_mesure);
 8   PWR_EnterSTANDBYMode();
     while (1);
10
   }
```

Après configuration de la RTC, nous réalisons nos traitements. Quand ceux-ci sont fini nous fixons le moment où la RTC va produire son interruption et nous rentrons dans la fonction qui passe le STM32 en mode basse consommation. La boucle infinie ne sera finalement jamais atteinte car à la sortie de `WFI` le STM32 sera réinitialisé. Les mesures de consommation (table 1) se font dans un premier temps avec un convertisseur série-USB FT232RL en mode veille (RESET# en position basse), avant que les résultats excessivement élevés nous obligent à effectuer une mesure finale avec le STM32 seul, en mode veille, sans aucun composant périphérique autre que les condensateurs de découplage et les quartz. Lors de la présence du FTDI, la liaison asynchrone entre le FTDI (TX et RX) et le STM32 se fait au travers de résistances de 36 kΩ.

Dans cet exemple nous ne faisons rien de particulier, mais il peut sembler parfois laborieux voir consommateur en ressources de devoir, pour chaque réveil, faire toute une série d'initialisation du fait de la réinitialisation de la mémoire volatile. Pour palier à ce problème le STM32 dispose de 16 registres de 16 bits (`BKP_DRxx`) qui ne seront pas réinitialisés lors de la mise en veille, ils peuvent donc permettre d'éviter certains traitements pouvant être fait une fois et réutilisés ensuite.

À titre indicatif, la fonction `RTC_GetTime()` propose une lecture du contenu du registre de l'horloge temps-réel en vue d'une datation des trames acquises.

```
1 void RTC_GetTime(volatile u32 * THH, volatile u32 * TMM, volatile u32 * TSS)
```

Statut	consommation (mA)
STM32 seul	0,0036
FTDI seul (USB retiré, RESET#=bas)	1,72
mode veille profonde (quelque soit config. quartz)	1,89
mode veille (Sleep) (quelque soit config. quartz)	16
quartz externe 8 MHz, PLL 72 MHz	36
quartz externe 8 MHz, PLL 24 MHz	18
oscillateur interne, PLL 24 MHz	18
oscillateur interne, PLL 64 MHz	33

Table 1 – Mesures de consommation sur un circuit contenant un `STM32F103RCT6`, avec ou sans (première ligne uniquement) FT232 : nous constatons qu'en mode actif, la consommation croît avec la vitesse d'horloge du cœur, mais semble indépendant de la nature de l'oscillateur. Un courant de fuite important affecte les mesures lorsque le STM32 est connecté au FT232 : en enlevant ce dernier, la consommation en mode de veille profonde chute à 3,6 μA, une valeur qui n'est que le double de celle observée sur MSP430 [MSP430].

```
 {volatile u32 tmp;
3 tmp = rtc_get_counter_val();
   *THH = (tmp / 3600) % 24;
5  *TMM = (tmp / 60) % 60;
   *TSS = tmp % 60;
7}
```

Le STM32 n'a pas vocation à être exploité dans les applications à très basse consommation. Néanmoins, mentionnons que la série de processeurs Tiny Gecko de Energy Micro [9] annonce des consommations aussi basses que 900 nA en mode veille et 20 nA lorsque désactivés. Un port de la bibliothèque `libopencm3` pour ces processeurs semble être amorcée.

10 Stockage sur carte SD au format FAT : exploitation de EFSL

Nous avions déjà mentionné auparavant les perspectives spectaculaires offertes, pour les systèmes embarqués par la capacité à stocker des informations sur support non-volatile de type Secure Digital (SD), et en particulier la capacité à organiser les données selon des fichiers exploitables par tout utilisateur grâce au format FAT [LM117]. Nous avions proposé la bibliothèque EFSL qui, de par son excellente structuration, ne nécessite que le portage de quelques fonctions bas niveau pour être utilisable sur une nouvelle architecture. Nous avons donc modifié EFSL pour une utilisation sur STM32 avec bibliothèque `libopencm3`. Nous n'exploitons cependant pas les fonctionnalités avancées de cette architecture telles que la capacité à transférer les données par DMA, optimisation qui mériterait à être implémentée pour réduire l'impact du stockage sur l'efficacité du programme principal.

Dans l'arborescence d'EFSL, l'unique fichier contenant la description des accès bas-niveau entre le microcontrôleur et la carte SD communiquant au travers du protocole SPI se trouve dans `efsl/source/interface/efsl_spi.c`. Dans ce fichier, nous redéfinissons, tel que vu auparavant, la direction et la fonction des broches associées à SPI1 (port A) au niveau de `SPI_Config()`, ainsi que les macros définissant le niveau du signal d'activation de la carte (GPIOA 4 dans notre exemple).

```
1#define MSD_CS_LOW()   gpio_clear(GPIOA,GPIO4)
 #define MSD_CS_HIGH()  gpio_set(GPIOA,GPIO4)
```

Les fonctions de communication sur bus SPI sont déjà encapsulées dans `libopencm3` sous forme de `spi_send(SPI1, outgoing);` et `incoming = spi_read(SPI1);`. Rapidement, nous avons ainsi

9. http://www.energymicro.com/news-archive/energy-micro-launches-efm32-tiny-gecko-microcontrollers

la satisfaction d'accéder à un fichier dans un répertoire et d'y stocker des informations dans un format accessible sur tout ordinateur personnel.

```
char ouvre_sd()
{char err;
 err = efs_init(&efs, 0);
 if ((err) != 0) {return (0);}
    else {SD_Present = 1; // SD ok
         //Creation du repertoire
         if (mkdir(&efs.myFs, "GPR") == 0)       // si il n'existe pas
            affiche("# New data directory created.\r\n");
         else
            affiche("# unable to create new dir (already existing ?)\r\n");

// il FAUT utiliser 'a' pour append, sinon erreur quand le fichier existe deja
// et qu'on utilise 'w'
         if (file_fopen(&file, &efs.myFs, "GPR/DATA.TXT", 'a') != 0) {
            if (file_fopen(&file, &efs.myFs, "GPR/DATA.TXT", 'w') != 0)
               SD_Present = 0;
               }
            else affiche("#File Open OK\r\n");
         if (SD_Present == 1)
               {file_fclose(&file);
                fs_umount(&efs.myFs);
               }
         }
 return (SD_Present);
}

main() {
 SD_Present = ouvre_sd();
 if (SD_Present == 1)
    if (file_fopen(&file, &efs.myFs, "GPR/DATA.TXT", 'a') !=0)
       {if (file_fopen(&file, &efs.myFs, "GPR/DATA.TXT",'w') != 0)
           {SD_Present = 0;}
       }
 if (SD_Present == 1)
    {file_write(&file, sizeof(b), &b); // char *b contient le tableau a ecrire
     file_write(&file, 2, "\r\n");
     file_fclose(&file);
     fs_umount(&efs.myFs);
    } else {SD_Present = 0;}
  [...]
}
```

Nous testons dans cet exemple si le fichier de sauvegarde existe déjà en tentant d'y ajouter les informations (mode 'a'). En cas d'échec, le fichier est créé (mode 'w'). Si cette création échoue, il y a eu disparition de la carte depuis son initialisation (`ouvre_sd()`) et nous la marquons comme absente, sinon le contenu du tableau de caractères `b` est transféré sur la carte. Noter que les informations ne sont physiquement écrites que lors du démontage du système de fichiers `fs_umount()`.

Ainsi, une application naturelle qui découle de l'utilisation de l'horloge temps-réel (vue dans la section précédente) et du stockage au format FAT sur carte SD est l'obtention d'un enregistreur autonome capable de stocker des informations d'un instrument réveillé de façon périodique, et ce en l'absence d'intervention des utilisateurs pendant un intervalle de temps défini par l'autonomie des batteries alimentant le circuit. En ce sens, nous avons constaté que toutes les cartes SD ne se valent pas, et tandis que certains modèles passent automatiquement en mode veille une fois la phase d'écriture achevée, les modèles les moins chers continuent à drainer un courant important même en l'absence de toute opération. La façon la plus sûre de réduire la consommation est donc d'alimenter la carte SD au travers d'un régulateur DC-DC avec désactivation (*shutdown*) ou d'un interrupteur analogique.

11 Conclusion

Cet ensemble de mises en œuvre nous a permis de se faire une bonne idée de la manière d'-exploiter le STM32 et de configurer les périphériques. Il est ainsi possible de constater que pour un périphérique donné une partie est totalement liée à celui-ci (activation de l'horloge pour le périphérique et pour le port, configuration des broches, etc...). Une seconde partie de la configuration/utilisation se base sur des registres indexés par l'adresse du périphérique. Cette partie pourra donc être facilement incluse dans un module utilisable par tous les périphériques d'un même type. Nous n'avons pas détaillés dans le présent article l'usage du DMA, ceci est toutefois fait dans l'article `Traitement du signal sur système embarqué -- application au RADAR à onde continue`.

La compréhension de la structure du STM32 est l'étape inévitable pour être en mesure de réaliser le portage d'un exécutif tel que TinyOS sur ce microcontrôleur, ainsi que nous le verrons dans un prochain article.

Il est à noter que les STM32 disposent selon le modèle d'un *USB device* , voir *OTG* sur les hauts de gammes. La `libopencm3` ne propose à l'heure actuelle que l'implémentation pour le mode *device*, la partie *OTG* étant à faire. Des exemples de port-séries virtuels (entre autres) sont fournis avec la `libopencm3`.

Références

[CortexM3] J. Yiu, *The Definitive Guide to the ARM Cortex-M3, 2nd Ed.*, Newnes (2009)

[RM0008] RM0008, Reference manual rev. 13, Mai 2011, disponible à http://www.st.com/internet/com/TECHNICAL_RESOURCES/TECHNICAL_LITERATURE/REFERENCE_MANUAL/CD00171190.pdf

[morse] Page wikipedia concernant l'alphabet morse http://fr.wikipedia.org/wiki/Alphabet_morse

[MS5534] Datasheet du MS5534C (remplaçant du MS5534A) http://www.meas-spec.com/WorkArea/linkit.aspx?LinkIdentifier=id&ItemID=6673

[AN510] Utilisation du MS5534A en SPI http://www.meas-spec.com/downloads/Using_SPI_Protocol_with_Pressure_Sensor_Modules.pdf

[HH10D] Datasheet du capteur d'humidité HH10D http://www.hoperf.com/upload/sensor/HH10D.pdf

[TEMT6000] Datasheet du capteur de lumière TEMT6000 http://www.vishay.com/docs/81579/temt6000.pdf

[MSP430] J.-M. Friedt, A. Masse, F. Bassignot, *Les microcontrôleurs MSP430 pour les applications faibles consommations – asservissement d'un oscillateur sur le GPS*, GNU/Linux Magazine France **98**, Octobre 2007, disponible à http://jmfriedt.free.fr

[LM117] J.-M. Friedt & É. Carry, *Développement sur processeur à base de cœur ARM7 sous GNU/Linux* GNU/LINUX Magazine France **117**, juin 2009, pp.32-51, disponible à http://jmfriedt.free.fr